## CONTENTS

1 Preface  
---  
1.1 About ......................................................... 3  
1.2 Motivation for this tutorial .................................. 4  
1.3 Copyright notice and license info .............................. 4  
1.4 Acknowledgement ............................................... 4  
1.5 Feedback and suggestions ...................................... 5  

2 Why Spark with Python ?  
---  
2.1 Why Spark? ...................................................... 7  
2.2 Why Spark with Python (PySpark)? ............................ 8  

3 Configure Running Platform  
---  
3.1 Run on Databricks Community Cloud ........................... 11  
3.2 Configure Spark on Mac and Ubuntu ............................ 16  
3.3 Configure Spark on Windows .................................. 19  
3.4 PySpark With Text Editor or IDE .............................. 19  
3.5 PySparkling Water: Spark + H2O ............................... 26  
3.6 Set up Spark on Cloud ......................................... 27  
3.7 PySpark on Colaboratory ...................................... 27  
3.8 Demo Code in this Section ..................................... 28  

4 An Introduction to Apache Spark  
---  
4.1 Core Concepts .................................................. 31  
4.2 Spark Components ............................................. 31  
4.3 Architecture ................................................... 34  
4.4 How Spark Works? .............................................. 34  

5 Programming with RDDs  
---  
5.1 Create RDD ..................................................... 35  
5.2 Spark Operations .............................................. 39  
5.3 rdd.DataFrame vs pd.DataFrame ................................ 41  

6 Statistics and Linear Algebra Preliminaries  
---  
6.1 Notations ....................................................... 57  
6.2 Linear Algebra Preliminaries .................................. 57  
6.3 Measurement Formula .......................................... 59
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Confusion Matrix</td>
<td>60</td>
</tr>
<tr>
<td>6.5</td>
<td>Statistical Tests</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>Data Exploration</td>
<td>63</td>
</tr>
<tr>
<td>7.1</td>
<td>Univariate Analysis</td>
<td>63</td>
</tr>
<tr>
<td>7.2</td>
<td>Multivariate Analysis</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>Data Manipulation: Features</td>
<td>85</td>
</tr>
<tr>
<td>8.1</td>
<td>Feature Extraction</td>
<td>85</td>
</tr>
<tr>
<td>8.2</td>
<td>Feature Transform</td>
<td>95</td>
</tr>
<tr>
<td>8.3</td>
<td>Feature Selection</td>
<td>114</td>
</tr>
<tr>
<td>8.4</td>
<td>Unbalanced data: Undersampling</td>
<td>115</td>
</tr>
<tr>
<td>9</td>
<td>Regression</td>
<td>117</td>
</tr>
<tr>
<td>9.1</td>
<td>Linear Regression</td>
<td>117</td>
</tr>
<tr>
<td>9.2</td>
<td>Generalized linear regression</td>
<td>131</td>
</tr>
<tr>
<td>9.3</td>
<td>Decision tree Regression</td>
<td>140</td>
</tr>
<tr>
<td>9.4</td>
<td>Random Forest Regression</td>
<td>148</td>
</tr>
<tr>
<td>9.5</td>
<td>Gradient-boosted tree regression</td>
<td>156</td>
</tr>
<tr>
<td>10</td>
<td>Regularization</td>
<td>165</td>
</tr>
<tr>
<td>10.1</td>
<td>Ordinary least squares regression</td>
<td>165</td>
</tr>
<tr>
<td>10.2</td>
<td>Ridge regression</td>
<td>165</td>
</tr>
<tr>
<td>10.3</td>
<td>Least Absolute Shrinkage and Selection Operator (LASSO)</td>
<td>166</td>
</tr>
<tr>
<td>10.4</td>
<td>Elastic net</td>
<td>166</td>
</tr>
<tr>
<td>11</td>
<td>Classification</td>
<td>167</td>
</tr>
<tr>
<td>11.1</td>
<td>Binomial logistic regression</td>
<td>167</td>
</tr>
<tr>
<td>11.2</td>
<td>Multinomial logistic regression</td>
<td>179</td>
</tr>
<tr>
<td>11.3</td>
<td>Decision tree Classification</td>
<td>192</td>
</tr>
<tr>
<td>11.4</td>
<td>Random forest Classification</td>
<td>204</td>
</tr>
<tr>
<td>11.5</td>
<td>Gradient-boosted tree Classification</td>
<td>214</td>
</tr>
<tr>
<td>11.6</td>
<td>XGBoost: Gradient-boosted tree Classification</td>
<td>215</td>
</tr>
<tr>
<td>11.7</td>
<td>Naive Bayes Classification</td>
<td>217</td>
</tr>
<tr>
<td>12</td>
<td>Clustering</td>
<td>231</td>
</tr>
<tr>
<td>12.1</td>
<td>K-Means Model</td>
<td>231</td>
</tr>
<tr>
<td>13</td>
<td>RFM Analysis</td>
<td>245</td>
</tr>
<tr>
<td>13.1</td>
<td>RFM Analysis Methodology</td>
<td>246</td>
</tr>
<tr>
<td>13.2</td>
<td>Demo</td>
<td>248</td>
</tr>
<tr>
<td>13.3</td>
<td>Extension</td>
<td>254</td>
</tr>
<tr>
<td>14</td>
<td>Text Mining</td>
<td>261</td>
</tr>
<tr>
<td>14.1</td>
<td>Text Collection</td>
<td>261</td>
</tr>
<tr>
<td>14.2</td>
<td>Text Preprocessing</td>
<td>269</td>
</tr>
<tr>
<td>14.3</td>
<td>Text Classification</td>
<td>271</td>
</tr>
<tr>
<td>14.4</td>
<td>Sentiment analysis</td>
<td>278</td>
</tr>
<tr>
<td>14.5</td>
<td>N-grams and Correlations</td>
<td>285</td>
</tr>
<tr>
<td>Chapter 25: PySpark API</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>25.1 Stat API</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>25.2 Regression API</td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>25.3 Classification API</td>
<td>406</td>
<td></td>
</tr>
<tr>
<td>25.4 Clustering API</td>
<td>429</td>
<td></td>
</tr>
<tr>
<td>25.5 Recommendation API</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td>25.6 Pipeline API</td>
<td>450</td>
<td></td>
</tr>
<tr>
<td>25.7 Tuning API</td>
<td>452</td>
<td></td>
</tr>
<tr>
<td>25.8 Evaluation API</td>
<td>457</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 26: Main Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>465</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>467</td>
</tr>
<tr>
<td>Index</td>
<td>469</td>
</tr>
</tbody>
</table>
Welcome to my Learning Apache Spark with Python note! In this note, you will learn a wide array of concepts about PySpark in Data Mining, Text Mining, Machine Learning and Deep Learning. The PDF version can be downloaded from HERE.
1.1 About

1.1.1 About this note

This is a shared repository for Learning Apache Spark Notes. The PDF version can be downloaded from HERE. The first version was posted on Github in ChenFeng ([Feng2017]). This shared repository mainly contains the self-learning and self-teaching notes from Wenqiang during his IMA Data Science Fellowship. The reader is referred to the repository https://github.com/runawayhorse001/LearningApacheSpark for more details about the dataset and the .ipynb files.

In this repository, I try to use the detailed demo code and examples to show how to use each main functions. If you find your work wasn’t cited in this note, please feel free to let me know.

Although I am by no means an data mining programming and Big Data expert, I decided that it would be useful for me to share what I learned about PySpark programming in the form of easy tutorials with detailed example. I hope those tutorials will be a valuable tool for your studies.

The tutorials assume that the reader has a preliminary knowledge of programming and Linux. And this document is generated automatically by using sphinx.

1.1.2 About the authors

- **Wenqiang Feng**
  - Sr. Data Scientist and PhD in Mathematics
  - University of Tennessee at Knoxville
  - Email: von198@gmail.com

- **Biography**

  Wenqiang Feng is a Sr. Data Scientist at Machine Learning Lab, H&R Block. Before joining Block, Dr. Feng is a Data Scientist at Applied Analytics Group, DST (now SS&C). Dr. Feng’s responsibilities include providing clients with access to cutting-edge skills and technologies, including Big Data analytic solutions, advanced analytic and data enhancement techniques and modeling.
Dr. Feng has deep analytic expertise in data mining, analytic systems, machine learning algorithms, business intelligence, and applying Big Data tools to strategically solve industry problems in a cross-functional business. Before joining DST, Dr. Feng was an IMA Data Science Fellow at The Institute for Mathematics and its Applications (IMA) at the University of Minnesota. While there, he helped startup companies make marketing decisions based on deep predictive analytics.

Dr. Feng graduated from University of Tennessee, Knoxville, with Ph.D. in Computational Mathematics and Master’s degree in Statistics. He also holds Master’s degree in Computational Mathematics from Missouri University of Science and Technology (MST) and Master’s degree in Applied Mathematics from the University of Science and Technology of China (USTC).

• Declaration

The work of Wenqiang Feng was supported by the IMA, while working at IMA. However, any opinion, finding, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the IMA, UTK, DST and HR & Block.

1.2 Motivation for this tutorial

I was motivated by the IMA Data Science Fellowship project to learn PySpark. After that I was impressed and attracted by the PySpark. And I found that:

1. It is no exaggeration to say that Spark is the most powerful Bigdata tool.

2. However, I still found that learning Spark was a difficult process. I have to Google it and identify which one is true. And it was hard to find detailed examples which I can easily learned the full process in one file.

3. Good sources are expensive for a graduate student.

1.3 Copyright notice and license info

This Learning Apache Spark with Python PDF file is supposed to be a free and living document, which is why its source is available online at https://runawayhorse001.github.io/LearningApacheSpark/pyspark.pdf. But this document is licensed according to both MIT License and Creative Commons Attribution-NonCommercial 2.0 Generic (CC BY-NC 2.0) License.

When you plan to use, copy, modify, merge, publish, distribute or sublicense, Please see the terms of those licenses for more details and give the corresponding credits to the author.

1.4 Acknowledgement

At here, I would like to thank Ming Chen, Jian Sun and Zhongbo Li at the University of Tennessee at Knoxville for the valuable discussion and thank the generous anonymous authors for providing the detailed solutions and source code on the internet. Without those help, this repository would not have been possible to be made. Wenqiang also would like to thank the Institute for Mathematics and Its Applications (IMA) at
University of Minnesota, Twin Cities for support during his IMA Data Scientist Fellow visit and thank TAN THIAM HUAT and Mark Rabins for finding the typos.

A special thank you goes to Dr. Haiping Lu, Lecturer in Machine Learning at Department of Computer Science, University of Sheffield, for recommending and heavily using my tutorial in his teaching class and for the valuable suggestions.

1.5 Feedback and suggestions

Your comments and suggestions are highly appreciated. I am more than happy to receive corrections, suggestions or feedbacks through email (von198@gmail.com) for improvements.
WHY SPARK WITH PYTHON?

Chinese proverb
Sharpening the knife longer can make it easier to hack the firewood – old Chinese proverb

I want to answer this question from the following two parts:

2.1 Why Spark?

I think the following four main reasons from Apache Spark™ official website are good enough to convince you to use Spark.

1. Speed
   Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
   Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

   ![Fig. 1: Logistic regression in Hadoop and Spark](image)

2. Ease of Use
   Write applications quickly in Java, Scala, Python, R.
Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

3. Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

![Spark Stack Diagram](image)

Fig. 2: The Spark stack

4. Runs Everywhere

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

2.2 Why Spark with Python (PySpark)?

No matter you like it or not, Python has been one of the most popular programming languages.
2.2. Why Spark with Python (PySpark)?

Fig. 3: The Spark platform
Fig. 4: KDnuggets Analytics/Data Science 2017 Software Poll from kdnuggets.
Chinese proverb

Good tools are prerequisite to the successful execution of a job. – old Chinese proverb

A good programming platform can save you lots of troubles and time. Herein I will only present how to install my favorite programming platform and only show the easiest way which I know to set it up on Linux system. If you want to install on the other operator system, you can Google it. In this section, you may learn how to set up Pyspark on the corresponding programming platform and package.

3.1 Run on Databricks Community Cloud

If you don’t have any experience with Linux or Unix operator system, I would love to recommend you to use Spark on Databricks Community Cloud. Since you do not need to setup the Spark and it’s totally free for Community Edition. Please follow the steps listed below.

1. Sign up a account at: https://community.cloud.databricks.com/login.html
2. Sign in with your account, then you can creat your cluster(machine), table(dataset) and notebook(code).
3. Create your cluster where your code will run
4. Import your dataset

Note: You need to save the path which appears at Uploaded to DBFS: /File-Store/tables/05rmhuqv1489687378010/. Since we will use this path to load the dataset.
5. Create your notebook
3.1. Run on Databricks Community Cloud
Chapter 3. Configure Running Platform
3.1. Run on Databricks Community Cloud

### Linear Regression with PySpark on Databricks

**Author:** Wenqiang Feng

#### Set up SparkSession

```python
from pyspark.sql import SparkSession
spark = SparkSession(
    .builder
    .appName("Python Spark Linear Regression Example")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)
```

#### Load dataset

```python
def load():
    return spark.read.format("com.databricks.spark.csv").
        options(header='true', inferSchema='true').
        load("/FileStore/tables/95formulp/189687379810/", header= True)
```
After finishing the above 5 steps, you are ready to run your Spark code on Databricks Community Cloud. I will run all the following demos on Databricks Community Cloud. Hopefully, when you run the demo code, you will get the following results:

<table>
<thead>
<tr>
<th>_c0</th>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
<td>22.1</td>
</tr>
<tr>
<td>2</td>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
<td>10.4</td>
</tr>
<tr>
<td>3</td>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
<td>9.3</td>
</tr>
<tr>
<td>4</td>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
<td>18.5</td>
</tr>
<tr>
<td>5</td>
<td>180.8</td>
<td>10.8</td>
<td>58.4</td>
<td>12.9</td>
</tr>
</tbody>
</table>

Only showing top 5 rows

root
|-- _c0: integer (nullable = true)
|-- TV: double (nullable = true)
|-- Radio: double (nullable = true)
|-- Newspaper: double (nullable = true)
|-- Sales: double (nullable = true)

3.2 Configure Spark on Mac and Ubuntu

3.2.1 Installing Prerequisites

I will strongly recommend you to install Anaconda, since it contains most of the prerequisites and support multiple Operator Systems.

1. Install Python

Go to Ubuntu Software Center and follow the following steps:

a. Open Ubuntu Software Center
b. Search for python
c. And click Install

Or Open your terminal and using the following command:

```
sudo apt-get install build-essential checkinstall
dsuo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
sudo apt-get install python
sudo easy_install pip
sudo pip install ipython
```
3.2.2 Install Java

Java is used by many other softwares. So it is quite possible that you have already installed it. You can by using the following command in Command Prompt:

```
java -version
```

Otherwise, you can follow the steps in How do I install Java for my Mac? to install java on Mac and use the following command in Command Prompt to install on Ubuntu:

```
sudo apt-add-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

3.2.3 Install Java SE Runtime Environment

I installed ORACLE Java JDK.

---

Warning: Installing Java and Java SE Runtime Environment steps are very important, since Spark is a domain-specific language written in Java.

---

You can check if your Java is available and find it’s version by using the following command in Command Prompt:

```
java -version
```

If your Java is installed successfully, you will get the similar results as follows:

```
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

3.2.4 Install Apache Spark

Actually, the Pre-build version doesn’t need installation. You can use it when you unpack it.

   a. Download: You can get the Pre-built Apache Spark™ from Download Apache Spark™.
   b. Unpack: Unpack the Apache Spark™ to the path where you want to install the Spark.
   c. Test: Test the Prerequisites: change the direction `spark-#.#.#-bin-hadoop#.#/bin` and run

   ```
   ./pyspark
   ```

3.2. Configure Spark on Mac and Ubuntu
Learning Apache Spark with Python

Python 2.7.13 |Anaconda 4.4.0 (x86_64)| (default, Dec 20 2016, 23:05:08)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/08/30 13:30:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/08/30 13:30:17 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Welcome to
  ___/_\_ _ \____/ /___
 / \ \ /_ \_ / \_ /\_ / _/ \_
/___ /.__/._/_/._/._/._/._/_/ version 2.1.1
/_/
Using Python version 2.7.13 (default, Dec 20 2016 23:05:08)
SparkSession available as 'spark'.

3.2.5 Configure the Spark

a. Mac Operator System: open your bash_profile in Terminal

```bash
vim ~/.bash_profile
```
And add the following lines to your bash_profile (remember to change the path)

```bash
# add for spark
export SPARK_HOME=your_spark_installation_path
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
```
At last, remember to source your bash_profile

```bash
source ~/.bash_profile
```

b. Ubuntu Operator System: open your bashrc in Terminal

```bash
vim ~/.bashrc
```
And add the following lines to your `bashrc` (remember to change the path)

```bash
# add for spark
export SPARK_HOME=your_spark_installation_path
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_DRIVE_PYTHON="jupyter"
export PYSPARK_DRIVE_PYTHON_OPTS="notebook"
```

At last, remember to source your `bashrc`

```
source ~/.bashrc
```

### 3.3 Configure Spark on Windows

Installing open source software on Windows is always a nightmare for me. Thanks for Deelesh Mandloi. You can follow the detailed procedures in the blog [Getting Started with PySpark on Windows](#) to install the Apache Spark™ on your Windows Operator System.

### 3.4 PySpark With Text Editor or IDE

#### 3.4.1 PySpark With Jupyter Notebook

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to write and run your PySpark Code in Jupyter notebook.

#### 3.4.2 PySpark With PyCharm

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to add the PySpark to your PyCharm project.

1. Create a new PyCharm project
2. Go to Project Structure
   - Option 1: File -> Settings -> Project: -> Project Structure
   - Option 2: PyCharm -> Preferences -> Project: -> Project Structure
3. Add Content Root: all ZIP files from `$SPARK_HOME/python/lib`
4. Run your script
Chapter 3. Configure Running Platform
3.4. PySpark With Text Editor or IDE
Chapter 3. Configure Running Platform
3.4.3 PySpark With Apache Zeppelin

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to write and run your PySpark Code in Apache Zeppelin.

![PySpark with Apache Zeppelin](image)

3.4.4 PySpark With Sublime Text

After you finishing the above setup steps in *Configure Spark on Mac and Ubuntu*, then you should be good to use Sublime Text to write your PySpark Code and run your code as a normal python code in Terminal.

```
python test_pyspark.py
```

Then you should get the output results in your terminal.
3.4.5 PySpark With Eclipse

If you want to run PySpark code on Eclipse, you need to add the paths for the External Libraries for your Current Project as follows:

1. Open the properties of your project

2. Add the paths for the External Libraries
3.4. PySpark With Text Editor or IDE
And then you should be good to run your code on Eclipse with PyDev.

### 3.5 PySparkling Water: Spark + H2O

1. **Download Sparkling Water from:** [https://s3.amazonaws.com/h2o-release/sparkling-water/rel-2.4.5/index.html](https://s3.amazonaws.com/h2o-release/sparkling-water/rel-2.4.5/index.html)

2. **Test PySparkling**

   ```
   unzip sparkling-water-2.4.5.zip
cd ~/sparkling-water-2.4.5/bin
./pysparkling
   ```

If you have a correct setup for PySpark, then you will get the following results:

```
Using Spark defined in the SPARK_HOME=/Users/dt216661/spark environmental property

Python 3.7.1 (default, Dec 14 2018, 13:28:58) [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
2019-02-15 14:08:30 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```
Setting default log level to "WARN".
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/08/30 13:30:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/08/30 13:30:17 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Welcome to 
\____\_ __ ______/ /__
\_/ _ \_ \_ \ \__ __ \/_/ /_/ \
\_/ / \_\_\_\_\_/\_\_\_\_\_\_
\_\_\_\_/\_\_/\_\_/\_\_/\_\_/\_\_\_/ version 2.4.0
SparkSession available as 'spark'.

3. Setup pysparkling with Jupyter notebook
Add the following alias to your bashrc (Linux systems) or bash_profile (Mac system)
```bash
alias sparkling="PYSPARK_DRIVER_PYTHON="ipython" PYSPARK_DRIVER_PYTHON_OPTS="notebook" /~/sparkling-water-2.4.5/bin/pysparkling"
```

4. Open pysparkling in terminal
```
sparkling
```

3.6 Set up Spark on Cloud

Following the setup steps in *Configure Spark on Mac and Ubuntu*, you can set up your own cluster on the cloud, for example AWS, Google Cloud. Actually, for those clouds, they have their own Big Data tool. You can run them directly without any setting just like Databricks Community Cloud. If you want more details, please feel free to contact with me.

3.7 PySpark on Colaboratory

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud.
3.7.1 Installation

!pip install pyspark

3.7.2 Testing

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext \
    .parallelize([(1, 2, 3, 'a b c'), 
                   (4, 5, 6, 'd e f'), 
                   (7, 8, 9, 'g h i')]) \
    .toDF(['col1', 'col2', 'col3', 'col4'])

df.show()
```

Output:

```
+----+----+----+-----+  
|col1|col2|col3|col4|  
+----+----+----+-----+  
 | 1| 2| 3|a b c|  
 | 4| 5| 6|d e f|  
 | 7| 8| 9|g h i|  
+----+----+----+-----+  
```

3.8 Demo Code in this Section

The Jupyter notebook can be download from installation on colab.

- Python Source code

```python
## set up SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv') \
    .load('example.csv')
```

(continues on next page)
options(header='true', \  
inferschema='true').\  
load("/home/feng/Spark/Code/data/Advertising.csv
˓→", header=True)

df.show(5)
df.printSchema()
4.1 Core Concepts

Most of the following content comes from [Kirillov2016]. So the copyright belongs to Anton Kirillov. I will refer you to get more details from Apache Spark core concepts, architecture and internals.

Before diving deep into how Apache Spark works, let's understand the jargon of Apache Spark:

- **Job**: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.

- **Stages**: Jobs are divided into stages. Stages are classified as a Map or reduce stages (It’s easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be updated in a single Stage. It happens over many stages.

- **Tasks**: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).

- **DAG**: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.

- **Executor**: The process responsible for executing a task.

- **Master**: The machine on which the Driver program runs

- **Slave**: The machine on which the Executor program runs

4.2 Spark Components

1. Spark Driver
Learning Apache Spark with Python

• separate process to execute user applications
• creates SparkContext to schedule jobs execution and negotiate with cluster manager

2. Executors
• run tasks scheduled by driver
• store computation results in memory, on disk or off-heap
• interact with storage systems

3. Cluster Manager
• Mesos
• YARN
• Spark Standalone

Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:

• SparkContext
  – represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster

• DAGScheduler
  – computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs

• TaskScheduler
  – responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers

• SchedulerBackend

4.2. Spark Components
– backend interface for scheduling systems that allows plugging in different implementations (Mesos, YARN, Standalone, local)

- BlockManager
  – provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

### 4.3 Architecture

### 4.4 How Spark Works?

Spark has a small code base and the system is divided in various layers. Each layer has some responsibilities. The layers are independent of each other.

The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications. As you enter your code in spark console (creating RDD’s and applying operators), Spark creates a operator graph. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage. This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn’t know about dependencies among stages.

![Diagram of Spark Architecture](image-url)
CHAPTER
FIVE

PROGRAMMING WITH RDDS

Chinese proverb

If you only know yourself, but not your opponent, you may win or may lose. If you know neither
yourself nor your enemy, you will always endanger yourself – idiom, from Sunzi’s Art of War

RDD represents Resilient Distributed Dataset. An RDD in Spark is simply an immutable distributed
collection of objects sets. Each RDD is split into multiple partitions (similar pattern with smaller sets),
which may be computed on different nodes of the cluster.

5.1 Create RDD

Usually, there are two popular ways to create the RDDs: loading an external dataset, or distributing a
set of collection of objects. The following examples show some simplest ways to create RDDs by using
parallelize() function which takes an already existing collection in your program and pass the same
to the Spark Context.

1. By using parallelize( ) function

```python
from pyspark.sql import SparkSession

spark = SparkSession \
.builder \
 .appName("Python Spark create RDD example") \
 .config("spark.some.config.option", "some-value") \
 .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
                                      (4, 5, 6, 'd e f'),
                                      (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3', 'col4'])
```

Then you will get the RDD data:

```python
df.show()
+----+----+----+-----+
(continues on next page)
```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

myData = spark.sparkContext.parallelize([(1,2), (3,4), (5,6), (7,8), (9,10)])

Then you will get the RDD data:

myData.collect()

[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]

2. By using createDataFrame( ) function

from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

Employee = spark.createDataFrame([  
    ('1', 'Joe', '70000', '1'),  
    ('2', 'Henry', '80000', '2'),  
    ('3', 'Sam', '60000', '2'),  
    ('4', 'Max', '90000', '1')],  
    ['Id', 'Name', 'Sallary', 'DepartmentId'])

Then you will get the RDD data:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Name</td>
<td>Sallary</td>
<td>DepartmentId</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>Joe</td>
<td>70000</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Henry</td>
<td>80000</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Sam</td>
<td>60000</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Max</td>
<td>90000</td>
<td>1</td>
</tr>
</tbody>
</table>
3. By using read and load functions
   a. **Read dataset from .csv file**

   ```python
   ## set up SparkSession
   from pyspark.sql import SparkSession

   spark = SparkSession \
     .builder \
     .appName("Python Spark create RDD example") \
     .config("spark.some.config.option", "some-value") \
     .getOrCreate()

   df = spark.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load("/home/feng/Spark/Code/data/Advertising.csv", header=True)

   df.show(5)
   df.printSchema()
   ```

   Then you will get the RDD data:

<table>
<thead>
<tr>
<th></th>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
<td>22.1</td>
</tr>
<tr>
<td>2</td>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
<td>10.4</td>
</tr>
<tr>
<td>3</td>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
<td>9.3</td>
</tr>
<tr>
<td>4</td>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
<td>18.5</td>
</tr>
<tr>
<td>5</td>
<td>180.8</td>
<td>10.8</td>
<td>58.4</td>
<td>12.9</td>
</tr>
</tbody>
</table>

   only showing top 5 rows

   root
   |-- _c0: integer (nullable = true)
   |-- TV: double (nullable = true)
   |-- Radio: double (nullable = true)
   |-- Newspaper: double (nullable = true)
   |-- Sales: double (nullable = true)

   Once created, RDDs offer two types of operations: transformations and actions.

   b. **Read dataset from Database**

   ```python
   ## set up SparkSession
   from pyspark.sql import SparkSession

   spark = SparkSession \
     .builder \
     .appName("Python Spark create RDD example") \
     .config("spark.some.config.option", "some-value") \
     .getOrCreate()

   df = spark.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load("/home/feng/Spark/Code/data/Advertising.csv", header=True)

   df.show(5)
   df.printSchema()
   ```

(continues on next page)
## User information
user = 'your_username'
pw = 'your_password'

## Database information
table_name = 'table_name'
url = 'jdbc:postgresql://##.###.###.##:5432/dataset?user='+user+'&password='+pw
properties = {'driver': 'org.postgresql.Driver', 'password': pw, 'user': user}
df = spark.read.jdbc(url=url, table=table_name, properties=properties)
df.show(5)
df.printSchema()

Then you will get the RDD data:

```
+---+-----+-----+---------+-----+
|_c0| TV |Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
| 1|230.1| 37.8| 69.2| 22.1|
| 2| 44.5| 39.3| 45.1| 10.4|
| 3| 17.2| 45.9| 69.3| 9.3 |
| 4|151.5|41.3|58.5|18.5|
| 5|180.8|10.8|58.4|12.9|
+---+-----+-----+---------+-----+
```

only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)

Note: Reading tables from Database needs the proper drive for the corresponding Database. For example, the above demo needs org.postgresql.Driver and you need to download it and put it in jars folder of your spark installation path. I download postgresql-42.1.1.jar from the official website and put it in jars folder.

C. Read dataset from HDFS

```python
from pyspark.conf import SparkConf
from pyspark.context import SparkContext
from pyspark.sql import HiveContext
```
Learning Apache Spark with Python

(sc= SparkContext('local','example')
   hc = HiveContext(sc)
   tfl = sc.textFile("hdfs://cdhstltest/user/data/demo.CSV")
   print(tfl.first())

   hc.sql("use intg_cme_w")
   spf = hc.sql("SELECT * FROM spf LIMIT 100")
   print(spf.show(5))

5.2 Spark Operations

**Warning:** All the figures below are from Jeffrey Thompson. The interested reader is referred to pyspark pictures

There are two main types of Spark operations: Transformations and Actions [Karau2015].

![Spark Operations Diagram]

**Note:** Some people defined three types of operations: Transformations, Actions and Shuffles.
5.2.1 Spark Transformations

Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.

5.2.2 Spark Actions

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).
5.3 rdd.DataFrame VS pd.DataFrame

5.3.1 Create DataFrame

1. From List

```python
my_list = [['a', 1, 2], ['b', 2, 3], ['c', 3, 4]]
col_name = ['A', 'B', 'C']

:: Python Code:

# caution for the columns=
pd.DataFrame(my_list, columns=col_name)
#
spark.createDataFrame(my_list, col_name).show()

:: Comparison:

+---+---+---+
| A | B | C |
+---+---+---+
0  a | 1 | 2 |
1  b | 2 | 3 |
2  c | 3 | 4 |
+---+---+---+
```

**Attention:** Pay attention to the parameter `columns=` in `pd.DataFrame`. Since the default value will make the list as rows.

```python
# caution for the columns=
pd.DataFrame(my_list, columns=col_name)
#
pd.DataFrame(my_list, col_name)

:: Comparison:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>c</td>
</tr>
</tbody>
</table>
```

5.3. rdd.DataFrame VS pd.DataFrame
2. From Dict

```python
d = {'A': [0, 1, 0],
     'B': [1, 0, 1],
     'C': [1, 0, 0]}
```

:: Python Code:

```python
pd.DataFrame(d)
```

# Tedious for PySpark

```python
spark.createDataFrame(np.array(list(d.values())).T.tolist(),list(d.keys())).
  → show()
```

:: Comparison:

```
+---+---+---+
| A | B | C |
+---+---+---+
| 0 | 0 | 1 |
| 1 | 1 | 0 |
| 2 | 0 | 1 |
```

5.3.2 Load DataFrame

1. From DataBase

Most of the time, you need to share your code with your colleagues or release your code for Code Review or Quality assurance(QA). You will definitely do not want to have your User Information in the code. So you can save them in login.txt:

```
runawayhorse001
PythonTips
```

and use the following code to import your User Information:

```python
#User Information
try:
    login = pd.read_csv(r'login.txt', header=None)
    user = login[0][0]
    pw = login[0][1]
    print('User information is ready!')
except:
    print('Login information is not available!!!')
```

#Database information

```python
host = '##.###.###.##'
db_name = 'db_name'
table_name = 'table_name'
```
conn = psycopg2.connect(host=host, database=db_name, user=user, password=pw)
cur = conn.cursor()

sql = ""
    select *
    from `{table_name}`
"".format(table_name=table_name)
dp = pd.read_sql(sql, conn)

# connect to database
url = 'jdbc:postgresql://'+host+':5432/'+db_name+'?user='+user+'&password='+pw
properties ={'driver': 'org.postgresql.Driver', 'password': pw,'user': user}
ds = spark.read.jdbc(url=url, table=table_name, properties=properties)

**Attention:** Reading tables from Database with PySpark needs the proper drive for the corresponding Database. For example, the above demo needs org.postgresql.Driver and you need to download it and put it in `jars` folder of your spark installation path. I download postgresql-42.1.1.jar from the official website and put it in jars folder.

2. From `.csv`

:: Comparison:

```
# pd.DataFrame dp: DataFrame pandas
dp = pd.read_csv('Advertising.csv')
# rdd.DataFrame dp: DataFrame spark
ds = spark.read.csv(path='Advertising.csv',
    # sep=',',
    # encoding='UTF-8',
    # comment=None,
    # header=True,
    inferSchema=True)
```

3. From `.json`

Data from: http://api.luftdaten.info/static/v1/data.json

```
dp = pd.read_json("data/data.json")
ds = spark.read.json('data/data.json')
```

:: Python Code:

```
dp[['id','timestamp']].head(4)
#
ds[['id','timestamp']].show(4)
```

:: Comparison:

5.3. rdd.DataFrame VS pd.DataFrame
5.3.3 First n Rows

:: Python Code:

dp.head(4)
#
ds.show(4)

:: Comparison:

<table>
<thead>
<tr>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
<td>22.1</td>
</tr>
<tr>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
<td>10.4</td>
</tr>
<tr>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
<td>9.3</td>
</tr>
<tr>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
<td>18.5</td>
</tr>
</tbody>
</table>

5.3.4 Column Names

:: Python Code:

dp.columns
#
ds.columns

:: Comparison:
5.3.5 Data types

:: Python Code:

dp.dtypes
#
ds.dtypes

:: Comparison:

TV float64 ('TV', 'double'),
Radio float64 ('Radio', 'double'),
Newspaper float64 ('Newspaper', 'double'),
Sales float64 ('Sales', 'double')
dtype: object

5.3.6 Fill Null

my_list = [['male', 1, None], ['female', 2, 3], ['male', 3, 4]]
dp = pd.DataFrame(my_list, columns=['A', 'B', 'C'])
ds = spark.createDataFrame(my_list, ['A', 'B', 'C'])
#
dp.head()
ds.show()

:: Comparison:

+------+---+----+
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>male</td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>female</td>
<td>2</td>
<td>3.0</td>
</tr>
<tr>
<td>male</td>
<td>3</td>
<td>4.0</td>
</tr>
</tbody>
</table>
+------+---+----+

:: Python Code:

dp.fillna(-99)
#
ds.fillna(-99).show()

:: Comparison:

+-------+---+---+
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>male</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>female</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>male</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
+-------+---+---+

(continues on next page)
5.3.7 Replace Values

:: Python Code:

```
# caution: you need to chose specific col
dp.A.replace(['male', 'female'],[1, 0], inplace=True)
dp
#caution: Mixed type replacements are not supported
ds.na.replace(['male','female'],['1','0']).show()
```

:: Comparison:

```
+---+---+----+
| A| B| C|
+---+---+----+
0 1 1 NaN | 1| 1|null|
1 0 2 3.0 | 0| 2| 3|
2 1 3 4.0 | 1| 3| 4|
+---+---+----+
```

5.3.8 Rename Columns

1. Rename all columns

:: Python Code:

```
#
dp.columns = ['a','b','c','d']
dp.head(4)
ds.toDF('a','b','c','d').show(4)
```

:: Comparison:

```
+-----+----+----+----+
| a| b| c| d|
+-----+----+----+----+
0 230.1|37.8|69.2|22.1 |
1 44.5 |39.3|45.1|10.4 |
2 17.2 |45.9|69.3| 9.3 |
3 151.5|41.3|58.5|18.5 |
+-----+----+----+----+
only showing top 4 rows
```
2. Rename one or more columns

```python
mapping = {'Newspaper': 'C', 'Sales': 'D'}

dp.rename(columns=mapping).head(4)

#
new_names = [mapping.get(col, col) for col in ds.columns]
ds.toDF(*new_names).show(4)
```

:: Comparison:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>Radio</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
<td>22.1</td>
</tr>
<tr>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
<td>10.4</td>
</tr>
<tr>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
<td>9.3</td>
</tr>
<tr>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
<td>18.5</td>
</tr>
</tbody>
</table>

Note: You can also use `withColumnRenamed` to rename one column in PySpark.

:: Python Code:

```python
ds.withColumnRenamed('Newspaper', 'Paper').show(4)
```

:: Comparison:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>Radio</td>
<td>Paper</td>
</tr>
<tr>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
</tr>
<tr>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
</tr>
<tr>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
</tr>
<tr>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
</tr>
</tbody>
</table>

only showing top 4 rows

5.3.9 Drop Columns

```python
drop_name = ['Newspaper', 'Sales']
```

:: Python Code:

```
5.3. rdd.DataFrame vs pd.DataFrame
dp.drop(drop_name, axis=1).head(4)
#
ds.drop(\*drop_name).show(4)

:: Comparison:

+-----+-----+---------+-----+
| TV  | Radio| Newspaper| Sales |
+-----+-----+---------+-----+
0 230.1| 37.8|         |      |
1  44.5| 39.3|         |      |
2  17.2| 45.9|         |      |
3 151.5| 41.3|         |      |
+-----+-----+---------+-----+

only showing top 4 rows

5.3.10 Filter

dp = pd.read_csv('Advertising.csv')
#
ds = spark.read.csv(path='Advertising.csv',
header=True,
inferSchema=True)

:: Python Code:

dp[dp.Newspaper<20].head(4)
#
ds[ds.Newspaper<20].show(4)

:: Comparison:

+----------------+----------------+-----+-----+---------+-----+
| TV  | Radio| Newspaper| Sales |
+----------------+----------------+-----+-----+---------+-----+
7 120.2| 19.6|    11.6|  13.2|
8  8.6| 2.1 |     1.0|   4.8|
11 214.7| 24.0|     4.0|  17.4|
13 97.5| 7.6 |     7.2|   9.7|
+----------------+----------------+-----+-----+---------+-----+

only showing top 4 rows

:: Python Code:

dp[(dp.Newspaper<20) & (dp.TV>100)].head(4)
#
ds[(ds.Newspaper<20) & (ds.TV>100)].show(4)

:: Comparison:
### 5.3.11 With New Column

:: Python Code:

```python
dp['tv_norm'] = dp.TV/sum(dp.TV)
dp.head(4)
#
```

:: Comparison:

```plaintext
<table>
<thead>
<tr>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
<th>tv_norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>120.2</td>
<td>19.6</td>
<td>11.6</td>
<td>13.2</td>
</tr>
<tr>
<td>11</td>
<td>214.7</td>
<td>24.0</td>
<td>4.0</td>
<td>17.4</td>
</tr>
<tr>
<td>19</td>
<td>147.3</td>
<td>23.9</td>
<td>19.1</td>
<td>14.6</td>
</tr>
<tr>
<td>25</td>
<td>262.9</td>
<td>3.5</td>
<td>19.5</td>
<td>12.0</td>
</tr>
</tbody>
</table>
```

:: Python Code:

```python
dp['cond'] = dp.apply(lambda c: 1 if ((c.TV>100)&(c.Radio<40)) else 2 if c.Sales> 10 else 3,axis=1)
#
```

:: Comparison:

```plaintext
<table>
<thead>
<tr>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>120.2</td>
<td>19.6</td>
<td>11.6</td>
<td>0.007824</td>
</tr>
<tr>
<td>11</td>
<td>214.7</td>
<td>24.0</td>
<td>4.0</td>
<td>0.001513</td>
</tr>
<tr>
<td>19</td>
<td>147.3</td>
<td>23.9</td>
<td>19.1</td>
<td>5.848649E-4</td>
</tr>
<tr>
<td>25</td>
<td>262.9</td>
<td>3.5</td>
<td>19.5</td>
<td>0.00515157</td>
</tr>
</tbody>
</table>
```

---

### 5.3. rdd.DataFrame VS pd.DataFrame
:: Python Code:

dp['log_tv'] = np.log(dp.TV)
dp.head(4)

```
# import pyspark.sql.functions as F
ds.withColumn('log_tv', F.log(ds.TV)).show(4)
```

:: Comparison:

<table>
<thead>
<tr>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
<th>Sales</th>
<th>log_tv</th>
</tr>
</thead>
<tbody>
<tr>
<td>230.1</td>
<td>37.8</td>
<td>69.2</td>
<td>22.1</td>
<td>5.438514</td>
</tr>
<tr>
<td>44.5</td>
<td>39.3</td>
<td>45.1</td>
<td>10.4</td>
<td>3.795489</td>
</tr>
<tr>
<td>17.2</td>
<td>45.9</td>
<td>69.3</td>
<td>9.3</td>
<td>2.844909</td>
</tr>
<tr>
<td>151.5</td>
<td>41.3</td>
<td>58.5</td>
<td>18.5</td>
<td>5.020586</td>
</tr>
</tbody>
</table>

```

:: Python Code:

dp['tv+10'] = dp.TV.apply(lambda x: x+10)
dp.head(4)

```

```
# ds.withColumn('tv+10', ds.TV+10).show(4)
## 5.3.12 Join

```python
leftp = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
'A0' : ['B0', 'B1', 'B2', 'B3'],
'C0' : ['C0', 'C1', 'C2', 'C3'],
'D0' : ['D0', 'D1', 'D2', 'D3'],
index=[0, 1, 2, 3])
rightp = pd.DataFrame({'A': ['A0', 'A1', 'A6', 'A7'],
'B4' : ['B4', 'B5', 'B6', 'B7'],
'C4' : ['C4', 'C5', 'C6', 'C7'],
'D4' : ['D4', 'D5', 'D6', 'D7'],
index=[4, 5, 6, 7])

lefts = spark.createDataFrame(leftp)
right = spark.createDataFrame(rightp)
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td>D0</td>
<td>4</td>
<td>A0</td>
<td>B4</td>
<td>C4</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>5</td>
<td>A1</td>
<td>B5</td>
<td>C5</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
<td>6</td>
<td>A6</td>
<td>B6</td>
<td>C6</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
<td>7</td>
<td>A7</td>
<td>B7</td>
<td>C7</td>
</tr>
</tbody>
</table>

1. Left Join

:: Python Code:

```python
leftp.merge(rightp, on='A', how='left')
# lefts.join(rights, on='A', how='left')
.orderBy('A', ascending=True).show()
```
2. Right Join

:: Python Code:

```python
leftp.merge(rightp,on='A',how='right')

lefts.join(rights,on='A',how='right').orderBy('A',ascending=True).show()
```

:: Comparison:

```
+---+---+---+---+----+
| A | B | C | D | F |
+---+---+---+---+----+
| G | H |

2 A6 NaN NaN NaN NaN NaN| NaN| NaN| NaN| NaN| NaN| NaN| NaN| NaN|

3. Inner Join

:: Python Code:

```python
leftp.merge(rightp,on='A',how='inner')

# (continues on next page)
```
lefts.join(rights,on='A',how='inner')
    .orderBy('A',ascending=True).show()

:: Comparison:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td>D0</td>
<td>B4</td>
<td>C4</td>
<td>D4</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>B5</td>
<td>C5</td>
<td>D5</td>
</tr>
</tbody>
</table>

---

4. Full Join

:: Python Code:

leftp.merge(rightp,on='A',how='outer')
#
lefts.join(rights,on='A',how='full')
    .orderBy('A',ascending=True).show()

:: Comparison:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td>D0</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>A2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>A3</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>A6</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>A7</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

---

5.3.13 Concat Columns

my_list = [('a', 2, 3),
           ('b', 5, 6),
           ('c', 8, 9),
           ('a', 2, 3),
           ]

(continues on next page)
(('b', 5, 6),
('c', 8, 9)]
col_name = ['col1', 'col2', 'col3']
#
dp = pd.DataFrame(my_list, columns=col_name)
ds = spark.createDataFrame(my_list, schema=col_name)

col1 col2 col3
0 a 2 3
1 b 5 6
2 c 8 9
3 a 2 3
4 b 5 6
5 c 8 9

:: Python Code:

dp['concat'] = dp.apply(lambda x: '%s%s' % (x['col1'], x['col2']), axis=1)
dp
#
ds.withColumn('concat', F.concat('col1', 'col2')).show()

:: Comparison:

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>concat</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 2 3</td>
<td>a 2 3</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b 5 6</td>
<td>b 5 6</td>
<td>b5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c 8 9</td>
<td>c 8 9</td>
<td>c8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 2 3</td>
<td>a 2 3</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b 5 6</td>
<td>b 5 6</td>
<td>b5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c 8 9</td>
<td>c 8 9</td>
<td>c8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3.14 GroupBy

:: Python Code:

dp.groupby(['col1']).agg({'col2': 'min', 'col3': 'mean'})
#
ds.groupBy(['col1']).agg({'col2': 'min', 'col3': 'avg'}).show()

:: Comparison:

<table>
<thead>
<tr>
<th>col2</th>
<th>col3</th>
<th>col1</th>
<th>col1</th>
<th>min(col2)</th>
<th>avg(col3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 2 3</td>
<td>c 8 9</td>
<td>8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
### 5.3.15 Pivot

:: Python Code:

```python
pd.pivot_table(dp, values='col3', index='col1', columns='col2', aggfunc=np.sum)
```

```python
# ds.groupBy(['col1']).pivot('col2').sum('col3').show()
```

:: Comparison:

```
+----+----+----+----+
col2 2 5 8 | col1| 2| 5| 8|
col1 +----+----+----+----+
a 6.0 NaN NaN | c| null| null| 18|
b NaN 12.0 NaN | b| null| 12| null|
c NaN NaN 18.0 | a| 6| null| null|
+----+----+----+----+
```

### 5.3.16 Window

```python
d = {'A':['a','b','c','d'],'B':['m','m','n','n'],'C':[1,2,3,6]}
dp = pd.DataFrame(d)
ds = spark.createDataFrame(dp)

:: Python Code:

```python
dp['rank'] = dp.groupby('B')['C'].rank('dense',ascending=False)

# from pyspark.sql.window import Window
w = Window.partitionBy('B').orderBy(ds.C.desc())
ds = ds.withColumn('rank',F.rank().over(w))
```

:: Comparison:

```
+---+---+---+----+
| A| B| C| rank |
|---+---+---+----+
| 0| a| m| 1| 2.0 |
| 1| b| m| 2| 1.0 |
| 2| c| n| 3| 2.0 |
| 3| d| n| 6| 1.0 |
+---+---+---+----+
```
5.3.17 rank vs dense_rank

```python
d = {'Id': [1, 2, 3, 4, 5, 6],
     'Score': [4.00, 4.00, 3.85, 3.65, 3.65, 3.50]}

data = pd.DataFrame(d)
dp = data.copy()
ds = spark.createDataFrame(data)

# Python Code:
dp['Rank_dense'] = dp['Score'].rank(method='dense', ascending=False)
dp['Rank'] = dp['Score'].rank(method='min', ascending=False)
dp

# import pyspark.sql.functions as F
from pyspark.sql.window import Window
w = Window.orderBy(ds.Score.desc())
ds = ds.withColumn('Rank_spark_dense', F.dense_rank().over(w))
ds = ds.withColumn('Rank_spark', F.rank().over(w))
ds.show()

:: Comparison:
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Score</th>
<th>Rank_dense</th>
<th>Rank_spark_dense</th>
<th>Rank_spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td>1.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>4.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>5.0</td>
<td>3.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>5</td>
<td>6.0</td>
<td>4.0</td>
<td>6.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>
```

Chapter 5. Programming with RDDs
STATISTICS AND LINEAR ALGEBRA PRELIMINARIES

Chinese proverb

If you only know yourself, but not your opponent, you may win or may lose. If you know neither yourself nor your enemy, you will always endanger yourself – idiom, from Sunzi’s Art of War

6.1 Notations

• \( m \) : the number of the samples
• \( n \) : the number of the features
• \( y_i \) : i-th label
• \( \hat{y}_i \) : i-th predicted label
• \( \bar{y} = \frac{1}{m} \sum_{i=1}^{m} y_i \) : the mean of \( y \).
• \( y \) : the label vector.
• \( \hat{y} \) : the predicted label vector.

6.2 Linear Algebra Preliminaries

Since I have documented the Linear Algebra Preliminaries in my Prelim Exam note for Numerical Analysis, the interested reader is referred to [Feng2014] for more details (Figure. Linear Algebra Preliminaries).
1 Preliminaries

1.1 Linear Algebra Preliminaries

1.1.1 Common Properties

**Properties 1.1. (Structure of Matrices)** Let $A = [A_{ij}]$ be a square or rectangular matrix, $A$ is called

- **diagonal** if $a_{ij} = 0$, $\forall i \neq j$.
- **upper triangular** if $a_{ij} = 0$, $\forall i > j$.
- **upper Hessenberg** if $a_{ij} = 0$, $\forall i > j + 1$.
- **block diagonal** $A = \text{diag}(A_{11}, A_{22}, \ldots, A_{nn})$.
- **tridiagonal** if $a_{ij} = 0$, $\forall |i - j| > 1$.
- **lower triangular** if $a_{ij} = 0$, $\forall i < j$.
- **lower Hessenberg** if $a_{ij} = 0$, $\forall j > i + 1$.
- **block diagonal** $A = \text{diag}(A_{i-1,i}, A_{ii}, \ldots, A_{i,i+1})$.

**Properties 1.2. (Type of Matrices)** Let $A = [A_{ij}]$ be a square or rectangular matrix, $A$ is called

- **Hermitian** if $A^* = A$.
- **symmetric** if $A^T = A$.
- **skew symmetric** if $A^T = -A$.
- **normal** if $A^*A = AA^*$, when $A \in \mathbb{R}^{m \times n}$.
- **orthogonal** if $A^*A = I$, when $A \in \mathbb{R}^{m \times n}$.
- **unitary** if $A^*A = I$, when $A \in \mathbb{C}^{m \times n}$.

**Properties 1.3. (Properties of invertible matrices)** Let $A$ be an $n \times n$ square matrix. If $A$ is invertible, then

- $\det(A) \neq 0$.
- $\text{rank}(A) = n$.
- $\lambda_i \neq 0$, ($\lambda_i$ eigenvalues).
- $Ax = b$ has a unique solution for every $b \in \mathbb{R}^n$.
- The row vectors are linearly independent.
- $A^*A = AA^*$, when $A \in \mathbb{C}^{m \times n}$.
- The column vectors are linearly independent.
- The row vectors of $A$ form a basis for $\mathbb{R}^n$.
- The column vectors of $A$ form a basis for $\mathbb{R}^n$.
- The column vectors of $A$ span $\mathbb{R}^n$.

**Properties 1.4. (Properties of conjugate transpose)** Let $A, B$ be an $n \times n$ square matrix and $\gamma$ be a complex constant, then

- $(A^*)^\gamma = A^\gamma$.
- $(AB)^* = B^*A^*$.
- $(A + B)^* = A^* + B^*$.
- $\det(A^*) = \det(A)$.
- $\text{trace}(A^*) = \text{trace}(A)$.
- $\gamma A^* = \gamma^* A^\gamma$.

**Properties 1.5. (Properties of similar matrices)** If $A \sim B$, then

- $\det(A) = \det(B)$.
- $\text{rank}(A) = \text{rank}(B)$.
- $\text{eig}(A) = \text{eig}(B)$.
- If $B \sim C$, then $A \sim C$.
- $A \sim A$, $B \sim B - A$.

Fig. 1: Linear Algebra Preliminaries
6.3 Measurement Formula

6.3.1 Mean absolute error

In statistics, **MAE** (Mean absolute error) is a measure of difference between two continuous variables. The Mean Absolute Error is given by:

\[
MAE = \frac{1}{m} \sum_{i=1}^{m} |\hat{y}_i - y_i|.
\]

6.3.2 Mean squared error

In statistics, the **MSE** (Mean Squared Error) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors or deviations—that is, the difference between the estimator and what is estimated.

\[
MSE = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2
\]

6.3.3 Root Mean squared error

\[
RMSE = \sqrt{MSE} = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2}
\]

6.3.4 Total sum of squares

In statistical data analysis the **TSS** (Total Sum of Squares) is a quantity that appears as part of a standard way of presenting results of such analyses. It is defined as being the sum, over all observations, of the squared differences of each observation from the overall mean.

\[
TSS = \sum_{i=1}^{m} (y_i - \bar{y})^2
\]

6.3.5 Explained Sum of Squares

In statistics, the **ESS** (Explained sum of squares), alternatively known as the model sum of squares or sum of squares due to regression.

The ESS is the sum of the squares of the differences of the predicted values and the mean value of the response variable which is given by:

\[
ESS = \sum_{i=1}^{m} (\hat{y}_i - \bar{y})^2
\]
6.3.6 Residual Sum of Squares

In statistics, RSS (Residual sum of squares), also known as the sum of squared residuals (SSR) or the sum of squared errors of prediction (SSE), is the sum of the squares of residuals which is given by:

\[
RSS = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2
\]

6.3.7 Coefficient of determination \( R^2 \)

\[
R^2 := \frac{ESS}{TSS} = 1 - \frac{RSS}{TSS}.
\]

Note: In general \((y^T \bar{y} = \hat{y}^T \hat{y})\), total sum of squares = explained sum of squares + residual sum of squares, i.e.:

\[
TSS = ESS + RSS \text{ if and only if } y^T \bar{y} = \hat{y}^T \hat{y}.
\]

More details can be found at Partitioning in the general ordinary least squares model.

6.4 Confusion Matrix

Fig. 2: Confusion Matrix

6.4.1 Recall

\[
Recall = \frac{TP}{TP + FN}
\]
6.4.2 Precision

\[
\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}
\]

6.4.3 Accuracy

\[
\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total}}
\]

6.4.4 $F_1$-score

\[
F_1 = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}
\]

6.5 Statistical Tests

6.5.1 Correlational Test

- Pearson correlation: Tests for the strength of the association between two continuous variables.
- Spearman correlation: Tests for the strength of the association between two ordinal variables (does not rely on the assumption of normal distributed data).
- Chi-square: Tests for the strength of the association between two categorical variables.

6.5.2 Comparison of Means test

- Paired T-test: Tests for difference between two related variables.
- Independent T-test: Tests for difference between two independent variables.
- ANOVA: Tests the difference between group means after any other variance in the outcome variable is accounted for.

6.5.3 Non-parametric Test

- Wilcoxon rank-sum test: Tests for difference between two independent variables - takes into account magnitude and direction of difference.
- Wilcoxon sign-rank test: Tests for difference between two related variables - takes into account magnitude and direction of difference.
- Sign test: Tests if two related variables are different – ignores magnitude of change, only takes into account direction.
Chinese proverb

A journey of a thousand miles begins with a single step – idiom, from Laozi.

I wouldn’t say that understanding your dataset is the most difficult thing in data science, but it is really important and time-consuming. Data Exploration is about describing the data by means of statistical and visualization techniques. We explore data in order to understand the features and bring important features to our models.

### 7.1 Univariate Analysis

In mathematics, univariate refers to an expression, equation, function or polynomial of only one variable. “Uni” means “one”, so in other words your data has only one variable. So you do not need to deal with the causes or relationships in this step. Univariate analysis takes data, summarizes that variables (attributes) one by one and finds patterns in the data.

There are many ways that can describe patterns found in univariate data include central tendency (mean, mode and median) and dispersion: range, variance, maximum, minimum, quartiles (including the interquartile range), coefficient of variation and standard deviation. You also have several options for visualizing and describing data with univariate data. Such as frequency Distribution Tables, bar Charts, histograms, frequency Polygons, pie Charts.

The variable could be either categorical or numerical, I will demonstrate the different statistical and visualization techniques to investigate each type of the variable.

- The Jupyter notebook can be download from Data Exploration.
- The data can be download from German Credit.

#### 7.1.1 Numerical Variables

**Describe**

The describe function in pandas and spark will give us most of the statistical results, such as min, median, max, quartiles and standard deviation. With the help of the user defined function,
Learning Apache Spark with Python

you can get even more statistical results.

```python
# selected varables for the demonstration
num_cols = ['Account Balance', 'No of dependents']
df.select(num_cols).describe().show()
```

+-------+------------------+-------------------+
|summary| Account Balance| No of dependents|
+-------+------------------+-------------------+
| count | 1000| 1000| |
| mean  | 2.577| 1.155| |
| stddev|1.2576377271108936|0.36208577175319395| |
| min   | 1| 1| |
| max   | 4| 2| |
+-------+------------------+-------------------+

You may find out that the default function in PySpark does not include the quartiles. The following function will help you to get the same results in Pandas

```python
def describe_pd(df_in, columns, deciles=False):
    '''
    Function to union the basic stats results and deciles
    :param df_in: the input dataframe
    :param columns: the column name list of the numerical variable
    :param deciles: the deciles output
    :return : the numerical describe info. of the input dataframe
    
    :author: Ming Chen and Wenqiang Feng
    :email: von198@gmail.com
    '''

    if deciles:
        percentiles = np.array(range(0, 110, 10))
    else:
        percentiles = [25, 50, 75]

    percs = np.transpose([np.percentile(df_in.select(x).collect(), percentiles) for x in columns])

    percs = pd.DataFrame(percs, columns=columns)
    percs['summary'] = [str(p) + '%' for p in percentiles]

    spark_describe = df_in.describe().toPandas()
    new_df = pd.concat([spark_describe, percs], ignore_index=True)
    new_df = new_df.round(2)
    return new_df[['summary'] + columns]
```

describe_pd(df, num_cols)

+-------+------------------+-----------------+
|summary| Account Balance| No of dependents|
+-------+------------------+-----------------+

(continues on next page)
Sometimes, because of the confidential data issues, you can not deliver the real data and your clients may ask more statistical results, such as deciles. You can apply the following function to achieve it.

```python
describe_pd(df, num_cols, deciles=True)
```

<table>
<thead>
<tr>
<th>summary</th>
<th>Account Balance</th>
<th>No of dependents</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>1000.0</td>
<td>1000.0</td>
</tr>
<tr>
<td>mean</td>
<td>2.577</td>
<td>1.155</td>
</tr>
<tr>
<td>stddev</td>
<td>1.2576377271108936</td>
<td>0.362085771753194</td>
</tr>
<tr>
<td>min</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>max</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0%</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10%</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>20%</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>30%</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>40%</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>50%</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>60%</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>70%</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>80%</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>90%</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>100%</td>
<td>4.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Skewness and Kurtosis**

This subsection comes from Wikipedia Skewness.

In probability theory and statistics, skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive or negative, or undefined. For a unimodal distribution, negative skew commonly indicates that the tail is on the left side of the distribution, and positive skew indicates that the tail is on the right.

Consider the two distributions in the figure just below. Within each graph, the values on the right side of the distribution taper differently from the values on the left side. These tapering sides are called tails, and they provide a visual means to determine which of the two kinds of skewness a distribution has:

1. negative skew: The left tail is longer; the mass of the distribution is concentrated on the right of the
Learning Apache Spark with Python

figure. The distribution is said to be left-skewed, left-tailed, or skewed to the left, despite the fact that
the curve itself appears to be skewed or leaning to the right; left instead refers to the left tail being
drawn out and, often, the mean being skewed to the left of a typical center of the data. A left-skewed
distribution usually appears as a right-leaning curve.

2. positive skew: The right tail is longer; the mass of the distribution is concentrated on the left of the
figure. The distribution is said to be right-skewed, right-tailed, or skewed to the right, despite the fact
that the curve itself appears to be skewed or leaning to the left; right instead refers to the right tail
being drawn out and, often, the mean being skewed to the right of a typical center of the data. A
right-skewed distribution usually appears as a left-leaning curve.

This subsection comes from Wikipedia Kurtosis.

In probability theory and statistics, kurtosis (kyrtos or kurtos, meaning “curved, arching”) is a measure of the
“tailedness” of the probability distribution of a real-valued random variable. In a similar way to the concept
of skewness, kurtosis is a descriptor of the shape of a probability distribution and, just as for skewness, there
are different ways of quantifying it for a theoretical distribution and corresponding ways of estimating it
from a sample from a population.

```
from pyspark.sql.functions import col, skewness, kurtosis
df.select(skewness(var), kurtosis(var)).show()
```

```
+---------------------+---------------------+
|skewness(Age (years))|kurtosis(Age (years))|
|---------------------+---------------------+
| 1.0231743160548064| 0.6114371688367672|
```

Warning: Sometimes the statistics can be misleading!

F. J. Anscombe once said that make both calculations and graphs. Both sorts of output should be stud-
ied; each will contribute to understanding. These 13 datasets in Figure Same Stats, Different Graphs (the
Datasaurus, plus 12 others) each have the same summary statistics (x/y mean, x/y standard deviation, and
Pearson’s correlation) to two decimal places, while being drastically different in appearance. This work
describes the technique we developed to create this dataset, and others like it. More details and interesting results can be found in Same Stats Different Graphs.

![Histogram](image)

**Fig. 1: Same Stats, Different Graphs**

**Histogram**

*Warning: Histograms are often confused with Bar graphs!*

The fundamental difference between histogram and bar graph will help you to identify the two easily is that there are gaps between bars in a bar graph but in the histogram, the bars are adjacent to each other. The interested reader is referred to Difference Between Histogram and Bar Graph.

```python
var = 'Age (years)'
x = data1[var]
bins = np.arange(0, 100, 5.0)

plt.figure(figsize=(10,8))
# the histogram of the data
plt.hist(x, bins, alpha=0.8, histtype='bar', color='gold',
```
Learning Apache Spark with Python

(continued from previous page)

ec='black',weights=np.zeros_like(x) + 100. / x.size)
plt.xlabel(var)
plt.ylabel('percentage')
plt.xticks(bins)
plt.show()
fig.savefig(var+".pdf", bbox_inches='tight')

var = 'Age (years)'
x = data1[var]
bins = np.arange(0, 100, 5.0)

########################################################################
hist, bin_edges = np.histogram(x,bins,
weights=np.zeros_like(x) + 100. / x.size)
# make the histogram
(continues on next page)

68

Chapter 7. Data Exploration


```python
fig = plt.figure(figsize=(20, 8))
ax = fig.add_subplot(1, 2, 1)

# Plot the histogram heights against integers on the x axis
ax.bar(range(len(hist)), hist, width=1, alpha=0.8, ec='black', color='gold')

# Set the ticks to the middle of the bars
ax.set_xticks([0.5+i for i, j in enumerate(hist)])
# Set the xticklabels to a string that tells us what the bin edges were
labels = ['{:.0f}'.format(int(bins[i+1])) for i, j in enumerate(hist)]
labels.insert(0, '0')
ax.set_xticklabels(labels)
plt.xlabel(var)
plt.ylabel('percentage')

hist, bin_edges = np.histogram(x, bins) # make the histogram

ax = fig.add_subplot(1, 2, 2)
# Plot the histogram heights against integers on the x axis
ax.bar(range(len(hist)), hist, width=1, alpha=0.8, ec='black', color='gold')

# Set the ticks to the middle of the bars
ax.set_xticks([0.5+i for i, j in enumerate(hist)])
# Set the xticklabels to a string that tells us what the bin edges were
labels = ['{:.0f}'.format(int(bins[i+1])) for i, j in enumerate(hist)]
labels.insert(0, '0')
ax.set_xticklabels(labels)
plt.xlabel(var)
plt.ylabel('count')
plt.suptitle('Histogram of {}: Left with percentage output; Right with count output'.format(var), size=16)
plt.show()
fig.savefig(var + '.pdf', bbox_inches='tight')
```

Sometimes, some people will ask you to plot the unequal width (invalid argument for histogram) of the bars. You can still achieve it by the following trick.

```python
var = 'Credit Amount'
plot_data = df.select(var).toPandas()
x = plot_data[var]

bins = [0, 200, 400, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 10000, 25000]

hist, bin_edges = np.histogram(x, bins, weights=np.zeros_like(x) + 100. / x.size) # make the histogram

fig = plt.figure(figsize=(10, 8))
```

7.1. Univariate Analysis
Box plot and violin plot

Note that although violin plots are closely related to Tukey’s (1977) box plots, the violin plot can show more information than box plot. When we perform an exploratory analysis, nothing about the samples could be known. So the distribution of the samples can not be assumed to a normal distribution and usually when you get a big data, the normal distribution will show some out liars in box plot.

However, the violin plots are potentially misleading for smaller sample sizes, where the density plots can appear to show interesting features (and group-differences therein) even when produced for standard normal data. Some poster suggested the sample size should larger that 250. The sample sizes (e.g. n>250 or ideally even larger), where the kernel density plots provide a reasonably accurate representation of the distributions, potentially showing nuances such as bimodality or other forms of non-normality that would be invisible or less clear in box plots. More details can be found in A simple comparison of box plots and violin plots.
7.1. Univariate Analysis
Learning Apache Spark with Python

```python
x = df.select(var).toPandas()
fig = plt.figure(figsize=(20, 8))
ax = fig.add_subplot(1, 2, 1)
ax = sns.boxplot(data=x)
ax = fig.add_subplot(1, 2, 2)
ax = sns.violinplot(data=x)
```

7.1.2 Categorical Variables

Compared with the numerical variables, the categorical variables are much more easier to do the exploration.

**Frequency table**

```python
from pyspark.sql import functions as F
from pyspark.sql.functions import rank, sum, col
from pyspark.sql import Window

window = Window.rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
# withColumn('Percent %', F.format_string("%5.0f%%\n", col('Credit_num')*100/col('total'))).
tab = df.select(['age_class', 'Credit Amount']).
  groupBy('age_class').
  agg(F.count('Credit Amount').alias('Credit_num'),
      F.mean('Credit Amount').alias('Credit_avg'),
      F.min('Credit Amount').alias('Credit_min'),
      F.max('Credit Amount').alias('Credit_max')).
  withColumn('total', sum(col('Credit_num')).over(window)).
  withColumn('Percent', col('Credit_num')*100/col('total')).
  drop(col('total'))
```
<table>
<thead>
<tr>
<th>age_class</th>
<th>Credit_num</th>
<th>Credit_avg</th>
<th>Credit_min</th>
<th>Credit_max</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-54</td>
<td>120</td>
<td>3183.066666666666666</td>
<td>338</td>
<td>12612</td>
<td>12.0</td>
</tr>
<tr>
<td>&lt;25</td>
<td>150</td>
<td>3493.660714285714285</td>
<td>385</td>
<td>15945</td>
<td>5.6</td>
</tr>
<tr>
<td>55-64</td>
<td>56</td>
<td>3403.771653543307083</td>
<td>250</td>
<td>18424</td>
<td>39.7</td>
</tr>
<tr>
<td>35-44</td>
<td>254</td>
<td>3210.173913043478261</td>
<td>571</td>
<td>14896</td>
<td>2.3</td>
</tr>
<tr>
<td>25-34</td>
<td>397</td>
<td>3183.066666666666666</td>
<td>338</td>
<td>12612</td>
<td>12.0</td>
</tr>
<tr>
<td>65+</td>
<td>23</td>
<td>3210.173913043478261</td>
<td>571</td>
<td>14896</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Pie plot

```python
# Data to plot
labels = plot_data.age_class
sizes = plot_data.Percent
colors = ['gold', 'yellowgreen', 'lightcoral', 'blue', 'lightskyblue', 'green',
          'red']
explode = (0, 0.1, 0, 0, 0, 0)  # explode 1st slice

# Plot
plt.figure(figsize=(10,8))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='\%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.show()
```

Bar plot

```python
labels = plot_data.age_class
missing = plot_data.Percent
ind = [x for x, _ in enumerate(labels)]

plt.figure(figsize=(10,8))
plt.bar(ind, missing, width=0.8, label='missing', color='gold')
plt.xticks(ind, labels)
plt.ylabel("percentage")
plt.show()
```

labels = ['missing', '<25', '25-34', '35-44', '45-54', '55-64', '65+']
missing = np.array([0.000095, 0.024830, 0.028665, 0.029477, 0.031918, 0.037073, 0.026699])
man = np.array([0.000147, 0.036311, 0.038684, 0.044761, 0.051269, 0.059542, 0.054259])
woman = np.array([0.004035, 0.032935, 0.035351, 0.041778, 0.048437, 0.056236, 0.048091])

(continues on next page)
7.1. Univariate Analysis
ind = [x for x, _ in enumerate(labels)]

plt.figure(figsize=(10,8))
plt.bar(ind, women, width=0.8, label='women', color='gold', bottom=man+missing)
plt.bar(ind, man, width=0.8, label='man', color='silver', bottom=missing)
plt.bar(ind, missing, width=0.8, label='missing', color='#CD853F')

plt.xticks(ind, labels)
plt.ylabel("percentage")
plt.legend(loc="upper left")
plt.title("demo")
plt.show()
7.2 Multivariate Analysis

In this section, I will only demonstrate the bivariate analysis. Since the multivariate analysis is the generation of the bivariate.

7.2.1 Numerical V.S. Numerical

Correlation matrix

```python
from pyspark.mllib.stat import Statistics
import pandas as pd

corr_data = df.select(num_cols)

col_names = corr_data.columns
features = corr_data.rdd.map(lambda row: row[0:])
corr_mat = Statistics.corr(features, method="pearson")
corr_df = pd.DataFrame(corr_mat)
corr_df.index, corr_df.columns = col_names, col_names

print(corr_df.to_string())
```

<table>
<thead>
<tr>
<th></th>
<th>Account Balance</th>
<th>No of dependents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account Balance</td>
<td>1.0</td>
<td>-0.01414542650320914</td>
</tr>
<tr>
<td>No of dependents</td>
<td>-0.01414542650320914</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Scatter Plot

```python
import seaborn as sns
sns.set(style="ticks")

df = sns.load_dataset("iris")
sns.pairplot(df, hue="species")
plt.show()
```

7.2.2 Categorical V.S. Categorical

Pearson’s Chi-squared test

**Warning:** pyspark.ml.stat is only available in Spark 2.4.0.
```python
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import ChiSquareTest

data = [(0.0, Vectors.dense(0.5, 10.0)),
        (0.0, Vectors.dense(1.5, 20.0)),
        (1.0, Vectors.dense(1.5, 30.0)),
        (0.0, Vectors.dense(3.5, 30.0)),
        (0.0, Vectors.dense(3.5, 40.0)),
        (1.0, Vectors.dense(3.5, 40.0))]
df = spark.createDataFrame(data, ['label', 'features'])

r = ChiSquareTest.test(df, 'features', 'label').head()
print("pValues: " + str(r.pValues))
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
print("statistics: " + str(r.statistics))
pValues: [0.687289278791, 0.682270330336]
degreesOfFreedom: [2, 3]
statistics: [0.75, 1.5]

Cross table

df.stat.crosstab("age_class", "Occupation").show()

+--------------------+---+---+---+---+
|age_class_Occupation| 1 | 2 | 3 | 4 |
+--------------------+---+---+---+---+
|<25                 | 4 | 34|108| 4 |
|55-64               | 1 | 15| 31| 9 |
|25-34               | 7 | 61|269|60 |
|35-44               | 4 | 58|143|49 |
|65+                 | 5 | 3 | 6 | 9 |
|45-54               | 1 | 29| 73|17 |
+--------------------+---+---+---+---+

Stacked plot

labels = ['missing', '<25', '25-34', '35-44', '45-54', '55-64', '65+']
moving = np.array([0.000095, 0.024830, 0.028665, 0.029477, 0.031918, 0.037073, 0.026699])
man = np.array([0.000147, 0.036311, 0.038684, 0.044761, 0.051269, 0.059542, 0.054259])
women = np.array([0.004035, 0.032935, 0.035351, 0.041778, 0.048437, 0.056236, 0.048091])
ind = [x for x, _ in enumerate(labels)]
plt.figure(figsize=(10,8))
plt.bar(ind, women, width=0.8, label='women', color='gold', bottom=man+moving)
```

(continues on next page)
7.2.3 Numerical V.S. Categorical

Line Chart with Error Bars

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)
import seaborn as sns
from scipy import stats
%matplotlib inline

plt.rcParams['figure.figsize'] = (16, 9)
plt.style.use('ggplot')
sns.set()

ax = sns.pointplot(x="day", y="tip", data=tips, capsize=.2)
plt.show()

Combination Chart

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
%matplotlib inline

plt.rcParams['figure.figsize'] = (16, 9)
plt.style.use('ggplot')
sns.set()

# create list of months

7.2. Multivariate Analysis
Month = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June',
        'July', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
# create list for made up average temperatures
Avg_Temp = [35, 45, 55, 65, 75, 85, 95, 100, 85, 65, 45, 35]
# create list for made up average percipitation %
Avg_Percipitation_Perc = [.90, .75, .55, .10, .35, .05, .05, .08, .20, .45, .
                       .65, .80]
# assign lists to a value
data = {'Month': Month, 'Avg_Temp': Avg_Temp, 'Avg_Percipitation_Perc': Avg_ 
Percipitation_Perc}
# convert dictionary to a dataframe
df = pd.DataFrame(data)

fig, ax1 = plt.subplots(figsize=(10,6))
ax1.set_title('Average Percipitation Percentage by Month', fontsize=16)
ax1.tick_params(axis='y')

ax2 = sns.barplot(x='Month', y='Avg_Temp', data = df, color = 'gold')
ax2 = ax1.twinx()
ax2 = sns.lineplot(x='Month', y='Avg_Percipitation_Perc', data = df,
                   sort=False, color=color)
ax1.set_xlabel('Month', fontsize=16)
ax1.set_ylabel('Avg Temp', fontsize=16)
ax2.tick_params(axis='y', color=color)
ax2.set_ylabel('Avg Percipitation %', fontsize=16)
plt.show()
Average Precipitation Percentage by Month

7.2. Multivariate Analysis
Feature building is a super important step for modeling which will determine the success or failure of your model. Otherwise, you will get: garbage in; garbage out! The techniques have been covered in the following chapters, the followings are the brief summary. I recently found that the Spark official website did a really good job for tutorial documentation. The chapter is based on Extracting transforming and selecting features.

8.1 Feature Extraction

8.1.1 TF-IDF

Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus. More details can be found at: https://spark.apache.org/docs/latest/ml-features#feature-extractors

Stackoverflow TF: Both HashingTF and CountVectorizer can be used to generate the term frequency vectors. A few important differences:

a. partially reversible (CountVectorizer) vs irreversible (HashingTF) - since hashing is not reversible you cannot restore original input from a hash vector. From the other hand count vector with model (index) can be used to restore unordered input. As a consequence models created using hashed input can be much harder to interpret and monitor.

b. memory and computational overhead - HashingTF requires only a single data scan and no additional memory beyond original input and vector. CountVectorizer requires additional scan over the data to build a model and additional memory to store vocabulary (index). In case of unigram language model it is usually not a problem but in case of higher n-grams it can be prohibitively expensive or not feasible.

c. hashing depends on a size of the vector, hashing function and a document. Counting depends on a size of the vector, training corpus and a document.
d. a source of the information loss - in case of HashingTF it is dimensionality reduction with possible collisions. CountVectorizer discards infrequent tokens. How it affects downstream models depends on a particular use case and data.

HashingTF and CountVectorizer are the two popular algorithms which used to generate term frequency vectors. They basically convert documents into a numerical representation which can be fed directly or with further processing into other algorithms like LDA, MinHash for Jaccard Distance, Cosine Distance.

- $t$: term
- $d$: document
- $D$: corpus
- $|D|$: the number of the elements in corpus
- $TF(t,d)$: Term Frequency: the number of times that term $t$ appears in document $d$
- $DF(t,D)$: Document Frequency: the number of documents that contains term $t$
- $IDF(t,D)$: Inverse Document Frequency is a numerical measure of how much information a term provides

\[
IDF(t,D) = \log \frac{|D| + 1}{DF(t,D) + 1}
\]

- $TFIDF(t,d,D)$ the product of TF and IDF

\[
TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D)
\]

Let’s look at the example:

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([(
    0, "Python python Spark Spark"),
    (1, "Python SQL")], [
    "document", "sentence"])

sentenceData.show(truncate=False)
```

<table>
<thead>
<tr>
<th>document</th>
<th>sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Python python Spark Spark</td>
</tr>
<tr>
<td>1</td>
<td>Python SQL</td>
</tr>
</tbody>
</table>

Then:

- $TF("python", document1) = 1, TF("spark", document1) = 2$
- $DF(Spark, D) = 2, DF(sql, D) = 1$
- $IDF:
\[ IDF(python, D) = \log \frac{|D| + 1}{DF(t, D) + 1} = \log \frac{2 + 1}{2 + 1} = 0 \]

\[ IDF(spark, D) = \log \frac{|D| + 1}{DF(t, D) + 1} = \log \frac{2 + 1}{1 + 1} = 0.4054651081081644 \]

\[ IDF(sql, D) = \log \frac{|D| + 1}{DF(t, D) + 1} = \log \frac{2 + 1}{1 + 1} = 0.4054651081081644 \]

- **TFIDF**

\[ TFIDF(python, document1, D) = 3 \times 0 = 0 \]

\[ TFIDF(spark, document1, D) = 2 \times 0.4054651081081644 = 0.8109302162163288 \]

\[ TFIDF(sql, document1, D) = 1 \times 0.4054651081081644 = 0.4054651081081644 \]

**CountVectorizer**

**Stackoverflow TF:** CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame(
    ([0, "Python python Spark Spark"],
     [1, "Python SQL"]),
    ["document", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
vectorizer = CountVectorizer(inputCol="words", outputCol="rawFeatures")

idf = IDF(inputCol="rawFeatures", outputCol="features")

pipeline = Pipeline(stages=[tokenizer, vectorizer, idf])

model = pipeline.fit(sentenceData)
```

```python
import numpy as np

total_counts = model.transform(sentenceData)
    .select('rawFeatures').rdd
    .map(lambda row: row['rawFeatures'].toArray())
    .reduce(lambda x,y: [x[i]+y[i] for i in range(len(y))])

vocabList = model.stages[1].vocabulary

D = {'vocabList':vocabList, 'counts':total_counts}
```

(continues on next page)
Learning Apache Spark with Python

spark.createDataFrame(np.array(list(d.values())).T.tolist(),list(d.keys())).
→show()

counts = model.transform(sentenceData).select('rawFeatures').collect()
counts

[Row(rawFeatures=\nSparseVector(8, {0: 1.0, 1: 1.0, 2: 1.0})),
Row(rawFeatures=\nSparseVector(8, {0: 1.0, 1: 1.0, 4: 1.0})),
Row(rawFeatures=\nSparseVector(8, {0: 1.0, 3: 1.0, 5: 1.0, 6: 1.0, 7: 1.0}))]

+---------+------+
| vocabList| counts|
+---------+------+
| python| 3.0|
| spark| 2.0|
| sql| 1.0|
+---------+------+

model.transform(sentenceData).show(truncate=False)

+--------+-------------------------+------------------------------+-----------
|document|sentence |words
|rawFeatures |features |
+--------+-------------------------+------------------------------+-----------
|0 |Python python Spark Spark|\n[python, python, spark, spark]|(3,[0,1],
| |[2.0,2.0])|(3,[0,1],[0.0,0.8109302162163288])|
|1 |Python SQL|\n[python, sql] |(3,[0,2],
| |[1.0,1.0])|(3,[0,2],[0.0,0.4054651081081644])|
+--------+-------------------------+------------------------------+-----------

from pyspark.sql.types import ArrayType, StringType
def termsIdx2Term(vocabulary):
    def termsIdx2Term(termIndices):
        return [vocabulary[int(index)] for index in termIndices]
    return udf(termsIdx2Term, ArrayType(StringType()))

vectorizerModel = model.stages[1]
vocabList = vectorizerModel.vocabulary
vocabList

['python', 'spark', 'sql']

rawFeatures = model.transform(sentenceData).select('rawFeatures')
rawFeatures.show()
Learning Apache Spark with Python

(continued from previous page)

<table>
<thead>
<tr>
<th>rawFeatures</th>
<th>indices</th>
<th>values</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3, [0, 1], [2.0, 2.0])</td>
<td>[0, 1]</td>
<td>[2.0, 2.0, 0.0]</td>
<td>[python, spark]</td>
</tr>
<tr>
<td>(3, [0, 2], [1.0, 1.0])</td>
<td>[0, 2]</td>
<td>[1.0, 0.0, 1.0]</td>
<td>[python, sql]</td>
</tr>
</tbody>
</table>

HashingTF

Stackoverflow TF: HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. In text processing, a “set of terms” might be a bag of words. HashingTF utilizes the hashing trick. A raw feature is mapped into an index (term) by applying a hash function. The hash function used here is MurmurHash 3. Then term frequencies are calculated based on the mapped indices. This approach avoids the need to compute a global term-to-index map, which can be expensive for a large corpus, but it suffers from potential hash collisions, where different raw features may become the same term after hashing.

from pyspark.ml import Pipeline
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame(
    [(0, "Python python Spark Spark"),
     (1, "Python SQL")],
    ["document", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

(continues on next page)
vectorizer = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=5)

df = IDF(inputCol="rawFeatures", outputCol="features")

pipeline = Pipeline(stages=[tokenizer, vectorizer, idf])

model = pipeline.fit(sentenceData)
model.transform(sentenceData).show(truncate=False)

<table>
<thead>
<tr>
<th>document</th>
<th>sentence</th>
<th>words</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Python python Spark Spark</td>
<td>[python, python, spark, spark]</td>
</tr>
<tr>
<td>1</td>
<td>Python SQL</td>
<td>[python, sql]</td>
</tr>
</tbody>
</table>

8.1.2 Word2Vec

**Word Embeddings**

Word2Vec is one of the popular methods to implement the Word Embeddings. Word embeddings (The best tutorial I have read. The following word and images content are from Chris Bail, PhD Duke University. So the copyright belongs to Chris Bail, PhD Duke University.) gained fame in the world of automated text analysis when it was demonstrated that they could be used to identify analogies. Figure 1 illustrates the output of a word embedding model where individual words are plotted in three dimensional space generated by the model. By examining the adjacency of words in this space, word embedding models can complete analogies such as “Man is to woman as king is to queen.” If you’d like to explore what the output of a large word embedding model looks like in more detail, check out this fantastic visualization of most words in the English language that was produced using a word embedding model called GloVE.

**The Context Window**

Word embeddings are created by identifying the words that occur within something called a “Context Window.” The Figure below illustrates context windows of varied length for a single sentence. The context window is defined by a string of words before and after a focal or “center” word that will be used to train a word embedding model. Each center word and context words can be represented as a vector of numbers that describe the presence or absence of unique words within a dataset, which is perhaps why word embedding models are often described as “word vector” models, or “word2vec” models.
Fig. 1: output of a word embedding model

- **Center Word**
- **Context Word**

\[\begin{align*}
c=0 & \quad \text{The cute } \textcolor{blue}{\text{cat}} \text{ jumps over the lazy dog.} \\
c=1 & \quad \text{The cute } \textcolor{red}{\text{cat}} \text{ jumps } \textcolor{blue}{\text{over}} \text{ over the lazy dog.} \\
c=2 & \quad \text{The cute } \textcolor{red}{\text{cat}} \text{ jumps } \textcolor{blue}{\text{over}} \text{ the lazy dog.}
\end{align*}\]
Two Types of Embedding Models

Word embeddings are usually performed in one of two ways: “Continuous Bag of Words” (CBOW) or a “Skip-Gram Model.” The figure below illustrates the differences between the two models. The CBOW model reads in the context window words and tries to predict the most likely center word. The Skip-Gram Model predicts the context words given the center word. The examples above were created using the Skip-Gram model, which is perhaps most useful for people who want to identify patterns within texts to represent them in multidimensional space, whereas the CBOW model is more useful in practical applications such as predictive web search.

Word Embedding Models in PySpark

```python
from pyspark.ml.feature import Word2Vec
from pyspark.ml import Pipeline

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="words", outputCol="feature")
	pipeline = Pipeline(stages=[tokenizer, word2Vec])

model = pipeline.fit(sentenceData)
result = model.transform(sentenceData)
```
result.show()

+-----+--------------------+--------------------+--------------------+
| label| sentence            | words              | feature             |
+-----+--------------------+--------------------+--------------------+
| 0.0 | I love Spark       | [i, love, spark]   | [0.05594437588782... |
| 0.0 | I love python      | [i, love, python]  | [-0.0350368790871... |
| 1.0 | I think ML is awe... | [i, think, ml, is...] | [0.01242086507845... |
+-----+--------------------+--------------------+--------------------+

w2v = model.stages[1]
w2v.getVectors().show()

+-------+-----------------------------------------------------------------+
<table>
<thead>
<tr>
<th>word</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>[0.13657838106155396, 0.060924094170331955, -0.03379475697875023]</td>
</tr>
<tr>
<td>awesome</td>
<td>[0.037024181336164474, -0.02385900391936302, 0.0760037824511528]</td>
</tr>
<tr>
<td>i</td>
<td>[-0.0014482572441920638, 0.049365971237421036, 0.12016955763101578]</td>
</tr>
<tr>
<td>ml</td>
<td>[-0.14006119966506958, 0.016264444421708584, 0.042281970381736755]</td>
</tr>
<tr>
<td>spark</td>
<td>[0.1589149385690689, -0.10970081388950348, -0.10547549277544022]</td>
</tr>
<tr>
<td>think</td>
<td>[0.030011219903826714, -0.08994936943054199, 0.16471518576145172]</td>
</tr>
<tr>
<td>love</td>
<td>[0.01036646633859396, -0.017782460898160934, 0.08870164304971695]</td>
</tr>
<tr>
<td>python</td>
<td>[-0.11402882635593414, 0.045119188725948334, -0.02987742377467155]</td>
</tr>
</tbody>
</table>
+-------+-----------------------------------------------------------------+

from pyspark.sql.functions import format_number as fmt
w2v.findSynonyms("could", 2).select("word", fmt("similarity", 5).alias("similarity")).show()

+--------+----------+
<table>
<thead>
<tr>
<th>word</th>
<th>similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>0.90232</td>
</tr>
<tr>
<td>i</td>
<td>0.75424</td>
</tr>
</tbody>
</table>
+--------+----------+

8.1.3 FeatureHasher

from pyspark.ml.feature import FeatureHasher

dataset = spark.createDataFrame([(
    2.2, True, "1", "foo"),
    (3.3, False, "2", "bar"),
    (4.4, False, "3", "baz"),
    (5.5, False, "4", "foo")
    ], ["real", "bool", "stringNum", "string"])

hasher = FeatureHasher(inputCols="[real", "bool", "stringNum", "string"],
outputCol="features")

(continues on next page)
featurized = hasher.transform(dataset)
featurized.show(truncate=False)

8.1.4 RFormula

from pyspark.ml.feature import RFormula

dataset = spark.createDataFrame(
    [(7, "US", 18, 1.0),
     (8, "CA", 12, 0.0),
     (9, "CA", 15, 0.0)],
    ["id", "country", "hour", "clicked"]
)

formula = RFormula(
    formula="clicked ~ country + hour",
    featuresCol="features",
    labelCol="label")

output = formula.fit(dataset).transform(dataset)
output.select("features", "label").show()
8.2 Feature Transform

8.2.1 Tokenizer

```python
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType

sentenceDataFrame = spark.createDataFrame([ 
    (0, "Hi I heard about Spark"),
    (1, "I wish Java could use case classes"),
    (2, "Logistic,regression,models,are,neat")
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words",
    → pattern="\W")
# alternatively, pattern="\w+", gaps(False)
countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words")
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
```

<table>
<thead>
<tr>
<th>sentence</th>
<th>words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi I heard about Spark</td>
<td>[hi, i, heard, about, spark]</td>
</tr>
<tr>
<td>I wish Java could use case classes</td>
<td>[i, wish, java, could, use, case, classes]</td>
</tr>
<tr>
<td>Logistic,regression,models,are,neat</td>
<td>[logistic,regression,models,are,neat]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sentence</th>
<th>words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi I heard about Spark</td>
<td>[hi, i, heard, about, spark]</td>
</tr>
<tr>
<td>I wish Java could use case classes</td>
<td>[i, wish, java, could, use, case, classes]</td>
</tr>
<tr>
<td>Logistic,regression,models,are,neat</td>
<td>[logistic,regression,models,are,neat]</td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th>Hi I heard about Spark</th>
<th>[hi, i, heard, about, spark]</th>
</tr>
</thead>
<tbody>
<tr>
<td>I wish Java could use case classes</td>
<td>[i, wish, java, could, use, case, classes]</td>
</tr>
<tr>
<td>Logistic, regression, models, are, neat</td>
<td>[logistic, regression, models, are, neat]</td>
</tr>
</tbody>
</table>

8.2.2 StopWordsRemover

```python
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([  
    (0, ["I", "saw", "the", "red", "balloon"]),  
    (1, ["Mary", "had", "a", "little", "lamb"])  
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="removed")
remover.transform(sentenceData).show(truncate=False)
```

|---+----------------------------+--------------------|
|id |raw |removeded |
|---+----------------------------+--------------------|
|0 |[I, saw, the, red, balloon] |[saw, red, balloon] |
|1 |[Mary, had, a, little, lamb] |[Mary, little, lamb]|

8.2.3 NGram

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
from pyspark.ml.feature import Ngram

sentenceData = spark.createDataFrame([  
    (0.0, "I love Spark"),  
    (0.0, "I love python"),  
    (1.0, "I think ML is awesome"),  
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
ngram = Ngram(n=2, inputCol="words", outputCol="ngrams")

idf = IDF(inputCol="rawFeatures", outputCol="features")
```

(continues on next page)
pipeline = Pipeline(stages=[tokenizer, ngram])

model = pipeline.fit(sentenceData)

model.transform(sentenceData).show(truncate=False)

8.2.4 Binarizer

from pyspark.ml.feature import Binarizer

continuousDataFrame = spark.createDataFrame([n(0, 0.1),
                                              (1, 0.8),
                                              (2, 0.2),
                                              (3, 0.5)], ["id", "feature"])

binarizer = Binarizer(threshold=0.5, inputCol="feature", outputCol="binarized_feature")

binarizedDataFrame = binarizer.transform(continuousDataFrame)

print("Binarizer output with Threshold = \$f\" % binarizer.getThreshold())

binarizedDataFrame.show()
8.2.5 Bucketizer

[Bucketizer](https://spark.apache.org/docs/latest/ml-features.html#bucketizer) transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users.

```python
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer
data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.0)]
df = spark.createDataFrame(data, ['id', 'age'])
print(df.show())
splits = [-float('inf'), 3, 10, float('inf')]
result_bucketizer = Bucketizer(splits=splits, inputCol='age', outputCol='result').transform(df)
result_bucketizer.show()
```

```
+---+----+
| id| age|
+---+----+
| 0|18.0|
| 1|19.0|
| 2| 8.0|
| 3| 5.0|
| 4| 2.0|
+---+----+
None
```

8.2.6 QuantileDiscretizer

QuantileDiscretizer takes a column with continuous features and outputs a column with binned categorical features. The number of bins is set by the numBuckets parameter. It is possible that the number of buckets used will be smaller than this value, for example, if there are too few distinct values of the input to create enough distinct quantiles.

```python
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer
data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.0)]
df = spark.createDataFrame(data, ['id', 'age'])
print(df.show())
```

(continues on next page)
```python
qds = QuantileDiscretizer(numBuckets=5, inputCol="age", outputCol="buckets",
                        relativeError=0.01, handleInvalid="error")

bucketizer = qds.fit(df)
bucketizer.transform(df).show()
bucketizer.setHandleInvalid("skip").transform(df).show()
```

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18.0</td>
</tr>
<tr>
<td>1</td>
<td>19.0</td>
</tr>
<tr>
<td>2</td>
<td>8.0</td>
</tr>
<tr>
<td>3</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
</tbody>
</table>

None

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
<th>buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18.0</td>
<td>3.0</td>
</tr>
<tr>
<td>1</td>
<td>19.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>8.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>5.0</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

If the data has NULL values, then you will get the following results:

```python
from pyspark.ml.feature import QuantileDiscretizer, Bucketizer
data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, None)]
df = spark.createDataFrame(data, ["id", "age"])
print(df.show())
splits = [-float("inf"), 3, 10, float("inf")]
result_bucketizer = Bucketizer(splits=splits,
                                 inputCol="age", outputCol="result").
                        transform(df)
result_bucketizer.show()
```

8.2. Feature Transform
qds = QuantileDiscretizer(numBuckets=5, inputCol="age", outputCol="buckets", relativeError=0.01, handleInvalid="error")

bucketizer = qds.fit(df)
bucketizer.transform(df).show()
bucketizer.setHandleInvalid("skip").transform(df).show()

+---+----+------+
| id| age|result|
+---+----+------+
| 0|18.0| 2.0 |
| 1|19.0| 2.0 |
| 2| 8.0| 1.0 |
| 3| 5.0| 1.0 |
| 4|null| null |
+---+----+------+

None

+---+----+------+
| id| age|buckets |
+---+----+------+
| 0|18.0| 3.0 |
| 1|19.0| 4.0 |
| 2| 8.0| 2.0 |
| 3| 5.0| 1.0 |
| 4|null| null |
+---+----+------+

8.2.7 StringIndexer
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame(
    [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
    ["id", "category"])

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()

+---+--------+-------------+
| id|category|categoryIndex|
+---+--------+-------------+
| 0| a| 0.0|
| 1| b| 2.0|
| 2| c| 1.0|
| 3| a| 0.0|
| 4| a| 0.0|
| 5| c| 1.0|
+---+--------+-------------+

8.2.8 labelConverter

from pyspark.ml.feature import IndexToString, StringIndexer

df = spark.createDataFrame(
    [(0, "Yes"), (1, "Yes"), (2, "Yes"), (3, "No"), (4, "No"), (5, "No")],
    ["id", "label"])

indexer = StringIndexer(inputCol="label", outputCol="labelIndex")
model = indexer.fit(df)
indexed = model.transform(df)

print("Transformed string column '%s' to indexed column '%s'" % (indexer.getInputCol(), indexer.getOutputCol()))
indexed.show()

print("StringIndexer will store labels in output column metadata\n"

converter = IndexToString(inputCol="labelIndex", outputCol="originalLabel")
converted = converter.transform(indexed)

print("Transformed indexed column '%s' back to original string column '%s' using " % "labels in metadata" % (converter.getInputCol(), converter.getOutputCol()))
converted.select("id", "labelIndex", "originalLabel").show()
StringIndexer will store labels in output column metadata

Transformed indexed column 'labelIndex' back to original string column 'originalLabel' using labels in metadata

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString, StringIndexer

df = spark.createDataFrame(
    [(0, "Yes"), (1, "Yes"), (2, "Yes"), (3, "No"), (4, "No"), (5, "No")],
    ["id", "label"])

indexer = StringIndexer(inputCol="label", outputCol="labelIndex")
converter = IndexToString(inputCol="labelIndex", outputCol="originalLabel")

pipeline = Pipeline(stages=[indexer, converter])

model = pipeline.fit(df)
result = model.transform(df)
result.show()
```
8.2.9 VectorIndexer

from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.feature import RFormula

df = spark.createDataFrame([ 
    (0, 2.2, True, "1", "foo", 'CA'),
    (1, 3.3, False, "2", "bar", 'US'),
    (0, 4.4, False, "3", "baz", 'CHN'),
    (1, 5.5, False, "4", "foo", 'AUS')
], ['label', 'real', 'bool', 'stringNum', 'string', 'country'])

formula = RFormula(
    formula="label ~ real + bool + stringNum + string + country",
    featuresCol="features",
    labelCol="label")

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values
# are treated as continuous.
featureIndexer = VectorIndexer(inputCol="features",
                                outputCol="indexedFeatures",
                                maxCategories=2)

pipeline = Pipeline(stages=[formula, featureIndexer])

model = pipeline.fit(df)
result = model.transform(df)
result.show()
| 1 | 5.5 | false | 4 | foo | AUS | (10, [0, 2, 5], [5.5, ... | (10, [0, 2, 5], |
| 5.5, ... |
| +-----------------------+-------+|-----------------------+-------|

### 8.2.10 VectorAssembler

```python
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
dataset = spark.createDataFrame(
    [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)],
    ["id", "hour", "mobile", "userFeatures", "clicked"]
)
assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")
output = assembler.transform(dataset)
print("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'")
output.select("features", "clicked").show(truncate=False)
```

Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'
+-----------------------+-------+
|features               | clicked|
+-----------------------+-------+
|[18.0,1.0,0.0,10.0,0.5]|1.0    |
+-----------------------+-------+

### 8.2.11 OneHotEncoder

This is the note I wrote for one of my readers for explaining the OneHotEncoder. I would like to share it at here:

**Import and creating SparkSession**

```python
from pyspark.sql import SparkSession
spark = SparkSession
.builder
   .appName("Python Spark create RDD example")
   .config("spark.some.config.option", "some-value")
   .getOrCreate()
```

104 Chapter 8. Data Manipulation: Features
df = spark.createDataFrame([  
    (0, "a"),  
    (1, "b"),  
    (2, "c"),  
    (3, "a"),  
    (4, "a"),  
    (5, "c")  
], ["id", "category"])

df.show()

+---+--------+
| id|category |
+---+--------+
| 0 | a      |
| 1 | b      |
| 2 | c      |
| 3 | a      |
| 4 | a      |
| 5 | c      |
+---+--------+

OneHotEncoder

from pyspark.ml.feature import OneHotEncoder, StringIndexer

stringIndexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
model = stringIndexer.fit(df)
indexed = model.transform(df)

# default setting: dropLast=True
encoder = OneHotEncoder(inputCol="categoryIndex", outputCol="categoryVec", 
                         dropLast=False)
encoded = encoder.transform(indexed)
encoded.show()

+---+--------+-------------+-------------+
| id|category|categoryIndex|categoryVec  |
+---+--------+-------------+-------------+
| 0 | a      | 0.0|[3,[0],[1.0]] |
| 1 | b      | 2.0|[3,[2],[1.0]] |
| 2 | c      | 1.0|[3,[1],[1.0]] |
| 3 | a      | 0.0|[3,[0],[1.0]] |
| 4 | a      | 0.0|[3,[0],[1.0]] |
| 5 | c      | 1.0|[3,[1],[1.0]] |
+---+--------+-------------+-------------+

8.2. Feature Transform
Note: The default setting of OneHotEncoder is: dropLast=True

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler

categoricalCols = ['category']
indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
            for c in categoricalCols]

# default setting: dropLast=True
codes = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                        outputCol="{0}_encoded".format(indexer.getOutputCol()),
                        dropLast=False)
         for indexer in indexers]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in codes],
                            outputCol="features")

pipeline = Pipeline(stages=indexers + codes + [assembler])

model = pipeline.fit(df)
data = model.transform(df)
```

(continues on next page)
Application: Get Dummy Variable

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol, dropLast=False):
    '''
    Get dummy variables and concat with continuous variables for ml modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com

    >>> df = spark.createDataFrame(
    (0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c"
    ), ["id", "category"]
    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []
    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol)
    >>> mat.show()
    >>>
    +---+-------------+
    | id| features    |
    +---+-------------+
    | 0| [1.0,0.0,0.0]|
    | 1| [0.0,0.0,1.0]|
    | 2| [0.0,1.0,0.0]|
    | 3| [1.0,0.0,0.0]|
    | 4| [1.0,0.0,0.0]|
    | 5| [0.0,1.0,0.0]|
    +---+-------------+
    '''
```

(continues on next page)
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
   VectorAssembler
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
   for c in categoricalCols ]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
   outputCol="{0}_encoded".format(indexer.getOutputCol()),
   dropLast=dropLast)
   for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
   # for supervised learning
   data = data.withColumn('label',col(labelCol))
   return data.select(indexCol,'features','label')
elif not indexCol and labelCol:
   # for supervised learning
   data = data.withColumn('label',col(labelCol))
   return data.select('features','label')
elif indexCol and not labelCol:
   # for unsupervised learning
   return data.select(indexCol,'features')
elif not indexCol and not labelCol:
   # for unsupervised learning
   return data.select('features')

Unsupervised scenario

df = spark.createDataFrame([ (0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c") ], ["id", "category"])

(continues on next page)
df.show()

indexCol = 'id'
categoricalCols = ['category']
continuousCols = []
labelCol = []

mat = get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol)

mat.show()

+---+-------------+
| id| features    |
+---+-------------+
| 0|[1.0,0.0,0.0]|
| 1|[0.0,0.0,1.0]|
| 2|[0.0,1.0,0.0]|
| 3|[1.0,0.0,0.0]|
| 4|[1.0,0.0,0.0]|
| 5|[0.0,1.0,0.0]|
+---+-------------+

Supervised scenario

df = spark.read.csv(path='bank.csv',
                   sep=',', encoding='UTF-8', comment=None,
                   header=True, inferSchema=True)

indexCol = []
catCols = ['job', 'marital', 'education', 'default',
           'housing', 'loan', 'contact', 'poutcome']
contCols = ['balance', 'duration', 'campaign', 'pdays', 'previous']
labelCol = 'y'

data = get_dummy(df, indexCol, catCols, contCols, labelCol, dropLast=False)
data.show(5)

+--------------------+-----+
| features| label|
+--------------------+-----+
| (37,[8,12,17,19,2...| no  |
| (37,[4,12,15,19,2...| no  |
| (37,[0,13,16,19,2...| no  |
| (37,[0,12,16,19,2...| no  |
| (37,[1,12,15,19,2...| no  |
+--------------------+-----+
only showing top 5 rows

8.2. Feature Transform
The Jupyter Notebook can be found on Colab: OneHotEncoder.

### 8.2.12 Scaler

```python
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler, MaxAbsScaler

scaler_type = 'Normal'
if scaler_type=='Normal':
    scaler = Normalizer(inputCol="features", outputCol="scaledFeatures", p=1.0)
elif scaler_type=='Standard':
    scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures", withStd=True, withMean=False)
elif scaler_type=='MinMaxScaler':
    scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
elif scaler_type=='MaxAbsScaler':
    scaler = MaxAbsScaler(inputCol="features", outputCol="scaledFeatures")
```

```python
from pyspark.ml import Pipeline
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([(
    0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])
df.show()

pipeline = Pipeline(stages=[scaler])
model = pipeline.fit(df)
data = model.transform(df)
data.show()
```

+---+--------------+------------------+
| id| features | scaledFeatures |
+---+--------------+------------------+
| 0 |[1.0,0.5,-1.0]| [0.4,0.2,-0.4] |
| 1 |[2.0,1.0,1.0]| [0.5,0.25,0.25] |
| 2 |[4.0,10.0,2.0]| [0.25,0.625,0.125] |
+---+--------------+------------------+
Normalizer

```python
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([  
    (0, Vectors.dense([1.0, 0.5, -1.0]),),  
    (1, Vectors.dense([2.0, 1.0, 1.0]),),  
    (2, Vectors.dense([4.0, 10.0, 2.0]),)  
], ["id", "features"])

# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normFeatures", p=1.0)
l1NormData = normalizer.transform(dataFrame)
print("Normalized using L^1 norm")
l1NormData.show()

# Normalize each Vector using $L^\infty$ norm.
lInfNormData = normalizer.transform(dataFrame, {normalizer.p: float("inf")})
print("Normalized using L^\inf norm")
lInfNormData.show()
```

StandardScaler

```python
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,
                         MaxAbsScaler
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([  
    (0, Vectors.dense([1.0, 0.5, -1.0]),),  
    (1, Vectors.dense([2.0, 1.0, 1.0]),),  
    (2, Vectors.dense([4.0, 10.0, 2.0]),)  
], ["id", "features"])

# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normFeatures", p=1.0)
l1NormData = normalizer.transform(dataFrame)
print("Normalized using L^1 norm")
l1NormData.show()

# Normalize each Vector using $L^\infty$ norm.
lInfNormData = normalizer.transform(dataFrame, {normalizer.p: float("inf")})
print("Normalized using L^\inf norm")
lInfNormData.show()
```

8.2. Feature Transform
(2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                      withStd=True, withMean=False)
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)

MinMaxScaler

from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,
                      MaxAbsScaler

from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame(
    [0, Vectors.dense([1.0, 0.5, -1.0]),),
    [1, Vectors.dense([2.0, 1.0, 1.0]),),
    [2, Vectors.dense([4.0, 10.0, 2.0]),),
], ["id", "features"])

scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)
MaxAbsScaler

```python
from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler,
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([(
    0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

scaler = MaxAbsScaler(inputCol="features", outputCol="scaledFeatures")
scaledData = scaler.fit((dataFrame)).transform(dataFrame)
scaledData.show(truncate=False)
```

8.2.13 PCA

```python
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors

data = [(Vectors.sparse(5, [(1, 1.0), (3, 7.0)]),),
    (Vectors.dense([2.0, 0.0, 3.0, 4.0, 5.0]),),
    (Vectors.dense([4.0, 0.0, 0.0, 6.0, 7.0]),)]
df = spark.createDataFrame(data, ["features"])
pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)
result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```
8.2.14 DCT

```python
from pyspark.ml.feature import DCT
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame(
    [
        (Vectors.dense([0.0, 1.0, -2.0, 3.0]),),
        (Vectors.dense([-1.0, 2.0, 4.0, -7.0]),),
        (Vectors.dense([14.0, -2.0, -5.0, 1.0]),),
    ],
    "features"
)

dct = DCT(inverse=False, inputCol="features", outputCol="featuresDCT")

dctDf = dct.transform(df)

dctDf.select("featuresDCT").show(truncate=False)
```

8.3 Feature Selection

8.3.1 LASSO

Variable selection and the removal of correlated variables. The Ridge method shrinks the coefficients of correlated variables while the LASSO method picks one variable and discards the others. The elastic net penalty is a mixture of these two; if variables are correlated in groups then $\alpha = 0.5$ tends to select the groups as in or out. If $\alpha$ is close to 1, the elastic net performs much like the LASSO method and removes any degeneracies and wild behavior caused by extreme correlations.

8.3.2 RandomForest

AutoFeatures library based on RandomForest is coming soon.............
8.4 Unbalanced data: Undersampling

Since we use PySpark to deal with the big data, Undersampling for Unbalanced Classification is a useful method to deal with the Unbalanced data. Undersampling is a popular technique for unbalanced datasets to reduce the skew in class distributions. However, it is well-known that undersampling one class modifies the priors of the training set and consequently biases the posterior probabilities of a classifier. After you applied the Undersampling, you need to recalibrate the Probability Calibrating Probability with Undersampling for Unbalanced Classification.

```python
df = spark.createDataFrame([  
    (0, "Yes"),  
    (1, "Yes"),  
    (2, "Yes"),  
    (3, "Yes"),  
    (4, "No"),  
    (5, "No")  
], ["id", "label"])
df.show()
```

8.4.1 Calculate undersampling Ratio

```python
import math

def round_up(n, decimals=0):
    multiplier = 10 ** decimals
    return math.ceil(n * multiplier) / multiplier
```

(continues on next page)
# drop missing value rows
df = df.dropna()

# under-sampling majority set
label_Y = df.filter(df.label==’Yes’)
label_N = df.filter(df.label==’No’)
sampleRatio = round_up(label_N.count() / df.count(), 2)

## 8.4.2 Undersampling

label_Y_sample = label_Y.sample(False, sampleRatio)

# union minority set and the under-sampling majority set
data = label_N.unionAll(label_Y_sample)
data.show()

### 8.4.3 Recalibrating Probability

Undersampling is a popular technique for unbalanced datasets to reduce the skew in class distributions. However, it is well-known that undersampling one class modifies the priors of the training set and consequently biases the posterior probabilities of a classifier. Calibrating Probability with Undersampling for Unbalanced Classification.

```
prediction.withColumn(’adj_probability’, sampleRatio*F.col(’probability’)/((sampleRatio-1)*F.col(’probability’)+1))
```
Chinese proverb

A journey of a thousand miles begins with a single step. – old Chinese proverb

In statistical modeling, regression analysis focuses on investigating the relationship between a dependent variable and one or more independent variables. Wikipedia Regression analysis

In data mining, Regression is a model to represent the relationship between the value of label (or target, it is numerical variable) and on one or more features (or predictors they can be numerical and categorical variables).

9.1 Linear Regression

9.1.1 Introduction

Given that a data set \( \{ x_{i1}, \ldots, x_{in}, y_i \}_{i=1}^m \) which contains \( n \) features (variables) and \( m \) samples (data points), in simple linear regression model for modeling \( m \) data points with \( j \) independent variables: \( x_{ij} \), the formula is given by:

\[
y_i = \beta_0 + \beta_j x_{ij}, \text{ where, } i = 1, \ldots m, j = 1, \ldots n.
\]

In matrix notation, the data set is written as \( X = [x_1, \ldots, x_n] \) with \( x_j = \{ x_{ij} \}_{i=1}^m \), \( y = \{ y_i \}_{i=1}^m \) (see Fig. Feature matrix and label) and \( \beta^T = \{ \beta_j \}_{j=1}^n \). Then the matrix format equation is written as

\[
y = X \beta.
\] (9.1)

9.1.2 How to solve it?

1. Direct Methods (For more information please refer to my Prelim Notes for Numerical Analysis)
Fig. 1: Feature matrix and label

- For squared or rectangular matrices
  - Singular Value Decomposition
  - Gram-Schmidt orthogonalization
  - QR Decomposition

- For squared matrices
  - LU Decomposition
  - Cholesky Decomposition
  - Regular Splittings

2. Iterative Methods

- Stationary cases iterative method
  - Jacobi Method
  - Gauss-Seidel Method
  - Richardson Method
  - Successive Over Relaxation (SOR) Method

- Dynamic cases iterative method
  - Chebyshev iterative Method
  - Minimal residuals Method
  - Minimal correction iterative method
  - Steepest Descent Method
  - Conjugate Gradients Method
9.1.3 Ordinary Least Squares

In mathematics, (9.1) is an overdetermined system. The method of ordinary least squares can be used to find an approximate solution to overdetermined systems. For the system overdetermined system (9.1), the least squares formula is obtained from the problem

$$\min_{\beta} ||X\beta - y||,$$

the solution of which can be written with the normal equations:

$$\beta = (X^TX)^{-1}X^Ty$$

(9.3)

where $T$ indicates a matrix transpose, provided $(X^TX)^{-1}$ exists (that is, provided $X$ has full column rank).

**Note:** Actually, (9.3) can be derivated by the following way: multiply $X^T$ on side of (9.1) and then multiply $(X^TX)^{-1}$ on both side of the former result. You may also apply the Extreme Value Theorem to (9.2) and find the solution (9.3).

9.1.4 Gradient Descent

Let’s use the following hypothesis:

$$h_\beta = \beta_0 + \beta_j x_j, \text{ where } j = 1, \cdots n.$$  

Then, solving (9.2) is equivalent to minimize the following cost function:

$$J(\beta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\beta(x^{(i)}) - y^{(i)}\right)^2$$

(9.4)

**Note:** The reason why we prefer to solve (9.4) rather than (9.2) is because (9.4) is convex and it has some nice properties, such as it’s uniquely solvable and energy stable for small enough learning rate. The interested reader who has great interest in non-convex cost function (energy) case is referred to [Feng2016PSD] for more details.

9.1.6 Batch Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. It searches with the direction of the steepest descent which is defined by the negative of the gradient (see Fig. Gradient Descent in 1D and Gradient Descent in 2D for 1D and 2D, respectively) and with learning rate (search step) $\alpha$. 

9.1. Linear Regression 119
Fig. 2: Gradient Descent in 1D

Fig. 3: Gradient Descent in 2D
9.1.7 Stochastic Gradient Descent

9.1.8 Mini-batch Gradient Descent

9.1.9 Demo

- The Jupyter notebook can be download from Linear Regression which was implemented without using Pipeline.
- The Jupyter notebook can be download from Linear Regression with Pipeline which was implemented with using Pipeline.
- I will only present the code with pipeline style in the following.
- For more details about the parameters, please visit Linear Regression API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').\ 
    options(header='true', \ 
    inferschema='true').\ 
    load("../data/Advertising.csv",header=True);
```

check the data set

```python
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
| TV | Radio|Newspaper|Sales |
+-----+-----+---------+-----+
|230.1| 37.8| 69.2| 22.1 |
|44.5 |39.3 |45.1 |10.4 |
|17.2 |45.9 |69.3 |9.3 |
|151.5|41.3 |58.5 |18.5 |
|180.8|10.8 |58.4 |12.9 |
+-----+-----+---------+-----+
```

only showing top 5 rows

root
|-- TV: double (nullable = true)

(continues on next page)
You can also get the Statistical results from the data frame (Unfortunately, it only works for numerical).

df.describe().show()

Then you will get

<table>
<thead>
<tr>
<th>summary</th>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>mean</td>
<td>147.0425</td>
<td>23.264000000000024</td>
<td>30.553999999999995</td>
</tr>
<tr>
<td>stddev</td>
<td>85.85423631490805</td>
<td>14.846809176168728</td>
<td>21.77862083852283</td>
</tr>
<tr>
<td>min</td>
<td>0.7</td>
<td>0.0</td>
<td>0.3</td>
</tr>
<tr>
<td>max</td>
<td>296.4</td>
<td>49.6</td>
<td>114.0</td>
</tr>
</tbody>
</table>

3. Convert the data to dense vector (features and label)

```python
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# I provide two ways to build the features and labels

# method 1 (good for small feature):
def transData(row):
    return Row(label=row["Sales"],
               features=Vectors.dense([row["TV"],
                                       row["Radio"],
                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features', 'label'])
```

Note:

You are strongly encouraged to try my get_dummy function for dealing with the categorical
data in comple dataset.

Supervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):
    indexers = [StringIndexer(inputCol=c, outputCol=f"{c}_indexed".format(c))
                for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                              outputCol=f"{indexer.getOutputCol()}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols,
                                 outputCol="features")
```

(continues on next page)
pipeline = Pipeline(stages=indexers + encoders +
[assembler])

model=pipeline.fit(df)
data = model.transform(df)

data = data.withColumn('label',col(labelCol))

if indexCol:
    return data.select(indexCol,'features','label')
else:
    return data.select('features','label')

Unsupervised learning version:

def get_dummy(df,indexCol,categoricalCols,continuousCols):
    
    Get dummy variables and concat with continuous variables for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    
    indexers = [ StringIndexer(inputCol=c, outputCol="(0)_{0}_
    indexed".format(c))
        for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
    getOutputCol(),
        outputCol="(0)_{0}_encoded".format(indexer.
    getOutputCol()))
        for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
    getOutputCol() for encoder in encoders]
    + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders +
[assembler])

    model=pipeline.fit(df)
data = model.transform(df)
if indexCol:
    return data.select(indexCol,'features')
else:
    return data.select('features')

Two in one:

def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
              dropLast=False):
    '''
    Get dummy variables and concat with continuous variables for ml
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    
    >>> df = spark.createDataFrame(
        [(0, "a"),
         (1, "b"),
         (2, "c"),
         (3, "a"),
         (4, "a"),
         (5, "c")],
        ["id", "category"])
    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []
    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
                      labelCol)
    >>> mat.show()

    >>> ---+---+-------------------+
    | id | features          |
    ---+---+-------------------+
    | 0 | [1.0,0.0,0.0,0.0] |
    | 1 | [0.0,0.0,0.0,1.0] |
    | 2 | [0.0,1.0,0.0,0.0] |
    | 3 | [1.0,0.0,0.0,0.0] |
    | 4 | [1.0,0.0,0.0,0.0] |
    | 5 | [0.0,1.0,0.0,0.0] |
    ---+---+-------------------+
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".
format(c))

for c in categoricalCols ]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
outputCol="{0}_encoded".format(indexer.
getOutputCol()),dropLast=dropLast)

for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder
in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
# for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select(indexCol,'features','label')

elif not indexCol and labelCol:
# for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select('features','label')

elif indexCol and not labelCol:
# for unsupervised learning
return data.select(indexCol,'features')

elif not indexCol and not labelCol:
# for unsupervised learning
return data.select('features')

4. Transform the dataset to DataFrame

transformed = transData(df)
transformed.show(5)
Learning Apache Spark with Python

(continued from previous page)

Note: You will find out that all of the supervised machine learning algorithms in Spark are based on the features and label (unsupervised machine learning algorithms in Spark are based on the features). That is to say, you can play with all of the machine learning algorithms in Spark when you get ready the features and label in pipeline architecture.

5. Deal With Categorical Variables

```python
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated
# as continuous.

featureIndexer = VectorIndexer(inputCol="features", 
                               outputCol="indexedFeatures", 
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
```

Now you check your dataset with

```python
data.show(5,True)
```

you will get

```
+-----------------+-----+-----------------+
| features | label | indexedFeatures |
+-----------------+-----+-----------------+
| [230.1,37.8,69.2] | 22.1 | [230.1,37.8,69.2] |
| [44.5,39.3,45.1] | 10.4 | [44.5,39.3,45.1] |
| [17.2,45.9,69.3] | 9.3  | [17.2,45.9,69.3] |
| [151.5,41.3,58.5] | 18.5 | [151.5,41.3,58.5] |
| [180.8,10.8,58.4] | 12.9 | [180.8,10.8,58.4] |
+-----------------+-----+-----------------+
only showing top 5 rows
```

6. Split the data into training and test sets (40% held out for testing)

9.1. Linear Regression
You can check your train and test data as follows (In my opinion, it is always good to keep tracking your data during prototype phase):

```python
trainingData.show(5)
testData.show(5)
```

Then you will get

```text
+---------------+-----+---------------+
| features|label|indexedFeatures|
+---------------+-----+---------------+
| [4.1,11.6,5.7]| 3.2| [4.1,11.6,5.7]|
| [5.4,29.9,9.4]| 5.3| [5.4,29.9,9.4]|
| [7.3,28.1,41.4]| 5.5| [7.3,28.1,41.4]|
| [7.8,38.9,50.6]| 6.6| [7.8,38.9,50.6]|
| [8.6,2.1,1.0]| 4.8| [8.6,2.1,1.0]|
+---------------+-----+---------------+
only showing top 5 rows
```

```text
+----------------+-----+----------------+
| features|label|indexedFeatures|
+----------------+-----+----------------+
| [0.7,39.6,8.7]| 1.6| [0.7,39.6,8.7]|
| [8.4,27.2,2.1]| 5.7| [8.4,27.2,2.1]|
| [11.7,36.9,45.2]| 7.3| [11.7,36.9,45.2]|
| [13.2,15.9,49.6]| 5.6| [13.2,15.9,49.6]|
| [16.9,43.7,89.4]| 8.7| [16.9,43.7,89.4]|
+----------------+-----+----------------+
only showing top 5 rows
```

7. Fit Ordinary Least Square Regression Model

For more details about the parameters, please visit Linear Regression API.

```python
# Import LinearRegression class
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lr = LinearRegression()
```

8. Pipeline Architecture

```python
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, lr])

model = pipeline.fit(trainingData)
```

9. Summary of the Model
Spark has a poor summary function for data and model. I wrote a summary function which has similar format as R output for the linear regression in PySpark.

```python
def modelsummary(model):
    import numpy as np
    print("Note: the last rows are the information for Intercept")
    print("#"*30)
    print("##" ,"|" "|" "|" "|" ", "|" ")
    print("##"," Estimate | Std.Error | t Values | P-value")
    coef = np.append(list(model.coefficients),model.intercept)
    Summary=model.summary
    for i in range(len(Summary.pValues)):
        print("##”, '{:10.6f}'.format(coef[i]),
        '{:10.6f}'.format(Summary.coefficientStandardErrors[i]),
        '{:8.3f}'.format(Summary.tValues[i]),
        '{:10.6f}'.format(Summary.pValues[i]))
    print("##",'---')
    print("##",'Mean squared error: {:.6f} \ 
    % Summary.meanSquaredError, ", ", RMSE: {:.6f} \ 
    % Summary.rootMeanSquaredError )
    print("##",'Multiple R-squared: {:.6f} % Summary.r2, ", 
    Total iterations: {:.0f} % Summary.totalIterations)
```

modelsummary(model.stages[-1])

You will get the following summary results:

```
Note: the last rows are the information for Intercept
("##", "-----------------------------------------------")
("##", " Estimate | Std.Error | t Values | P-value")
("##", " 0.044186", " 0.001663", " 26.573", " 0.000000")
("##", " 0.206311", " 0.010846", " 19.022", " 0.000000")
("##", " 0.001963", " 0.007467", " 0.263", " 0.793113")
("##", " 2.596154", " 0.379550", " 6.840", " 0.000000")
("##", "---")
("##", "Mean squared error: 2.588230", ", RMSE: 1.608798")
("##", "Multiple R-squared: 0.911869", ", Total iterations: 1")
```

10. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)
```

```python
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+----------------+-----+------------------+
| features|label| prediction|
+----------------+-----+------------------+
| [0.7,39.6,8.7]| 1.6| 10.81405928637388|
| [8.4,27.2,2.1]| 5.7| 8.583086404079918|
```

(continues on next page)

9.1. Linear Regression 129
9. Evaluation

```python
from pyspark.ml.evaluation import RegressionEvaluator
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="label",
                              predictionCol="prediction",
                              metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = \$g\) \% rmse")
```

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.63114
```

You can also check the $R^2$ value for the test data:

```python
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: \(0\).format(r2_score))
```

Then you will get

```
r2_score: 0.854486655585
```

**Warning:** You should know most softwares are using different formula to calculate the $R^2$ value when no intercept is included in the model. You can get more information from the discussion at StackExchange.
9.2 Generalized linear regression

9.2.1 Introduction

9.2.2 How to solve it?

9.2.3 Demo

• The Jupyter notebook can be downloaded from Generalized Linear Regression.
• For more details about the parameters, please visit Generalized Linear Regression API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
spark = SparkSession.
  .builder.
  .appName("Python Spark regression example").
  .config("spark.some.config.option", "some-value")
  .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').
  .options(header='true',
           inferschema='true').
  .load("../data/Advertising.csv",header=True);
```

check the data set

```python
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
| TV  | Radio| Newspaper| Sales |
+-----+-----+---------+-----+
| 230.1| 37.8 | 69.2 | 22.1 |
| 44.5 | 39.3 | 45.1 | 10.4 |
| 17.2 | 45.9 | 69.3 | 9.3  |
| 151.5| 41.3 | 58.5 | 18.5 |
| 180.8| 10.8 | 58.4 | 12.9 |
+-----+-----+---------+-----+
only showing top 5 rows
root
|-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```
You can also get the Statistical results from the data frame (Unfortunately, it only works for numerical).

```python
df.describe().show()
```

Then you will get

```
+-------+-----------------+-----------------+-----------------+-----------------+
| summary| TV| Radio| Newspaper|
+-------+-----------------+-----------------+-----------------+-----------------+
| count | 200| 200| 200| 200|
| mean  | 147.0425| 23.264000000000024| 30.553999999999995| 14.0220217456565710477|
| stddev| 85.85423631490805| 14.846809176168728| 21.77862083852283| 5.217456565710477|
| min   | 0.7| 0.0| 0.3| 1.6|
| max   | 296.4| 49.6| 114.0| 27.0|
+-------+-----------------+-----------------+-----------------+-----------------+
```

3. Convert the data to dense vector (features and label)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

Supervised learning version:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="\{0\}_indexed".format(c))
        for c in categoricalCols ]

    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
        outputCol="\{0\}_encoded".format(indexer.getOutputCol()))
        for indexer in indexers ]
```

(continues on next page)
assembler = VectorAssembler(inputCols=[encoder.
    → getOutputCol() for encoder in encoders]
    + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders +
    → [assembler])

model = pipeline.fit(df)
data = model.transform(df)
data = data.withColumn('label',col(labelCol))

if indexCol:
    return data.select(indexCol, 'features', 'label')
else:
    return data.select('features', 'label')

Unsupervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols):
    """
    Get dummy variables and concat with continuous variables
    for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    """

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
        → indexed".format(c))
        for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
        → getOutputCol(),
        → outputCol="{0}_encoded".format(indexer.
        → getOutputCol()))
        for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
        → getOutputCol() for encoder in encoders]
        + continuousCols, outputCol="features")

(continues on next page)
pipeline = Pipeline(stages=indexers + encoders + ...
  [assembler])

model=pipeline.fit(df)
data = model.transform(df)

if indexCol:
    return data.select(indexCol,'features')
else:
    return data.select('features')

Two in one:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
  dropLast=False):
    
    Get dummy variables and concat with continuous variables for ml 
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email: von198@gmail.com

>>> df = spark.createDataFrame(
    [(0, "a"),
     (1, "b"),
     (2, "c"),
     (3, "a"),
     (4, "a"),
     (5, "c")
    ], ["id", "category"])

>>> indexCol = 'id'
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
                      labelCol)
>>> mat.show()

    +---+-------------+
    | id| features    |
    +---+-------------+
    | 0| [1.0,0.0,0.0]|
    (continues on next page)
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                        for c in categoricalCols]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                          outputCol="{0}_encoded".format(indexer.
                          getOutputCol()), dropLast=dropLast)
                        for indexer in indexers]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                  + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
elif not indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select('features', 'label')
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol, 'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')

from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

9.2. Generalized linear regression
# I provide two ways to build the features and labels

# method 1 (good for small feature):
def transData(row):
    return Row(label=row["Sales"],
               features=Vectors.dense([row["TV"],
                                        row["Radio"],
                                        row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features', 'label'])

transformed= transData(df)
transformed.show(5)

+-----------------+-----+
| features|label|
+-----------------+-----+
| [230.1, 37.8, 69.2]| 22.1|
| [44.5, 39.3, 45.1]| 10.4|
| [17.2, 45.9, 69.3]|  9.3|
| [151.5, 41.3, 58.5]| 18.5|
| [180.8, 10.8, 58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows

Note: You will find out that all of the machine learning algorithms in Spark are based on the features and label. That is to say, you can play with all of the machine learning algorithms in Spark when you get ready the features and label.

4. Convert the data to dense vector

```python
# convert the data to dense vector
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).
                       toDF(['label','features'])

from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

data= transData(df)
data.show()
```

5. Deal with the Categorical variables
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4
# distinct values are treated as continuous.

featureIndexer = VectorIndexer(inputCol="features", 
                               outputCol="indexedFeatures",
                               maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)

When you check you data at this point, you will get

+-----------------+-----+-----------------+
| features|label| indexedFeatures|
+-----------------+-----+-----------------+
| [230.1,37.8,69.2]| 22.1|[230.1,37.8,69.2]|
| [44.5,39.3,45.1]| 10.4| [44.5,39.3,45.1]|
| [17.2,45.9,69.3]| 9.3| [17.2,45.9,69.3]|
| [151.5,41.3,58.5]| 18.5|[151.5,41.3,58.5]|
| [180.8,10.8,58.4]| 12.9|[180.8,10.8,58.4]|
+-----------------+-----+-----------------+
only showing top 5 rows

6. Split the data into training and test sets (40% held out for testing)

# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])

You can check your train and test data as follows (In my opinion, it is always to good to keep tracking your data during prototype phase):

trainingData.show(5)
testData.show(5)

Then you will get

+-----------------+-----+-----------------+
| features|label| indexedFeatures|
+-----------------+-----+-----------------+
| [5.4,29.9,9.4]| 5.3| [5.4,29.9,9.4]|
| [7.8,38.9,50.6]| 6.6| [7.8,38.9,50.6]|
| [8.4,27.2,2.1]| 5.7| [8.4,27.2,2.1]|
| [8.7,48.9,75.0]| 7.2| [8.7,48.9,75.0]|
| [11.7,36.9,45.2]| 7.3| [11.7,36.9,45.2]|
+-----------------+-----+-----------------+
only showing top 5 rows

(continues on next page)
7. Fit Generalized Linear Regression Model

```python
# Import LinearRegression class
from pyspark.ml.regression import GeneralizedLinearRegression

# Define LinearRegression algorithm
glr = GeneralizedLinearRegression(family="gaussian", link="identity",
maxIter=10, regParam=0.3)
```

8. Pipeline Architecture

```python
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, glr])

model = pipeline.fit(trainingData)
```

9. Summary of the Model

Spark has a poor summary function for data and model. I wrote a summary function which has similar format as R output for the linear regression in PySpark.

```python
def modelsummary(model):
    import numpy as np
    print("Note: the last rows are the information for Intercept")
    print("##" , "-------------------------------------------------")
    print("##" , " Estimate | Std.Error | t Values | P-value")
    coef = np.append(list(model.coefficients),model.intercept)
    Summary=model.summary
    for i in range(len(Summary.pValues)):
        print("##",'{:10.6f}'.format(coef[i]),
         '{:10.6f}'.format(Summary.coefficientStandardErrors[i]),
         '{:8.3f}'.format(Summary.tValues[i]),
         '{:10.6f}'.format(Summary.pValues[i]))
    print("##",'---')
    #print("##","Mean squared error: % .6f" 
    # % Summary.meanSquaredError, ", RMSE: % .6f" 
    # % Summary.rootMeanSquaredError 
    #print("##","Multiple R-squared: %f % Summary.r2", 
    # Total iterations: %i"% Summary.totalIterations)```
modelsummary(model.stages[-1])

You will get the following summary results:

<table>
<thead>
<tr>
<th>Estimate</th>
<th>Std.Error</th>
<th>t Values</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.042857</td>
<td>0.001668</td>
<td>25.692</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.199922</td>
<td>0.009881</td>
<td>20.232</td>
<td>0.000000</td>
</tr>
<tr>
<td>-0.001957</td>
<td>0.006917</td>
<td>-0.283</td>
<td>0.777757</td>
</tr>
<tr>
<td>3.007515</td>
<td>0.406389</td>
<td>7.401</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

10. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.7,39.6,8.7]</td>
<td>1.6</td>
<td>10.937383732327625</td>
</tr>
<tr>
<td>[4.1,11.6,5.7]</td>
<td>3.2</td>
<td>5.491166258750164</td>
</tr>
<tr>
<td>[7.3,28.1,41.4]</td>
<td>5.5</td>
<td>8.8571603947873</td>
</tr>
<tr>
<td>[8.6,2.1,1.0]</td>
<td>4.8</td>
<td>3.793966281660073</td>
</tr>
<tr>
<td>[17.2,4.1,31.6]</td>
<td>5.9</td>
<td>4.502507124763654</td>
</tr>
</tbody>
</table>

only showing top 5 rows

11. Evaluation

```python
from pyspark.ml.evaluation import RegressionEvaluator

# Select (prediction, true label) and compute test error

evaluator = RegressionEvaluator(labelCol="label",
predictionCol="prediction",
metricName="rmse")

rmse = evaluator.evaluate(predictions)

print("Root Mean Squared Error (RMSE) on test data = \$g\ \$ \ rmse")
```

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.89857
```

(continues on next page)
Then you will get the $R^2$ value:

```
r2_score: 0.87707391843
```

9.3 Decision tree Regression

9.3.1 Introduction

9.3.2 How to solve it?

9.3.3 Demo

- The Jupyter notebook can be download from Decision Tree Regression.
- For more details about the parameters, please visit Decision Tree Regressor API.
  
1. Set up spark context and SparkSession

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```
df = spark.read.format('com.databricks.spark.csv').
    options(header='true', 
            inferschema='true').
    load("../data/Advertising.csv",header=True);
```

check the data set

```
df.show(5,True)
df.printSchema()
```

Then you will get

```
+-----+-----+---------+-----+
| TV  |Radio|Newspaper|Sales |
+-----+-----+---------+-----+
|230.1| 37.8| 69.2| 22.1 |
```

(continues on next page)
Learning Apache Spark with Python

| 44.5 | 39.3 | 45.1 | 10.4 |
| 17.2 | 45.9 | 69.3 | 9.3  |
|151.5 | 41.3 | 58.5 | 18.5 |
|180.8 | 10.8 | 58.4 | 12.9 |

only showing top 5 rows

root
|-- TV: double (nullable = true)
|-- Radio: double (nullable = true)
|-- Newspaper: double (nullable = true)
|-- Sales: double (nullable = true)

You can also get the Statistical results from the data frame (Unfortunately, it only works for numerical).

```python
df.describe().show()
```

Then you will get

```
+-------+-----------------+------------------+------------------+-------------
|summary| TV| Radio| Newspaper| Sales|
+-------+-----------------+------------------+------------------+-------------
| count| 200| 200| 200| 200|
| mean| 147.0425|23.264000000000024|30.553999999999995|14.02250000000003|
| stddev|85.85423631490805|14.846809176168728| 21.77862083852283| 5.21745656710477|
| min| 0.7| 0.0| 0.3| 1.6|
| max| 296.4| 49.6| 114.0| 27.0|
+-------+-----------------+------------------+------------------+-------------
```

3. Convert the data to dense vector (features and label)

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

Supervised learning version:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol):
    from pyspark.ml import Pipeline
```

9.3. Decision tree Regression
from pyspark.ml.feature import StringIndexer, 
  OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_
  indexed".format(c))
  for c in categoricalCols ]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.
  getOutputCol(),
  outputCol="{0}_encoded".format(indexer.
  getOutputCol()))
  for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.
  getOutputCol() for encoder in encoders] +
  continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders +
  [assembler])

model = pipeline.fit(df)
data = model.transform(df)
data = data.withColumn('label',col(labelCol))

return data.select(indexCol,'features','label')

Unsupervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols): 
  '''
  Get dummy variables and concat with continuous variables.
  for unsupervised learning.
  :param df: the dataframe
  :param categoricalCols: the name list of the categorical data
  :param continuousCols: the name list of the numerical data
  :return k: feature matrix
  :
  :author: Wenqiang Feng
  :email: von198@gmail.com
  '''

  indexers = [StringIndexer(inputCol=c, outputCol="{0}_
    indexed".format(c))
    for c in categoricalCols ]

  # default setting: dropLast=True
```
encoders = [ OneHotEncoder(inputCol=indexer.
˓→getOutputCol(),
    outputCol="{0}_encoded".format(indexer.
       →getOutputCol()))
       for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.
       →getOutputCol() for encoder in encoders]
       + continuousCols, outputCol= "features")

pipeline = Pipeline(stages=indexers + encoders +
       →[assembler])

model=pipeline.fit(df)
data = model.transform(df)
return data.select(indexCol,'features')

Two in one:

def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
       →dropLast=False):

    ```
    Get dummy variables and concat with continuous variables for ml
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    
    :author: Wenqiang Feng
    :email: von198@gmail.com
    ```

    ```
    >>> df = spark.createDataFrame([[(0, "a"),
        (1, "b"),
        (2, "c"),
        (3, "a"),
        (4, "a"),
        (5, "c")
    ], ["id", "category"])
    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []
    >>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
       →labelCol)  

9.3. Decision tree Regression 143
```python
>>> mat.show()
>>> 
<table>
<thead>
<tr>
<th>id</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>1</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>2</td>
<td>[0.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>3</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>4</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>5</td>
<td>[0.0, 1.0, 0.0]</td>
</tr>
</tbody>
</table>

from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
   VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
   for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
   outputCol="{0}_encoded".format(indexer.
   getOutputCol()),dropLast=dropLast)
   for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
   # for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select(indexCol,'features','label')
elif not indexCol and labelCol:
   # for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select('features','label')
elif indexCol and not labelCol:
   # for unsupervised learning
return data.select(indexCol,'features')
elif not indexCol and not labelCol:

(continues on next page)```
Learning Apache Spark with Python

# for unsupervised learning
return data.select('features')

from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# I provide two ways to build the features and labels

# method 1 (good for small feature):
def transData(row):
    return Row(label=row["Sales"],
               features=Vectors.dense([row["TV"],
                                       row["Radio"],
                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF(['features', 'label'])

transformed = transData(df)
transformed.show(5)

+-----------------+-----+
| features|label|
+-----------------+-----+
| [230.1,37.8,69.2]| 22.1|
| [44.5,39.3,45.1]| 10.4|
| [17.2,45.9,69.3]| 9.3 |
| [151.5,41.3,58.5]| 18.5|
| [180.8,10.8,58.4]| 12.9|
+-----------------+-----+
only showing top 5 rows

Note: You will find out that all of the machine learning algorithms in Spark are based on the features and label. That is to say, you can play with all of the machine learning algorithms in Spark when you get ready the features and label.

4. Convert the data to dense vector

# convert the data to dense vector
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).toDF(['label', 'features'])

transformed = transData(df)
transformed.show(5)

9.3. Decision tree Regression 145
5. Deal with the Categorical variables

```python
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4
distinct values are treated as continuous.
featureIndexer = VectorIndexer(inputCol="features",
    outputCol="indexedFeatures",
    maxCategories=4).fit(transformed)
data = featureIndexer.transform(transformed)
```

When you check your data at this point, you will get

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedFeatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>[230.1, 37.8, 69.2]</td>
<td>22.1</td>
<td>[230.1, 37.8, 69.2]</td>
</tr>
<tr>
<td>[44.5, 39.3, 45.1]</td>
<td>10.4</td>
<td>[44.5, 39.3, 45.1]</td>
</tr>
<tr>
<td>[17.2, 45.9, 69.3]</td>
<td>9.3</td>
<td>[17.2, 45.9, 69.3]</td>
</tr>
<tr>
<td>[151.5, 41.3, 58.5]</td>
<td>18.5</td>
<td>[151.5, 41.3, 58.5]</td>
</tr>
<tr>
<td>[180.8, 10.8, 58.4]</td>
<td>12.9</td>
<td>[180.8, 10.8, 58.4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
only showing top 5 rows

6. Split the data into training and test sets (40% held out for testing)

```python
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])

You can check your train and test data as follows (In my opinion, it is always to good to keep tracking your
data during prototype phase):

```python
trainingData.show(5)
testData.show(5)
```

Then you will get

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedFeatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4.1, 11.6, 5.7]</td>
<td>3.2</td>
<td>[4.1, 11.6, 5.7]</td>
</tr>
<tr>
<td>[7.3, 28.1, 41.4]</td>
<td>5.5</td>
<td>[7.3, 28.1, 41.4]</td>
</tr>
<tr>
<td>[8.4, 27.2, 2.1]</td>
<td>5.7</td>
<td>[8.4, 27.2, 2.1]</td>
</tr>
<tr>
<td>[8.6, 2.1, 1.0]</td>
<td>4.8</td>
<td>[8.6, 2.1, 1.0]</td>
</tr>
<tr>
<td>[8.7, 48.9, 75.0]</td>
<td>7.2</td>
<td>[8.7, 48.9, 75.0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(continues on next page)
only showing top 5 rows

| features | label | indexedFeatures |
|----------------+-----+----------------|
| [0.7, 39.6, 8.7] | 1.6 | [0.7, 39.6, 8.7] |
| [5.4, 29.9, 9.4] | 5.3 | [5.4, 29.9, 9.4] |
| [7.8, 38.9, 50.6] | 6.6 | [7.8, 38.9, 50.6] |
| [17.2, 45.9, 69.3] | 9.3 | [17.2, 45.9, 69.3] |
| [18.7, 12.1, 23.4] | 6.7 | [18.7, 12.1, 23.4] |

7. Fit Decision Tree Regression Model

```python
from pyspark.ml.regression import DecisionTreeRegressor

# Train a DecisionTree model.
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")
```

8. Pipeline Architecture

```python
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, dt])

model = pipeline.fit(trainingData)
```

9. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

<table>
<thead>
<tr>
<th>prediction</th>
<th>label</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>1.6</td>
<td>[0.7, 39.6, 8.7]</td>
</tr>
<tr>
<td>7.3</td>
<td>5.3</td>
<td>[5.4, 29.9, 9.4]</td>
</tr>
<tr>
<td>7.2</td>
<td>6.6</td>
<td>[7.8, 38.9, 50.6]</td>
</tr>
<tr>
<td>8.64</td>
<td>9.3</td>
<td>[17.2, 45.9, 69.3]</td>
</tr>
<tr>
<td>6.45</td>
<td>6.7</td>
<td>[18.7, 12.1, 23.4]</td>
</tr>
</tbody>
</table>

10. Evaluation

```python
from pyspark.ml.evaluation import RegressionEvaluator

# Evaluation
```

(continues on next page)
# Select (prediction, true label) and compute test error

evaluator = RegressionEvaluator(labelCol="label",
                                predictionCol="prediction",
                                metricName="rmse")

rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)

The final Root Mean Squared Error (RMSE) is as follows:

```
Root Mean Squared Error (RMSE) on test data = 1.50999
```

```python
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {0}'.format(r2_score))
```

Then you will get the $R^2$ value:

```
r2_score: 0.911024318967
```

You may also check the importance of the features:

```
model.stages[1].featureImportances
```

The you will get the weight for each features:

```
SparseVector(3, {0: 0.6811, 1: 0.3187, 2: 0.0002})
```

## 9.4 Random Forest Regression

### 9.4.1 Introduction

### 9.4.2 How to solve it?

### 9.4.3 Demo

- The Jupyter notebook can be download from [Random Forest Regression](#).
- For more details about the parameters, please visit [Random Forest Regressor API](#).

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
spark = SparkSession \
```

(continues on next page)
2. Load dataset

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder 
    .appName("Python Spark RandomForest Regression example") 
    .config("spark.some.config.option", "some-value") 
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').
    options(header='true', inferschema='true').
    load("../data/Advertising.csv",header=True);

df.show(5,True)
df.printSchema()
```

```
+-----+-----+---------+-----+
| TV  | Radio| Newspaper| Sales|
+-----+-----+---------+-----+
|230.1| 37.8| 69.2    | 22.1|
| 44.5| 39.3| 45.1    | 10.4|
| 17.2| 45.9| 69.3    | 9.3 |
|151.5| 41.3| 58.5    | 18.5|
|180.8| 10.8| 58.4    | 12.9|
+-----+-----+---------+-----+

only showing top 5 rows

root
|-- TV: double (nullable = true)
|-- Radio: double (nullable = true)
|-- Newspaper: double (nullable = true)
|-- Sales: double (nullable = true)
```

```python
df.describe().show()
```

```
+-------+-----------------+------------------+------------------+
|summary| TV   | Radio| Newspaper| Sales      |
+-------+-------+-------+-----------+------------+
| count  | 200   | 200   | 200  | 200  |
| mean   | 147.0425| 23.264000000000024| 30.553999999999995| 14.02225000000003 |
| stddev | 85.85423631490805| 14.846809176168728| 21.77862083852283| 5.217456565710477 |
| min    | 0.7   | 0.0   | 0.3   | 1.6   |
| max    | 296.4 | 49.6  | 114.0 | 27.0  |
+-------+-------+-------+-----------+------------+
```

(continues on next page)
3. Convert the data to dense vector (**features** and **label**)

**Note:**

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in incomplete dataset.

**Supervised learning version:**

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
    OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                              outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

**Unsupervised learning version:**

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
    Get dummy variables and concat with continuous variables
    for unsupervised learning.
    '''
```

(continues on next page)
indexers = [ StringIndexer(inputCol=c, outputCol="\{0\}_ indexed".format(c))
   for c in categoricalCols ]

# default setting: dropLast=True
coders = [ OneHotEncoder(inputCol=indexer.
   \getOutputCol() , outputCol="\{0\}_encoded".format(indexer.
   \getOutputCol()))
   for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.
   \getOutputCol() for encoder in encoders] + continuousCols, outputCol=
   "features")

pipeline = Pipeline(stages=indexers + encoders +
   [assembler])

model=pipeline.fit(df)
data = model.transform(df)

return data.select(indexCol,'features')
>>> df = spark.createDataFrame(
    [(0, "a"),
     (1, "b"),
     (2, "c"),
     (3, "a"),
     (4, "a"),
     (5, "c")
   ], ["id", "category"])

>>> indexCol = 'id'
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
                  labelCol)

>>> mat.show()

```
+---+-------------+
<table>
<thead>
<tr>
<th>id</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1.0,0.0,0.0]</td>
</tr>
<tr>
<td>1</td>
<td>[0.0,0.0,1.0]</td>
</tr>
<tr>
<td>2</td>
<td>[0.0,1.0,0.0]</td>
</tr>
<tr>
<td>3</td>
<td>[1.0,0.0,0.0]</td>
</tr>
<tr>
<td>4</td>
<td>[1.0,0.0,0.0]</td>
</tr>
<tr>
<td>5</td>
<td>[0.0,1.0,0.0]</td>
</tr>
</tbody>
</table>
+---+-------------+
```

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
                          VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
                          format(c))
              for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                            outputCol="{0}_encoded".format(indexer.
                            getOutputCol()),dropLast=dropLast)
             for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                             + continuousCols, outputCol="features")
```
```python
pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
elif not indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label', col(labelCol))
    return data.select('features', 'label')
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol, 'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')
```

---

```python
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# convert the data to dense vector

from pyspark.ml.linalg import Vectors

def transData(row):
    return Row(label=row['Sales'], 
               features=Vectors.dense([row['TV'], 
                                       row['Radio'], 
                                       row['Newspaper']]))

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]), r[-1]]).toDF([ 'features', 'label'])
```

4. Convert the data to dense vector

```python
transformed = transData(df)
transformed.show(5)
```

```
+-----------------+-----+
| features | label|
+-----------------+-----+
| [230.1, 37.8, 69.2] | 22.1|
| [44.5, 39.3, 45.1]  | 10.4|
| [17.2, 45.9, 69.3]  |  9.3|
| [151.5, 41.3, 58.5] | 18.5|
| [180.8, 10.8, 58.4] | 12.9|
+-----------------+-----+
only showing top 5 rows
```

5. Deal with the Categorical variables

---

9.4. Random Forest Regression
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
data.show(5,True)

6. Split the data into training and test sets (40% held out for testing)

# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5)
testData.show(5)
Learning Apache Spark with Python

7. Fit RandomForest Regression Model

```
# Import LinearRegression class
from pyspark.ml.regression import RandomForestRegressor

# Define LinearRegression algorithm
rf = RandomForestRegressor() # featuresCol="indexedFeatures", numTrees=2, ...
    → maxDepth=2, seed=42

Note: If you decide to use the indexedFeatures features, you need to add the parameter featuresCol="indexedFeatures".
```

8. Pipeline Architecture

```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
model = pipeline.fit(trainingData)
```

9. Make predictions

```
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features", "label", "prediction").show(5)
```

```
+---------------+-----+------------------+
| features|label| prediction|
+---------------+-----+------------------+
| [4.1,11.6,5.7]| 3.2| 8.155439814814816|
| [5.4,29.9,9.4]| 5.3|10.412769901394899|
| [7.3,28.1,41.4]| 5.5|12.13735648148148|
| [7.8,38.9,50.6]| 6.6|11.321796703296704|
| [8.4,27.2,2.1]| 5.7|12.071421957671957|
+---------------+-----+------------------+
only showing top 5 rows
```

10. Evaluation

```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = \%g\" % rmse)
```

```
Root Mean Squared Error (RMSE) on test data = 2.35912
```

```
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {:4.3f}'.format(r2_score))
```

9.4. Random Forest Regression
**11. Feature importances**

model.stages[-1].featureImportances

```
SparseVector(3, {0: 0.4994, 1: 0.3196, 2: 0.181})
```

model.stages[-1].trees

```
[DecisionTreeRegressionModel (uid=dtr_c75f1c75442c) of depth 5 with 43 nodes,
 DecisionTreeRegressionModel (uid=dtr_70fc2d441581) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_bc8464f545a7) of depth 5 with 31 nodes,
 DecisionTreeRegressionModel (uid=dtr_a8a7e5367154) of depth 5 with 59 nodes,
 DecisionTreeRegressionModel (uid=dtr_3ea01314fcbc) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_be9a04ac22a6) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_38610d47328a) of depth 5 with 51 nodes,
 DecisionTreeRegressionModel (uid=dtr_bf14ae0ad3b) of depth 5 with 49 nodes,
 DecisionTreeRegressionModel (uid=dtr_cde24eb6bb6) of depth 5 with 39 nodes,
 DecisionTreeRegressionModel (uid=dtr_a1fc9bd4fbeb) of depth 5 with 57 nodes,
 DecisionTreeRegressionModel (uid=dtr_c0798d6db1ba) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_c0798d6db1ba) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_fd00e3a070ad) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_9d01d5fb8604) of depth 5 with 45 nodes,
 DecisionTreeRegressionModel (uid=dtr_8bd88bddd642) of depth 5 with 41 nodes,
 DecisionTreeRegressionModel (uid=dtr_e53b7bae30f8) of depth 5 with 49 nodes,
 DecisionTreeRegressionModel (uid=dtr_808a869db21c) of depth 5 with 47 nodes,
 DecisionTreeRegressionModel (uid=dtr_64d016bce9b0) of depth 5 with 33 nodes,
 DecisionTreeRegressionModel (uid=dtr_0891055f894) of depth 5 with 55 nodes,
 DecisionTreeRegressionModel (uid=dtr_19c8bbad26c2) of depth 5 with 51 nodes]
```

---

### 9.5 Gradient-boosted tree regression

**9.5.1 Introduction**

**9.5.2 How to solve it?**

**9.5.3 Demo**

- The Jupyter notebook can be download from Gradient-boosted tree regression.
- For more details about the parameters, please visit Gradient boosted tree API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession \n```

(continues on next page)
2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').
    options(header='true',
            inferschema='true').
    load("../data/Advertising.csv",header=True);

df.show(5,True)

df.printSchema()
```

```
+-----+-----+---------+-----+
| TV  | Radio| Newspaper| Sales|
|-----+-----+---------+-----|
|230.1| 37.8 | 69.2 | 22.1 |
|44.5 | 39.3 | 45.1 | 10.4 |
|17.2 | 45.9 | 69.3 | 9.3  |
|151.5| 41.3 | 58.5 | 18.5 |
|180.8| 10.8 | 58.4 | 12.9 |
+-----+-----+---------+-----+
only showing top 5 rows
```

```
root
|-- TV: double (nullable = true)
|-- Radio: double (nullable = true)
|-- Newspaper: double (nullable = true)
|-- Sales: double (nullable = true)
```

```python
df.describe().show()
```

```
<table>
<thead>
<tr>
<th>summary</th>
<th>TV</th>
<th>Radio</th>
<th>Newspaper</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>mean</td>
<td>147.0425</td>
<td>23.264</td>
<td>30.554</td>
</tr>
<tr>
<td>stddev</td>
<td>85.854</td>
<td>14.846</td>
<td>21.778</td>
</tr>
<tr>
<td>min</td>
<td>0.7</td>
<td>0.0</td>
<td>0.3</td>
</tr>
<tr>
<td>max</td>
<td>296.4</td>
<td>49.6</td>
<td>114.0</td>
</tr>
</tbody>
</table>
```

(continues on next page)
3. Convert the data to dense vector (**features** and **label**)

**Note:**

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in comple dataset.

**Supervised learning version:**

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
    OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                               outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label', col(labelCol))

    return data.select(indexCol, 'features', 'label')
```

**Unsupervised learning version:**

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols):
    
    Get dummy variables and concat with continuous variables for unsupervised learning.

    return
```

(continues on next page)
Two in one:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol, dropLast=False):
    
    Get dummy variables and concat with continuous variables for ml modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email: von198@gmail.com
```

9.5. Gradient-boosted tree regression
>>> df = spark.createDataFrame(
    [(0, "a"),
     (1, "b"),
     (2, "c"),
     (3, "a"),
     (4, "a"),
     (5, "c")
], ["id", "category"])

>>> indexCol = 'id'
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol)
>>> mat.show()

>>> ---+-------------+
     | id| features |
     +---+-------------+
     | 0| [1.0,0.0,0.0] |  
     | 1| [0.0,0.0,1.0] |  
     | 2| [0.0,1.0,0.0] |  
     | 3| [1.0,0.0,0.0] |  
     | 4| [1.0,0.0,0.0] |  
     | 5| [0.0,1.0,0.0] |  
     +---+-------------+

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,
    VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
               for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                                outputCol="{0}_encoded".format(indexer.getOutputCol()),
                                dropLast=dropLast)
                 for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                                + continuousCols, outputCol="features")
pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
elif not indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select('features', 'label')
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol, 'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')

from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# convert the data to dense vector

def transData(row):
    return Row(label=row['Sales'],
               features=Vectors.dense([row['TV'],
                                      row['Radio'],
                                      row['Newspaper']]))

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]), r[-1]]).toDF(['features', 'label'])

4. Convert the data to dense vector

transformed = transData(df)
transformed.show(5)

+-----------------+-----+
| features | label |
+-----------------+-----+
| [230.1, 37.8, 69.2] | 22.1 |
| [44.5, 39.3, 45.1]  | 10.4 |
| [17.2, 45.9, 69.3]  | 9.3  |
| [151.5, 41.3, 58.5] | 18.5 |
| [180.8, 10.8, 58.4] | 12.9 |
+-----------------+-----+
only showing top 5 rows

5. Deal with the Categorical variables

9.5. Gradient-boosted tree regression
from pyspark.ml import Pipeline
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

featureIndexer = VectorIndexer(inputCol="features", 
    outputCol="indexedFeatures",
    maxCategories=4).fit(transformed)

data = featureIndexer.transform(transformed)
data.show(5,True)

6. Split the data into training and test sets (40% held out for testing)

# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5)
testData.show(5)
7. Fit RandomForest Regression Model

```python
# Import LinearRegression class
from pyspark.ml.regression import GBTRegressor

# Define LinearRegression algorithm
rf = GBTRegressor(numTrees=2, maxDepth=2, seed=42)
```

**Note:** If you decide to use the `indexedFeatures` features, you need to add the parameter `featuresCol="indexedFeatures"`.

8. Pipeline Architecture

```python
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
model = pipeline.fit(trainingData)
```

9. Make predictions

```python
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label", "prediction").show(5)
```

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.8,38.9,50.6]</td>
<td>6.6</td>
<td>6.836040343319862</td>
</tr>
<tr>
<td>[8.6,2.1,1.0]</td>
<td>4.8</td>
<td>5.652202764688849</td>
</tr>
<tr>
<td>[8.7,48.9,75.0]</td>
<td>7.2</td>
<td>6.908750296855572</td>
</tr>
<tr>
<td>[13.1,0.4,25.6]</td>
<td>5.3</td>
<td>5.784020210692574</td>
</tr>
<tr>
<td>[19.6,20.1,17.0]</td>
<td>7.6</td>
<td>6.8678921062629295</td>
</tr>
</tbody>
</table>

only showing top 5 rows

10. Evaluation

```python
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = \%g \% rmse")
```

```
Root Mean Squared Error (RMSE) on test data = 1.36939
```

```python
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {:.4f}'.format(r2_score))
```

```
r2_score: 0.9530
```
r2_score: 0.932

11. Feature importances

```python
model.stages[-1].featureImportances
```

SparseVector(3, {0: 0.3716, 1: 0.3525, 2: 0.2759})

```python
model.stages[-1].trees
```

```
[DecisionTreeRegressionModel (uid=dtr_7f5cd2ef7cb6) of depth 5 with 61 nodes,
  DecisionTreeRegressionModel (uid=dtr_ef3ab6baeac9) of depth 5 with 39 nodes,
  DecisionTreeRegressionModel (uid=dtr_07c6e3cf3819) of depth 5 with 45 nodes,
  DecisionTreeRegressionModel (uid=dtr_ce724af79a2b) of depth 5 with 47 nodes,
  DecisionTreeRegressionModel (uid=dtr_d149ecc71658) of depth 5 with 55 nodes,
  DecisionTreeRegressionModel (uid=dtr_d3a79bdea516) of depth 5 with 43 nodes,
  DecisionTreeRegressionModel (uid=dtr_7abc1a337844) of depth 5 with 51 nodes,
  DecisionTreeRegressionModel (uid=dtr_480834b46d8f) of depth 5 with 33 nodes,
  DecisionTreeRegressionModel (uid=dtr_0cbd1ea3874) of depth 5 with 39 nodes,
  DecisionTreeRegressionModel (uid=dtr_4808ac71a204) of depth 5 with 57 nodes,
  DecisionTreeRegressionModel (uid=dtr_2ceb9e8deb45) of depth 5 with 47 nodes,
  DecisionTreeRegressionModel (uid=dtr_cc334e84e9a2) of depth 5 with 57 nodes,
  DecisionTreeRegressionModel (uid=dtr_a665c562929e) of depth 5 with 41 nodes,
  DecisionTreeRegressionModel (uid=dtr_2999b1ff2dc) of depth 5 with 45 nodes,
  DecisionTreeRegressionModel (uid=dtr_29965cbe8cfc) of depth 5 with 55 nodes,
  DecisionTreeRegressionModel (uid=dtr_731df51b0ad) of depth 5 with 41 nodes,
  DecisionTreeRegressionModel (uid=dtr_354cf33424da) of depth 5 with 51 nodes,
  DecisionTreeRegressionModel (uid=dtr_4230f200b1c0) of depth 5 with 41 nodes,
  DecisionTreeRegressionModel (uid=dtr_3279cddc1e1d) of depth 5 with 45 nodes,
  DecisionTreeRegressionModel (uid=dtr_f474a99ff06e) of depth 5 with 55 nodes]
```
In mathematics, statistics, and computer science, particularly in the fields of machine learning and inverse problems, regularization is a process of introducing additional information in order to solve an ill-posed problem or to prevent overfitting (Wikipedia Regularization).

Due to the sparsity within our data, our training sets will often be ill-posed (singular). Applying regularization to the regression has many advantages, including:

1. Converting ill-posed problems to well-posed by adding additional information via the penalty parameter $\lambda$

2. Preventing overfitting

3. Variable selection and the removal of correlated variables (Glmnet Vignette). The Ridge method shrinks the coefficients of correlated variables while the LASSO method picks one variable and discards the others. The elastic net penalty is a mixture of these two; if variables are correlated in groups then $\alpha = 0.5$ tends to select the groups as in or out. If $\alpha$ is close to 1, the elastic net performs much like the LASSO method and removes any degeneracies and wild behavior caused by extreme correlations.

### 10.1 Ordinary least squares regression

$$\min_{\beta \in \mathbb{R}^n} \frac{1}{n} \| X\beta - y \|^2$$

When $\lambda = 0$ (i.e. `regParam = 0`), then there is no penalty.

```python
 LinearRegression(featuresCol="features", labelCol="label", predictionCol= "prediction", maxIter=100, regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,
 standardization=True, solver="auto", weightCol=None, aggregationDepth=2)
```

### 10.2 Ridge regression

$$\min_{\beta \in \mathbb{R}^n} \frac{1}{n} \| X\beta - y \|^2 + \lambda \| \beta \|^2$$
When \( \lambda > 0 \) (i.e. \( \text{regParam} > 0 \)) and \( \alpha = 0 \) (i.e. \( \text{elasticNetParam} = 0 \)), then the penalty is an L2 penalty.

```python
LinearRegression(featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100,
regParam=0.1, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,
standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

### 10.3 Least Absolute Shrinkage and Selection Operator (LASSO)

\[
\min_{\beta \in \mathbb{R}^n} \frac{1}{n} \|X\beta - y\|^2 + \lambda \|\beta\|_1
\]

When \( \lambda > 0 \) (i.e. \( \text{regParam} > 0 \)) and \( \alpha = 1 \) (i.e. \( \text{elasticNetParam} = 1 \)), then the penalty is an L1 penalty.

```python
LinearRegression(featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100,
regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,
standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```

### 10.4 Elastic net

\[
\min_{\beta \in \mathbb{R}^n} \frac{1}{n} \|X\beta - y\|^2 + \lambda (\alpha \|\beta\|_1 + (1 - \alpha) \|\beta\|_2^2), \alpha \in (0, 1)
\]

When \( \lambda > 0 \) (i.e. \( \text{regParam} > 0 \)) and \( \text{elasticNetParam} \in (0, 1) \) (i.e. \( \alpha \in (0, 1) \)), then the penalty is an L1 + L2 penalty.

```python
LinearRegression(featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100,
regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True,
standardization=True, solver="auto",
weightCol=None, aggregationDepth=2)
```
Chinese proverb

Birds of a feather flock together. – old Chinese proverb

11.1 Binomial logistic regression

11.1.1 Introduction

11.1.2 Demo

- The Jupyter notebook can be download from Logistic Regression.
- For more details, please visit Logistic Regression API.

Note: In this demo, I introduced a new function get_dummy to deal with the categorical data. I highly recommend you to use my get_dummy function in the other cases. This function will save a lot of time for you.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession
 .builder
 .appName("Python Spark Logistic Regression example")
 .config("spark.some.config.option", "some-value")
 .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv')
 .options(header='true', inferschema='true')
```
```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('bank').getOrCreate()
df = spark.read(
    'file:/mnt/disk2/bank.csv',
    header=True)
df = df.drop('day', 'month', 'poutcome').show(5)
+---+------------+-------+---------+-------+-------+-------+----+-------+-----+
|   |   |       |       |       |       |       |    |       |    |
+---+------------+-------+---------+-------+-------+-------+----+-------+-----+
| 58|   | management| married| tertiary| no| 2143| yes| no|unknown|
| 261|   | 1| -1| 0| no| |
| 44|   | technician| single| secondary| no| 29| yes| no|unknown|
| 151|   | 1| -1| 0| no| |
| 33|   | entrepreneur| married| secondary| no| 2| yes| yes|unknown|
| 76|   | 1| -1| 0| no| |
| 47|   | blue-collar| married| unknown| no| 1506| yes| no|unknown|
| 92|   | 1| -1| 0| no| |
| 33|   | unknown| single| unknown| no| 1| no| no|unknown|
| 198|   | 1| -1| 0| no| |
+---+------------+-------+---------+-------+-------+-------+----+-------+-----+
```

```
only showing top 5 rows
```

```
df.printSchema()
```

<table>
<thead>
<tr>
<th>root</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

### Note:

You are strongly encouraged to try my get_dummy function for dealing with the categorical data in complex dataset.
Supervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
        for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
        outputCol="{0}_encoded".format(indexer.getOutputCol()))
        for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
```

Unsupervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols):
    ""
    Get dummy variables and concat with continuous variables for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    ""
```

(continues on next page)
indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
            for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
            getOutputCol(),
            outputCol="{0}_encoded".format(indexer.
            getOutputCol()))
            for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
            getOutputCol() for encoder in encoders] + continuousCols, outputCol=
            "features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')

Two in one:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
              dropLast=False):
    '''
    Get dummy variables and concat with continuous variables for ml
    modeling.

    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix

    :author: Wenqiang Feng
    :email: von198@gmail.com

    >>> df = spark.createDataFrame(
            [(0, "a"),
             (1, "b"),
             (2, "c"),
             (3, "a"),
             (4, "a"),
             (5, "c")], ["id", "category"])

    >>> indexCol = 'id'

    (continues on next page)```

(continued from previous page)
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol)
>>> mat.show()

>>> +---+-------------+
<table>
<thead>
<tr>
<th>id</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>1</td>
<td>[0.0, 0.0, 1.0]</td>
</tr>
<tr>
<td>2</td>
<td>[0.0, 1.0, 0.0]</td>
</tr>
<tr>
<td>3</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>4</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>5</td>
<td>[0.0, 1.0, 0.0]</td>
</tr>
</tbody>
</table>
+---+-------------+

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
for c in categoricalCols]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
outputCol="{0}_encoded".format(indexer.getOutputCol()),
dropLast=dropLast)
for indexer in indexers]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label', col(labelCol))
return data.select(indexCol, 'features', 'label')
eelif not indexCol and labelCol:
    # for supervised learning
```
```python
data = data.withColumn('label', col(labelCol))
return data.select('features', 'label')

elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol, 'features')

eelif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')
```

```python
def get_dummy(df, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,
    → VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
        for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
        outputCol="{0}_encoded".format(indexer.getOutputCol()))
        for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label', col(labelCol))

    return data.select('features', 'label')
```

3. Deal with categorical data and Convert the data to dense vector

```python
catcols = ['job', 'marital', 'education', 'default',
            'housing', 'loan', 'contact', 'poutcome']

num_cols = ['balance', 'duration', 'campaign', 'pdays', 'previous',]
labelCol = 'y'

data = get_dummy(df, catcols, num_cols, labelCol)
data.show(5)
```

```
+--------------------+-----+
| features | label |
+--------------------+-----+
```
4. Deal with Categorical Label and Variables

```python
from pyspark.ml.feature import StringIndexer

# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
                              outputCol='indexedLabel').fit(data)

labelIndexer.transform(data).show(5, True)
```

```text
+--------------------+-----+------------+
| features|label|indexedLabel|
+--------------------+-----+------------+
| (29, [1, 11, 14, 16, 1]...) no | 0.0|
| (29, [2, 12, 13, 16, 1]...) no | 0.0|
| (29, [7, 11, 13, 16, 1]...) no | 0.0|
| (29, [0, 11, 16, 17, 1]...) no | 0.0|
| (29, [12, 16, 18, 20, 0]...) no | 0.0|
+--------------------+-----+------------+
```

```python
from pyspark.ml.feature import VectorIndexer

# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as
# continuous.
featureIndexer =VectorIndexer(inputCol="features", 
                              outputCol="indexedFeatures", 
                              maxCategories=4).fit(data)

featureIndexer.transform(data).show(5, True)
```

```text
+--------------------+-----+--------------------+
| features|label| indexedFeatures|
+--------------------+-----+--------------------+
| (29, [1, 11, 14, 16, 1]...) no | (29, [1, 11, 14, 16, 1]...) |
| (29, [2, 12, 13, 16, 1]...) no | (29, [2, 12, 13, 16, 1]...) |
| (29, [7, 11, 13, 16, 1]...) no | (29, [7, 11, 13, 16, 1]...) |
| (29, [0, 11, 16, 17, 1]...) no | (29, [0, 11, 16, 17, 1]...) |
| (29, [12, 16, 18, 20, 0]...) no | (29, [12, 16, 18, 20, 0]...) |
+--------------------+-----+--------------------+
```

5. Split the data to training and test data sets

11.1. Binomial logistic regression
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])

trainingData.show(5, False)
testData.show(5, False)

6. Fit Logistic Regression Model

```python
from pyspark.ml.classification import LogisticRegression
logr = LogisticRegression(featuresCol='indexedFeatures', labelCol='indexedLabel')
```

7. Pipeline Architecture
# Convert indexed labels back to original labels.
labeledConverter = IndexToString(inputCol="prediction",
                                 outputCol="predictedLabel",
                                 labels=labelIndexer.labels)

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, logr,
                            labelConverter])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)

8. Make predictions

# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features", "label", "predictedLabel").show(5)

+--------------------+-----+--------------+
| features | label | predictedLabel |
+--------------------+-----+--------------+
| (29, [0, 11, 13, 16, 1... | no | no |
| (29, [0, 11, 13, 16, 1... | no | no |
| (29, [0, 11, 13, 16, 1... | no | no |
| (29, [0, 11, 13, 16, 1... | no | no |
| (29, [0, 11, 13, 16, 1... | no | no |
+--------------------+-----+--------------+
only showing top 5 rows

9. Evaluation

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g % (1.0 - accuracy))

Test Error = 0.0987688

lrModel = model.stages[2]
trainingSummary = lrModel.summary

# Obtain the objective per iteration
# objectiveHistory = trainingSummary.objectiveHistory
# print("objectiveHistory:")
# for objective in objectiveHistory:

(continues on next page)
# print(objective)

# Obtain the receiver-operating characteristic as a dataframe and areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupby().max('F-Measure').select('max(F-Measure)').head(5)
bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-Measure)']) \
    .select('threshold').head()['threshold']
lr.setThreshold(bestThreshold)

You can use z.show() to get the data and plot the ROC curves:

You can also register a TempTable data.registerTempTable('roc_data') and then use sql to plot the ROC curve:

10. visualization
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    
    ""
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    ""

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label

class_temp = predictions.select("label").groupBy("label")
            .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)
# # # print(class_name)
class_names

['no', 'yes']

from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()
y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred, labels=class_names)
cnf_matrix

array([[15657, 379],
        [1410, 667]])

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
        title='Confusion matrix, without normalization')
plt.show()

Confusion matrix, without normalization
[[15657  379]
 [1410  667]]

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
        title='Normalized confusion matrix')
plt.show()
11.2 Multinomial logistic regression

11.2.1 Introduction

11.2.2 Demo

- The Jupyter notebook can be download from Logistic Regression.
- For more details, please visit Logistic Regression API.

Note: In this demo, I introduced a new function `get_dummy` to deal with the categorical data. I highly recommend you to use my `get_dummy` function in the other cases. This function will save a lot of time for you.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
```
Learning Apache Spark with Python

```python
spark = SparkSession \
    .builder \
    .appName("Python Spark MultinomialLogisticRegression classification") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv') \
    .options(header='true', inferschema='true') \
    .load("./data/WineData2.csv",header=True);
```

df.show(5)

```sql
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
|fixed|volatile|citric|sugar|chlorides|free|total|density|pH |sulphates|alcohol|quality|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
| 7.4 | 0.7    | 0.0  | 1.9 | 0.076   |11.0| 34.0| 0.9978 |3.51| 0.56    |
| 7.8 | 0.88   | 0.0  | 2.6 | 0.098   |25.0| 67.0| 0.9968 |3.2 | 0.68    |
| 7.8 | 0.76   | 0.04 | 2.3 | 0.092   |15.0| 54.0| 0.997 |3.26| 0.65    |
| 11.2| 0.28   | 0.56 | 1.9 | 0.075   |17.0| 60.0| 0.998 |3.16| 0.58    |
| 7.4 | 0.7    | 0.0  | 1.9 | 0.076   |11.0| 34.0| 0.9978|3.51| 0.56    |
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
```

only showing top 5 rows

df.printSchema()

```sql
root
|-- fixed: double (nullable = true)
|-- volatile: double (nullable = true)
|-- citric: double (nullable = true)
|-- sugar: double (nullable = true)
|-- chlorides: double (nullable = true)
|-- free: double (nullable = true)
|-- total: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: string (nullable = true)
```
# Convert to float format

def string_to_float(x):
    return float(x)

#
def condition(r):
    if (0 <= r <= 4):
        label = "low"
    elif (4 < r <= 6):
        label = "medium"
    else:
        label = "high"
    return label

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType

string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())

df = df.withColumn("quality", quality_udf("quality"))

df.show(5,True)

df.printSchema()

root
|-- fixed: double (nullable = true)
|-- volatile: double (nullable = true)
|-- citric: double (nullable = true)
|-- sugar: double (nullable = true)
|-- chlorides: double (nullable = true)

(continues on next page)
3. Deal with categorical data and Convert the data to dense vector

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

Supervised learning version:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, 
    OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [ StringIndexer(inputCol=c, outputCol="{0}_
    indexed".format(c))
        for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
    getOutputCol(), outputCol="{0}_encoded".format(indexer.
    getOutputCol()))
        for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
    getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label',col(labelCol))

    return data.select(indexCol,'features','label')
```

Unsupervised learning version:
```python
def get_dummy(df, indexCol, categoricalCols, continuousCols):
    '''
    Get dummy variables and concat with continuous variables
    for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    '''

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                                outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol, 'features')
```

Two in one:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol, dropLast=False):
    '''
    Get dummy variables and concat with continuous variables for ml
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    '''

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                                outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol, 'features')
```

(continues on next page)
```python
>>> df = spark.createDataFrame([
    (0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c")
], ["id", "category"])

>>> indexCol = 'id'
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
   → labelCol)
>>> mat.show()

>>> +---+-------------+
>>> | id| features    |
>>> +---+-------------+
>>> | 0|[1.0,0.0,0.0]|
>>> | 1|[0.0,0.0,1.0]|
>>> | 2|[0.0,1.0,0.0]|
>>> | 3|[1.0,0.0,0.0]|
>>> | 4|[1.0,0.0,0.0]|
>>> | 5|[0.0,1.0,0.0]|
>>> +---+-------------+

from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
   VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
   for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
   outputCol="{0}_encoded".format(indexer.getOutputCol()),dropLast=dropLast)
   for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol()]
   for encoder in encoders]
```

(continues on next page)
def get_dummy(df, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,
    VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                               outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                                 + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label', col(labelCol))
    return data.select('features', 'label')
4. Transform the dataset to DataFrame

```python
from pyspark.ml.linalg import Vectors  # !!!!caution: not from pyspark.mllib.linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]), r[-1]]).toDF(['features', 'label'])

transformed = transData(df)
transformed.show(5)
```

4. Deal with Categorical Label and Variables

```python
# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label', outputCol='indexedLabel').fit(transformed)
labelIndexer.transform(transformed).show(5, True)
```

```text
+-------------------+----------+-------------------+---+
<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedLabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4,0.7,0.0,1.9,...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8,0.88,0.0,2.6...</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8,0.76,0.04,2....</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[11.2,0.28,0.56,1...</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.4,0.7,0.0,1.9,...</td>
<td>medium</td>
<td>0.0</td>
</tr>
</tbody>
</table>
|-------------------+----------+-------------------+---+
only showing top 5 rows
```

```python
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(transformed)
featureIndexer.transform(transformed).show(5, True)
```

```text
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
```

only showing top 5 rows

186 Chapter 11. Classification
5. Split the data to training and test data sets

```python
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])
```

```python
trainingData.show(5,False)
testData.show(5,False)
```

6. Fit Multinomial logisticRegression Classification Model

```python
from pyspark.ml.classification import LogisticRegression

logr = LogisticRegression(featuresCol='indexedFeatures', labelCol='indexedLabel')
```

7. Pipeline Architecture

```python
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel")
```

---

11.2. Multinomial logistic regression
labels=labelIndexer.labels)

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, logr, labelConverter])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)

8. Make predictions

# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)

+--------------------+------+--------------+
| features| label|predictedLabel|
+--------------------+------+--------------+
| [4.6,0.52,0.15,2....| low| medium|
| [4.9,0.42,0.0,2.1...| high| high|
| [5.0,0.42,0.24,2....| high| high|
| [5.0,1.02,0.04,1....| low| medium|
| [5.0,1.04,0.24,1....| medium| medium|
+--------------------+------+--------------+
only showing top 5 rows

9. Evaluation

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
        labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))

Test Error = 0.181287

lrModel = model.stages[2]
trainingSummary = lrModel.summary

# Obtain the objective per iteration
# objectiveHistory = trainingSummary.objectiveHistory
# print("objectiveHistory:")
# for objective in objectiveHistory:
#     print(objective)
# Obtain the receiver-operating characteristic as a dataframe and → areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').head(5)
bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-Measure)'])
# lr.setThreshold(bestThreshold)

You can use z.show() to get the data and plot the ROC curves:

You can also register a TempTable data.registerTempTable('roc_data') and then use sql to plot the ROC curve:

10. visualization

11.2. Multinomial logistic regression
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                      normalize=False,
                      title='Confusion matrix',
                      cmap=plt.cm.Blues):

    ""
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    ""
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                  horizontalalignment="center",
                  color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

class_temp = predictions.select("label").groupBy("label")
    .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)

# # # print(class_name)
class_names

['medium', 'high', 'low']

from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()
y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred, labels=class_names)
cnf_matrix

array([[526, 11, 2],
        [73, 33, 0],
        [38, 0, 1]])

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()

Confusion matrix, without normalization
[[526 11 2]
 [73 33 0]
 [38 0 1]]

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
title='Confusion matrix, with normalization')
plt.show()
11.3 Decision tree Classification

11.3.1 Introduction

11.3.2 Demo

- The Jupyter notebook can be download from Decision Tree Classification.
- For more details, please visit DecisionTreeClassifier API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
```
2. Load dataset

```
spark = SparkSession 
   .builder 
   .appName("Python Spark Decision Tree classification") 
   .config("spark.some.config.option", "some-value") 
   .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').
   options(header='true', 
   inferschema='true') 
   .load("../data/WineData2.csv",header=True);

df.show(5,True)
```

# Convert to float format

```
def string_to_float(x):
    return float(x)
```

```
def condition(r):
    if (0<= r <= 4):
        label = "low"
    elif(4< r <= 6):
        label = "medium"
    else:
        label = "high"
    return label
```

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType
```

11.3. Decision tree Classification
string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())

df = df.withColumn("quality", quality_udf("quality"))
df.show(5, True)
df.printSchema()

+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
|fixed|volatile|citric|sugar|chlorides|free|total|density|pH|sulphates|alcohol|quality|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
| 7.4 | 0.7 | 0.0 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | medium |
| 7.8 | 0.88 | 0.0 | 2.6 | 0.098 | 25.0 | 67.0 | 0.9968 | 3.2 | 0.68 | medium |
| 9.4 | medium |
| 7.8 | 0.76 | 0.04 | 2.3 | 0.092 | 15.0 | 54.0 | 0.997 | 3.26 | 0.65 | medium |
| 11.2 | 0.28 | 0.56 | 1.9 | 0.075 | 17.0 | 60.0 | 0.998 | 3.16 | 0.58 | medium |
| 9.8 | medium |
| 7.4 | 0.7 | 0.0 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | medium |
| 9.4 | medium |

---

3. Convert the data to dense vector

Note:

You are strongly encouraged to try my get_dummy function for dealing with the categorical data in complex dataset.

Supervised learning version:
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):

    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
        OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{}_indexed".format(c))
        for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
        outputCol="{}_encoded".format(indexer.getOutputCol()))
        for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]+continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)
    data = data.withColumn('label', col(labelCol))

    return data.select(indexCol, 'features', 'label')

Unsupervised learning version:

def get_dummy(df, indexCol, categoricalCols, continuousCols):
    '''
    Get dummy variables and concat with continuous variables for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    '''

    indexers = [StringIndexer(inputCol=c, outputCol="{}_indexed".format(c))
(continues on next page)
for c in categoricalCols ]

    # default setting: dropLast=True
    encoders = [ OneHotEncoder(inputCol=indexer.
    ~getOutputCol(),
    outputCol="{0}_encoded".format(indexer.
    ~getOutputCol()))]

    for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
    ~getOutputCol() for encoder in encoders]
    + continuousCols, outputCol=
    "features")

    pipeline = Pipeline(stages=indexers + encoders +
    ~[assembler])

    model=pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')

Two in one:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
    ~dropLast=False):

    '''
    Get dummy variables and concat with continuous variables for ml
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com

    >>> df = spark.createDataFrame(
    [(0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c")
    ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
```
(continues on next page)
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,
                  labelCol)
>>> mat.show()

```
+---+-------------+
| id| features |
+---+-------------+
| 0| [1.0,0.0,0.0] |
| 1| [0.0,0.0,1.0] |
| 2| [0.0,1.0,0.0] |
| 3| [1.0,0.0,0.0] |
| 4| [1.0,0.0,0.0] |
| 5| [0.0,1.0,0.0] |
+---+-------------+
```

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
   VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
            for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                           outputCol="{0}_encoded".format(indexer.getOutputCol()),dropLast=dropLast)
            for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select(indexCol,'features','label')

elif not indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label',col(labelCol))
return data.select('features','label')
```
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol,'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')

# !!!!caution: not from pyspark.mllib.linalg import Vectors
from pyspark.ml.linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString,StringIndexer, VectorIndexer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]),r[-1]]).toDF([˓→'features','label'])

4. Transform the dataset to DataFrame

transformed = transData(df)
transformed.show(5)

+-------------------+------+
| features| label|
+-------------------+------+
|[7.4,0.7,0.0,1.9,...|medium|
|[7.8,0.88,0.0,2.6...|medium|
|[7.8,0.76,0.04,2....|medium|
|[11.2,0.28,0.56,1...|medium|
|[7.4,0.7,0.0,1.9,...|medium|

only showing top 5 rows

5. Deal with Categorical Label and Variables

# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
    outputCol='indexedLabel').fit(transformed)
labelIndexer.transform(transformed).show(5, True)

+-------------------+-------+--------------+
| features| label|indexedLabel|
+-------------------+-------+--------------+
|[7.4,0.7,0.0,1.9,...|medium| 0.0|
|[7.8,0.88,0.0,2.6...|medium| 0.0|
|[7.8,0.76,0.04,2....|medium| 0.0|
|[11.2,0.28,0.56,1...|medium| 0.0|
|[7.4,0.7,0.0,1.9,...|medium| 0.0|

(continues on next page)
only showing top 5 rows

```python
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as
# continuous.
featureIndexer = VectorIndexer(inputCol="features", 
outputCol="indexedFeatures", 
maxCategories=4).fit(transformed)
featureIndexer.transform(transformed).show(5, True)
```

6. Split the data to training and test data sets

```python
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])

trainingData.show(5)
testData.show(5)
```

11.3. Decision tree Classification
7. Fit Decision Tree Classification Model

```python
from pyspark.ml.classification import DecisionTreeClassifier

# Train a DecisionTree model
dTree = DecisionTreeClassifier(labelCol='indexedLabel', featuresCol='indexedFeatures')
```

8. Pipeline Architecture

```python
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                                labels=labelIndexer.labels)

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dTree, labelConverter])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)
```

9. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+-------------------+------+--------------+
| features| label|predictedLabel|
|-------------------+------+--------------|
|[4.9,0.42,0.0,2.1...| high| high|
|[5.0,0.38,0.01,1....|medium| medium|
|[5.0,0.4,0.5,4.3,...|medium| medium|
|[5.0,0.42,0.24,2....| high| medium|
|[5.0,0.74,0.0,1.2...|medium| medium|
+-------------------+------+--------------+
only showing top 5 rows
```

10. Evaluation

```python
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = \$g\ % (1.0 - accuracy)"
```

(continues on next page)
rfModel = model.stages[-2]
print(rfModel)  # summary only

Test Error = 0.45509
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_
→4545ac8ca9c8438ef2a)
of depth 5 with 59 nodes

11. visualization

```python
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
    normalize=False,
    title='Confusion matrix',
    cmap=plt.cm.Blues):
    
    '''
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    '''

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
            horizontalalignment="center",
            color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

class_temp = predictions.select("label").groupBy("label")
    .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)
# # # print(class_name)
class_names
['medium', 'high', 'low']

from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()

y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred, labels=class_names)
cnf_matrix

array([[497, 29, 7],
       [40, 42, 0],
       [22, 0, 2]])

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()

Confusion matrix, without normalization
[[497 29 7]
 [40 42 0]
 [22 0 2]]

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')
plt.show()

Normalized confusion matrix
[[ 0.93245779 0.05440901 0.01313321]
 [ 0.48780488 0.51219512 0.]
 [ 0.91666667 0. 0.08333333]]
11.3. Decision tree Classification
### 11.4 Random forest Classification

#### 11.4.1 Introduction

#### 11.4.2 Demo

- The Jupyter notebook can be downloaded from Random forest Classification.
- For more details, please visit RandomForestClassifier API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
spark = SparkSession 
.builder 
.appName("Python Spark Decision Tree classification") 
.config("spark.some.config.option", "some-value") 
.getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').
    options(header='true', 
            inferSchema='true') 
    .load("../data/WineData2.csv",header=True);
df.show(5,True)
```

<table>
<thead>
<tr>
<th>fixed</th>
<th>volatile</th>
<th>citric</th>
<th>sugar</th>
<th>chlorides</th>
<th>free</th>
<th>total</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.9978</td>
</tr>
<tr>
<td>9.4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.88</td>
<td>0.0</td>
<td>2.6</td>
<td>0.098</td>
<td>25.0</td>
<td>67.0</td>
<td>0.9968</td>
</tr>
<tr>
<td>9.8</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.76</td>
<td>0.04</td>
<td>2.3</td>
<td>0.092</td>
<td>15.0</td>
<td>54.0</td>
<td>0.997</td>
</tr>
<tr>
<td>9.8</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2</td>
<td>0.28</td>
<td>0.56</td>
<td>1.9</td>
<td>0.075</td>
<td>17.0</td>
<td>60.0</td>
<td>0.998</td>
</tr>
<tr>
<td>9.8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.9978</td>
</tr>
<tr>
<td>9.4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

only showing top 5 rows

# Convert to float format

```python
def string_to_float(x):
    return float(x)
```

(continues on next page)
```python
# def condition(r):
    if (0 <= r <= 4):
        label = "low"
    elif (4 < r <= 6):
        label = "medium"
    else:
        label = "high"
    return label

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType

string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())

df = df.withColumn("quality", quality_udf("quality"))
df.show(5,True)
df.printSchema()
```

```
<table>
<thead>
<tr>
<th>fixed</th>
<th>volatile</th>
<th>citric</th>
<th>sugar</th>
<th>chlorides</th>
<th>free</th>
<th>total</th>
<th>density</th>
<th>pH</th>
<th>sulphates</th>
<th>alcohol</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.9978</td>
<td>3.51</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.88</td>
<td>0.0</td>
<td>2.6</td>
<td>0.098</td>
<td>25.0</td>
<td>67.0</td>
<td>0.9968</td>
<td>3.2</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.76</td>
<td>0.04</td>
<td>2.3</td>
<td>0.092</td>
<td>15.0</td>
<td>54.0</td>
<td>0.997</td>
<td>3.26</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2</td>
<td>0.28</td>
<td>0.56</td>
<td>1.9</td>
<td>0.075</td>
<td>17.0</td>
<td>60.0</td>
<td>0.998</td>
<td>3.16</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.997</td>
<td>3.51</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
only showing top 5 rows
```

(root
 |-- fixed: double (nullable = true)
 |-- volatile: double (nullable = true)
 |-- citric: double (nullable = true)
 |-- sugar: double (nullable = true)
 |-- chlorides: double (nullable = true)
 |-- free: double (nullable = true)
 |-- total: double (nullable = true)
 |-- density: double (nullable = true)
 |-- pH: double (nullable = true)

(continues on next page)
3. Convert the data to dense vector

Note:

You are strongly encouraged to try my get_dummy function for dealing with the categorical data in complex dataset.

Supervised learning version:

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer,
    OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
            for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                          outputCol="{0}_encoded".format(indexer.getOutputCol()))
            for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)

data = data.withColumn('label',col(labelCol))

return data.select(indexCol,'features','label')
```

Unsupervised learning version:

```python
def get_dummy(df,indexCol,categoricalCols,continuousCols):
    '''
(continues on next page)
```
Get dummy variables and concat with continuous variables for unsupervised learning.

:param df: the dataframe
:param categoricalCols: the name list of the categorical data
:param continuousCols: the name list of the numerical data

:return k: feature matrix

@author: Wenqiang Feng
@email: von198@gmail.com

```python
indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
            for c in categoricalCols]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                           outputCol="{0}_encoded".format(indexer.getOutputCol()))
            for indexer in indexers]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

return data.select(indexCol,'features')
```

Two in one:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol, dropLast=False):
    
    Get dummy variables and concat with continuous variables for ml modeling.
    
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
```

(continues on next page)
```python
>>> df = spark.createDataFrame(
    [(0, "a"),
     (1, "b"),
     (2, "c"),
     (3, "a"),
     (4, "a"),
     (5, "c")
    ], ["id", "category"])
```

```python
>>> indexCol = 'id'
>>> categoricalCols = ['category']
>>> continuousCols = []
>>> labelCol = []

>>> mat = get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol)
>>> mat.show()
```

```python
<table>
<thead>
<tr>
<th>id</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>1</td>
<td>[0.0, 0.0, 1.0]</td>
</tr>
<tr>
<td>2</td>
<td>[0.0, 1.0, 0.0]</td>
</tr>
<tr>
<td>3</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>4</td>
<td>[1.0, 0.0, 0.0]</td>
</tr>
<tr>
<td>5</td>
<td>[0.0, 1.0, 0.0]</td>
</tr>
</tbody>
</table>
```

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
   VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
   for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
   outputCol="{0}_encoded".format(indexer.getOutputCol()),
   dropLast=dropLast)
   for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() + continuousCols, outputCol="features"]
```

(continues on next page)
pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
elif not indexCol and labelCol:
    # for supervised learning
    data = data.withColumn('label', col(labelCol))
    return data.select('features', 'label')
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol, 'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')

# !!!! caution: not from pyspark.mllib.linalg. import Vectors
from pyspark.ml.linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]), r[-1]]).toDF(['features', 'label'])

4. Transform the dataset to DataFrame

transformed = transData(df)
transformed.show(5)

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[7.8, 0.88, 0.0, 2.6 ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[7.8, 0.76, 0.04, 2 ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[11.2, 0.28, 0.56, 1 ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
</tr>
</tbody>
</table>

only showing top 5 rows

5. Deal with Categorical Label and Variables

11.4. Random forest Classification
# Index labels, adding metadata to the label column

```python
labelIndexer = StringIndexer(inputCol='label',
                            outputCol='indexedLabel').fit(transformed)
labelIndexer.transform(transformed).show(5, True)
```

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedLabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8, 0.88, 0.0, 2.6, ...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8, 0.76, 0.04, 2.6, ...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[11.2, 0.28, 0.56, 1.9, ...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
<td>0.0</td>
</tr>
</tbody>
</table>

only showing top 5 rows

# Automatically identify categorical features, and index them.

# Set maxCategories so features with > 4 distinct values are treated as continuous.

```python
featureIndexer = VectorIndexer(inputCol="features",
                                outputCol="indexedFeatures",
                                maxCategories=4).fit(transformed)
featureIndexer.transform(transformed).show(5, True)
```

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedFeatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
</tr>
<tr>
<td>[7.8, 0.88, 0.0, 2.6, ...]</td>
<td>medium</td>
<td>[7.8, 0.88, 0.0, 2.6, ...]</td>
</tr>
<tr>
<td>[7.8, 0.76, 0.04, 2.6, ...]</td>
<td>medium</td>
<td>[7.8, 0.76, 0.04, 2.6, ...]</td>
</tr>
<tr>
<td>[11.2, 0.28, 0.56, 1.9, ...]</td>
<td>medium</td>
<td>[11.2, 0.28, 0.56, 1.9, ...]</td>
</tr>
<tr>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
<td>medium</td>
<td>[7.4, 0.7, 0.0, 1.9, ...]</td>
</tr>
</tbody>
</table>

only showing top 5 rows

6. Split the data to training and test data sets

```python
(trainingData, testData) = transformed.randomSplit([0.6, 0.4])
```

trainingData.show(5)  
testData.show(5)

<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4.6, 0.52, 0.15, 2, ...]</td>
<td>low</td>
</tr>
<tr>
<td>[4.7, 0.6, 0.17, 2.3, ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[5.0, 1.02, 0.04, 1, ...]</td>
<td>low</td>
</tr>
<tr>
<td>[5.0, 1.04, 0.24, 1, ...]</td>
<td>medium</td>
</tr>
<tr>
<td>[5.1, 0.585, 0.0, 1, ...]</td>
<td>high</td>
</tr>
</tbody>
</table>

(continues on next page)
7. Fit Random Forest Classification Model

```python
from pyspark.ml.classification import RandomForestClassifier

# Train a RandomForest model.
rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)
```

8. Pipeline Architecture

```python
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)
```

9. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+--------------------+------|--------------+
| features| label|predictedLabel|
+--------------------+------|--------------+
| [4.9,0.42,0.0,2.1]...| high| high|
| [5.0,0.38,0.01,1....| medium| medium|
| [5.0,0.4,0.5,4.3,...| medium| medium|
| [5.0,0.42,0.24,2....| high| medium|
| [5.0,0.74,0.0,1.2...| medium| medium|
```

(continues on next page)
10. Evaluation

```python
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(  
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g\n" % (1.0 - accuracy))

rfModel = model.stages[-2]
print(rfModel) # summary only
```

Test Error = 0.173502
RandomForestClassificationModel (uid=rfc_a3395531f1d2) with 10 trees

11. visualization

```python
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,  
normalize=False,  
title='Confusion matrix',  
cmap=plt.cm.Blues):
    
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
```

(continues on next page)
```python
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')

class_temp = predictions.select("label").groupBy("label")
               .count().sort('count', ascending=False).toPandas()
class_names = map(str, class_temp)

# print(class_name)
class_names

['medium', 'high', 'low']

from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()

y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred, labels=class_names)
cnf_matrix

array([[502, 9, 0],
       [73, 22, 0],
       [28, 0, 0]])

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()

Confusion matrix, without normalization
[[502 9 0]
 [73 22 0]
 [28 0 0]]

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')
plt.show()
```

11.4. Random forest Classification
Normalized confusion matrix

\[
\begin{bmatrix}
0.98238748 & 0.01761252 & 0. \\
0.76842105 & 0.23157895 & 0. \\
1. & 0. & 0. \\
\end{bmatrix}
\]

11.5 Gradient-boosted tree Classification

11.5.1 Introduction

11.5.2 Demo

- The Jupyter notebook can be download from [Gradient boosted tree Classification](#).
- For more details, please visit [GBTClassifier API](#).

**Warning:** Unfortunately, the GBTClassifier currently only supports binary labels.
11.6 XGBoost: Gradient-boosted tree Classification

11.6.1 Introduction

11.6.2 Demo

- The Jupyter notebook can be download from Gradient boosted tree Classification.
- For more details, please visit GBTClassifier API.

Warning: Unfortunately, I didn’t find a good way to setup the XGBoost directly in Spark. But I do get the XGBoost work with pysparkling on my machine.

1. Start H2O cluster inside the Spark environment

    ```python
    from pysparkling import *
    hc = H2OContext.getOrCreate(spark)
    ```

    Connecting to H2O server at http://192.168.0.102:54323... successful.
    H2O cluster uptime: 07 secs
    H2O cluster timezone: America/Chicago
    H2O data parsing timezone: UTC
    H2O cluster version: 3.22.1.3

    (continues on next page)
Learning Apache Spark with Python

H2O cluster version age: 20 days
H2O cluster name: sparkling-water-dt216661_local-1550259209801
H2O cluster total nodes: 1
H2O cluster free memory: 848 Mb
H2O cluster total cores: 8
H2O cluster allowed cores: 8
H2O cluster status: accepting new members, healthy
H2O connection url: http://192.168.0.102:54323
H2O connection proxy: None
H2O internal security: False
H2O API Extensions: XGBoost, Algos, AutoML, Core V3, Core V4
Python version: 3.7.1 final

Sparkling Water Context:
* H2O name: sparkling-water-dt216661_local-1550259209801
* cluster size: 1
* list of used nodes:
  (executorId, host, port)
  ------------------------
  (driver,192.168.0.102,54323)
  ------------------------

Open H2O Flow in browser: http://192.168.0.102:54323 (CMD + click in MacOS)

2. Parse the data using H2O and convert them to Spark Frame

```python
import h2o
frame = h2o.import_file("https://raw.githubusercontent.com/h2oai/sparkling-water/master/examples/smalldata/prostate/prostate.csv")
spark_frame = hc.as_spark_frame(frame)

Parse progress: [==========================================] 100%

spark_frame.show(4)
```

```
+---+-------+---+----+-----+-----+----+----+-------+
| ID|CAPSULE|AGE|RACE|DPROS|DCAPS| PSA| VOL|GLEASON|
+---+-------+---+----+-----+-----+----+----+-------+
| 1| 0| 65| 1| 2| 1| 1.4| 0.0| 6|
| 2| 0| 72| 1| 3| 2| 6.7| 0.0| 7|
| 3| 0| 70| 1| 1| 2| 4.9| 0.0| 6|
| 4| 0| 76| 2| 2| 1|51.2|20.0| 7|
```

only showing top 4 rows

3. Train the model

```python
from pysparkling.ml import H2OXGBoost
estimator = H2OXGBoost(predictionCol="AGE")
model = estimator.fit(spark_frame)
```
4. Run Predictions

```python
predictions = model.transform(spark_frame)
predictions.show(4)
```

```
+---+-------+---+----+-----+-----+----+----+-------+-------------------+
| ID|CAPSULE|AGE|RACE|DPROS|DCAPS| PSA| VOL|GLEASON| prediction_output|
+---+-------+---+----+-----+-----+----+----+-------+-------------------+
| 1| 0|65| 1| 2| 1|1.4|0.0|6|64.85852813720703|
| 2| 0|72| 1| 3| 2|6.7|0.0|7|72.0611801147461|
| 3| 0|70| 1| 1| 2|4.9|0.0|6|70.26496887207031|
| 4| 0|76| 2| 2| 1|51.2|20.0|7|75.26521301269531|
+---+-------+---+----+-----+-----+----+----+-------+-------------------+
only showing top 4 rows
```

11.7 Naive Bayes Classification

11.7.1 Introduction

11.7.2 Demo

- The Jupyter notebook can be download from Naive Bayes Classification.
- For more details, please visit NaiveBayes API.

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession \\
.builder \\
   .appName("Python Spark Naive Bayes classification") \\
   .config("spark.some.config.option", "some-value") \\
   .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv') \\
   .options(header='true', inferschema='true') \\
   .load("./data/WineData2.csv",header=True);

df.show(5)
```

```
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
|fixed|volatile|citric|sugar|chlorides|free|total|density|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
| 7.4| 0.7| 0.0| 1.9|0.076|11.0|34.0|0.9978|3.51|0.56| 9.4|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
```

(continues on next page)
```
(df.printSchema()

root
|-- fixed: double (nullable = true)
|-- volatile: double (nullable = true)
|-- citric: double (nullable = true)
|-- sugar: double (nullable = true)
|-- chlorides: double (nullable = true)
|-- free: double (nullable = true)
|-- total: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: string (nullable = true)

# Convert to float format

def string_to_float(x):
    return float(x)

# def condition(r):
#     if (0 <= r <= 6):
#         label = "low"
#     else:
#         label = "high"
# return label

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType
string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())

df = df.withColumn("quality", quality_udf("quality"))

df.show(5,True)
```
<table>
<thead>
<tr>
<th></th>
<th>fixed</th>
<th>volatile</th>
<th>citric</th>
<th>sugar</th>
<th>chlorides</th>
<th>free</th>
<th>total</th>
<th>density</th>
<th>pH</th>
<th>sulphates</th>
<th>alcohol</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.9978</td>
<td>3.51</td>
<td>0.56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.88</td>
<td>0.0</td>
<td>2.6</td>
<td>0.098</td>
<td>25.0</td>
<td>67.0</td>
<td>0.9968</td>
<td>3.2</td>
<td>0.68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>0.76</td>
<td>0.0</td>
<td>2.3</td>
<td>0.092</td>
<td>15.0</td>
<td>54.0</td>
<td>0.997</td>
<td>3.26</td>
<td>0.65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2</td>
<td>0.28</td>
<td>0.56</td>
<td>1.9</td>
<td>0.075</td>
<td>17.0</td>
<td>60.0</td>
<td>0.998</td>
<td>3.16</td>
<td>0.58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>0.7</td>
<td>0.0</td>
<td>1.9</td>
<td>0.076</td>
<td>11.0</td>
<td>34.0</td>
<td>0.9978</td>
<td>3.51</td>
<td>0.56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

only showing top 5 rows

df.printSchema()

```
root
|-- fixed: double (nullable = true)
|-- volatile: double (nullable = true)
|-- citric: double (nullable = true)
|-- sugar: double (nullable = true)
|-- chlorides: double (nullable = true)
|-- free: double (nullable = true)
|-- total: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: string (nullable = true)
```

3. Deal with categorical data and Convert the data to dense vector

**Note:**

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

**Supervised learning version:**

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
```
from pyspark.sql.functions import col

indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
    for c in categoricalCols]

# default setting: dropLast=True
encoders = [OneHotEncoder(inputCol=indexer.
    ->getOutputCol(),
    outputCol="{0}_encoded".format(indexer.
    ->getOutputCol()))
    for indexer in indexers]

assembler = VectorAssembler(inputCols=[encoder.
    ->getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)
data = model.transform(df)
data = data.withColumn('label', col(labelCol))

return data.select(indexCol, 'features', 'label')

Unsupervised learning version:

def get_dummy(df, indexCol, categoricalCols, continuousCols):
    '''
    Get dummy variables and concat with continuous variables for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :
    :author: Wenqiang Feng
    :email: von198@gmail.com
    '''

    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
        for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.
        ->getOutputCol()),
        for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.
        ->getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
data = model.transform(df)
data = data.withColumn('label', col(labelCol))

    return data.select(indexCol, 'features', 'label')
outputCol="(0)_encoded".format(indexer.
getOutputCol()))
        for indexer in indexers ]

    assembler = VectorAssembler(inputCols=[encoder.
getOutputCol() for encoder in encoders]
        + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders +
        [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    return data.select(indexCol,'features')

Two in one:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol,
    dropLast=False):
    
    '''
    Get dummy variables and concat with continuous variables for ml
    modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com

    >>> df = spark.createDataFrame([ 
    (0, "a"),
    (1, "b"),
    (2, "c"),
    (3, "a"),
    (4, "a"),
    (5, "c")
    ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
    >>> labelCol = []

    >>> mat = get_dummy(df, indexCol, categoricalCols, continuousCols,
    ...    labelCol)
    >>> mat.show()
    ```

(continues on next page)
```python
>>> +---+-------------+
| id| features    |
+---+-------------+
| 0|[1.0,0.0,0.0] |
| 1|[0.0,0.0,1.0] |
| 2|[0.0,1.0,0.0] |
| 3|[1.0,0.0,0.0] |
| 4|[1.0,0.0,0.0] |
| 5|[0.0,1.0,0.0] |
+---+-------------+

from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.sql.functions import col

indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".
format(c))
            for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
outputCol="{0}_encoded".format(indexer.
getOutputCol()),dropLast=dropLast)
            for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model=pipeline.fit(df)
data = model.transform(df)

if indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label',col(labelCol))
    return data.select(indexCol,'features','label')
elif not indexCol and labelCol:
    # for supervised learning
data = data.withColumn('label',col(labelCol))
    return data.select('features','label')
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol,'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
```
def get_dummy(df, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer, OneHotEncoder,
    → VectorAssembler
    from pyspark.sql.functions import col
    indexers = [StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
               for c in categoricalCols ]
    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                               outputCol="{0}_encoded".format(indexer.getOutputCol()))
                for indexer in indexers ]
    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder, in encoders]
                               + continuousCols, outputCol="features")
    pipeline = Pipeline(stages=indexers + encoders + [assembler])
    model = pipeline.fit(df)
    data = model.transform(df)
    data = data.withColumn('label', col(labelCol))
    return data.select('features','label')

4. Transform the dataset to DataFrame

from pyspark.ml.linalg import Vectors # !!!!caution: not from pyspark.mllib.
    → linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1]), r[-1]]).toDF(['features',
    → 'label'])

transformed = transData(df)
transformed.show(5)
4. Deal with Categorical Label and Variables

```python
# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
                             outputCol='indexedLabel').fit(transformed)
labelIndexer.transform(transformed).show(5, True)
```

```text
<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedLabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4, 0.7, 0.0, 1.9,...]</td>
<td>low</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8, 0.88, 0.0, 2.6...</td>
<td>low</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.8, 0.76, 0.04, 2....</td>
<td>low</td>
<td>0.0</td>
</tr>
<tr>
<td>[11.2, 0.28, 0.56, 1...</td>
<td>low</td>
<td>0.0</td>
</tr>
<tr>
<td>[7.4, 0.7, 0.0, 1.9,...</td>
<td>low</td>
<td>0.0</td>
</tr>
</tbody>
</table>
```

![only showing top 5 rows](image)

```python
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer = VectorIndexer(inputCol="features",
                               outputCol="indexedFeatures",
                               maxCategories=4).fit(transformed)
featureIndexer.transform(transformed).show(5, True)
```

```text
<table>
<thead>
<tr>
<th>features</th>
<th>label</th>
<th>indexedFeatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7.4, 0.7, 0.0, 1.9,...]</td>
<td>low</td>
<td>[7.4, 0.7, 0.0, 1.9,...]</td>
</tr>
<tr>
<td>[7.8, 0.88, 0.0, 2.6...</td>
<td>low</td>
<td>[7.8, 0.88, 0.0, 2.6...</td>
</tr>
<tr>
<td>[7.8, 0.76, 0.04, 2....</td>
<td>low</td>
<td>[7.8, 0.76, 0.04, 2....</td>
</tr>
<tr>
<td>[11.2, 0.28, 0.56, 1...</td>
<td>low</td>
<td>[11.2, 0.28, 0.56, 1...</td>
</tr>
<tr>
<td>[7.4, 0.7, 0.0, 1.9,...</td>
<td>low</td>
<td>[7.4, 0.7, 0.0, 1.9,...</td>
</tr>
</tbody>
</table>
```

![only showing top 5 rows](image)

5. Split the data to training and test data sets

```python
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])
```

(continues on next page)
### 6. Fit Naive Bayes Classification Model

```python
from pyspark.ml.classification import NaiveBayes

nb = NaiveBayes(featuresCol='indexedFeatures', labelCol='indexedLabel')
```

### 7. Pipeline Architecture

```python
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, nb, labelConverter])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)
```

### 8. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```
Learning Apache Spark with Python

9. Evaluation

```python
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy"
)
accuracy = evaluator.evaluate(predictions)
print("Test Error = \%g\) % (1.0 - accuracy))
```

Test Error = 0.307339

```python
lrModel = model.stages[2]
trainingSummary = lrModel.summary

# Obtain the objective per iteration
# objectiveHistory = trainingSummary.objectiveHistory
# print("objectiveHistory:")
# for objective in objectiveHistory:
#     print(objective)

# Obtain the receiver-operating characteristic as a dataframe and areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').head(5)
# bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-Measure)'])
# .select('threshold').head()['threshold']
# lr.setOutput(bestThreshold)
```

You can use `z.show()` to get the data and plot the ROC curves:

You can also register a TempTable `data.registerTempTable('roc_data')` and then use `sql` to plot the ROC curve:

10. visualization
11.7. Naive Bayes Classification
import matplotlib.pyplot as plt
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                        normalize=False,
                        title='Confusion matrix',
                        cmap=plt.cm.Blues):
    
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

class_temp = predictions.select("label").groupBy("label")
               .count().sort('count', ascending=False).toPandas()
class_temp = class_temp["label"].values.tolist()
class_names = map(str, class_temp)
# # # print(class_names)
class_names

['low', 'high']

from sklearn.metrics import confusion_matrix
y_true = predictions.select("label")
y_true = y_true.toPandas()
y_pred = predictions.select("predictedLabel")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred, labels=class_names)
cnf_matrix

array([[392, 169],
       [ 32,  61]])

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')
plt.show()

Confusion matrix, without normalization
[[392 169]
 [ 32  61]]

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')
plt.show()
Normalized confusion matrix

\[
\begin{bmatrix}
0.69875223 & 0.30124777 \\
0.34408602 & 0.65591398
\end{bmatrix}
\]
CHAPER
TWELVE

CLUSTERING

Chinese proverb
Sharpening the knife longer can make it easier to hack the firewood – old Chinese proverb

The above figure was generated by the code from: Python Data Science Handbook.

12.1 K-Means Model

12.1.1 Introduction

k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. The approach k-means follows to solve the problem is called Expectation-Maximization. It can be described as follows:

1. Assign some cluster centers
2. Repeated until converged
   - E-Step: assign points to the nearest center
   - M-step: set the cluster center to the mean

Given a set of observations \((x_1, x_2, \cdots, x_m)\). The objective function is

\[
J = \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} ||x_i - c_k||^2
\]
where \( w_{ik} = 1 \) if \( x_i \) is in cluster \( k \); otherwise \( w_{ik} = 0 \) and \( c_k \) is the centroid of \( x_i \)'s cluster.

Mathematically, k-means is a minimization problem with two parts: First, we minimize \( J \) w.r.t \( w_{ik} \) with \( c_k \) fixed; Then minimize \( J \) w.r.t \( c_k \) with \( w_{ik} \) fixed. i.e.

**E-step:**

\[
\frac{\partial J}{\partial w_{ik}} = \sum_{i=1}^{m} \sum_{k=1}^{K} ||x_i - c_k||^2
\]

\[\Rightarrow w_{ik} = \begin{cases} 1, & \text{if } k = \text{argmin}_j ||x_i - c_j||^2 \\ 0, & \text{otherwise} \end{cases}\]

**M-step:**

\[
\frac{\partial J}{\partial c_k} = 2 \sum_{i=1}^{m} mw_{ik}(x_i - c_k) = 0 \Rightarrow c_k = \frac{\sum_{i=1}^{m} w_{ik}x_i}{\sum_{i=1}^{m} w_{ik}}
\]

### 12.1.2 Demo

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession.(
    .builder
    .appName("Python Spark K-means example")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').(
    options(header='true',
    inferschema='true').
    load("../data/iris.csv",header=True);
)
```

check the data set

```python
df.show(5,True)
df.printSchema()
```

Then you will get

```
+------------+-----------+------------+-----------+-------+
|sepal_length|sepal_width|petal_length|petal_width|species|
+------------+-----------+------------+-----------+-------+
| 5.1| 3.5| 1.4| 0.2| setosa|
| 4.9| 3.0| 1.4| 0.2| setosa|
| 4.7| 3.2| 1.3| 0.2| setosa|
| 4.6| 3.1| 1.5| 0.2| setosa|
```

(continues on next page)
You can also get the Statistical results from the data frame (Unfortunately, it only works for numerical).

```
df.describe().show()
```

Then you will get

```
+---------------------+---------------------+---------------------+---------------------+----------+
| summary             | sepal_length        | sepal_width         | petal_length        |
|---------------------+---------------------+---------------------+---------------------+----------+
| count               | 150                 | 150                 | 150                 | 150      |
| mean                | 5.843333333333335   | 3.0540000000000007  | 3.758666666666693   |
| stddev              | 0.8280661279778637  | 0.43359431136217375 | 1.764420419952262   |
| min                 | 4.3                 | 2.0                 | 1.0                 |
| max                 | 7.9                 | 4.4                 | 6.9                 |
| summary             | petal_width         | species             |
|---------------------+---------------------+---------------------+----------+
| count               | 150                 | setosa              |
| mean                | 1.4                 | null                |
| stddev              | 0.2                 | null                |
| min                 | 0.1                 | setosa              |
| max                 | 2.5                 | virginica           |
```

3. Convert the data to dense vector (features)

```python
# convert the data to dense vector
from pyspark.mllib.linalg import Vectors
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[:-1])]).toDF(['features'])
```

Note:

You are strongly encouraged to try my `get_dummy` function for dealing with the categorical data in complex dataset.

Supervised learning version:
```python
def get_dummy(df, indexCol, categoricalCols, continuousCols, labelCol):
    from pyspark.ml import Pipeline
    from pyspark.ml.feature import StringIndexer,
                            OneHotEncoder, VectorAssembler
    from pyspark.sql.functions import col

    indexers = [StringIndexer(inputCol=c, outputCol="{}_indexed".format(c))
                for c in categoricalCols]

    # default setting: dropLast=True
    encoders = [OneHotEncoder(inputCol=indexer.getOutputCol(),
                              outputCol="{}encoded".format(indexer.getOutputCol()))
                for indexer in indexers]

    assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

    pipeline = Pipeline(stages=indexers + encoders + [assembler])

    model = pipeline.fit(df)
    data = model.transform(df)

    data = data.withColumn('label', col(labelCol))
    return data.select(indexCol, 'features', 'label')
```

Unsupervised learning version:

```python
def get_dummy(df, indexCol, categoricalCols, continuousCols):
    ""
    Get dummy variables and concat with continuous variables for unsupervised learning.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :return k: feature matrix
    :author: Wenqiang Feng
    :email: von198@gmail.com
    ""

    indexers = [StringIndexer(inputCol=c, outputCol="{}_indexed".format(c))]
```

(continues on next page)
for c in categoricalCols ]

# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.
˓→getOutputCol(),
outputCol="(0)_encoded".format(indexer.
˓→getOutputCol()))
       for indexer in indexers ]

assembler = VectorAssembler(inputCols=[encoder.
˓→getOutputCol() for encoder in encoders] + continuousCols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])

model = pipeline.fit(df)

data = model.transform(df)

return data.select(indexCol,'features')

Two in one:

def get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol,
˓→dropLast=False):
    
    
    Get dummy variables and concat with continuous variables for ml
    \-modeling.
    :param df: the dataframe
    :param categoricalCols: the name list of the categorical data
    :param continuousCols: the name list of the numerical data
    :param labelCol: the name of label column
    :param dropLast: the flag of drop last column
    :return: feature matrix
    
    :author: Wenqiang Feng
    :email: von198@gmail.com

    >>> df = spark.createDataFrame([[
       (0, "a"),
       (1, "b"),
       (2, "c"),
       (3, "a"),
       (4, "a"),
       (5, "c")
    ], ["id", "category"])

    >>> indexCol = 'id'
    >>> categoricalCols = ['category']
    >>> continuousCols = []
```python
>>> labelCol = []

>>> mat = get_dummy(df,indexCol,categoricalCols,continuousCols,labelCol)
>>> mat.show()

```
elif indexCol and not labelCol:
    # for unsupervised learning
    return data.select(indexCol,'features')
elif not indexCol and not labelCol:
    # for unsupervised learning
    return data.select('features')

4. Transform the dataset to DataFrame

```python
transformed= transData(df)
transformed.show(5, False)
```

```
+-----------------+
|features         |
+-----------------+
|[5.1,3.5,1.4,0.2]|
|[4.9,3.0,1.4,0.2]|
|[4.7,3.2,1.3,0.2]|
|[4.6,3.1,1.5,0.2]|
|[5.0,3.6,1.4,0.2]|
+-----------------+
only showing top 5 rows
```

5. Deal With Categorical Variables

```python
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated
# as continuous.
featureIndexer = VectorIndexer(inputCol="features", 
    outputCol="indexedFeatures", 
    maxCategories=4).fit(transformed)
data = featureIndexer.transform(transformed)
```

```python
Now you check your dataset with

data.show(5,True)
```

you will get

```
+-----------------+-----------------+
| features| indexedFeatures|
+-----------------+-----------------+
|[5.1,3.5,1.4,0.2]| [5.1,3.5,1.4,0.2]|
(continues on next page)```
Learning Apache Spark with Python

(continued from previous page)

| [4.9, 3.0, 1.4, 0.2] | [4.9, 3.0, 1.4, 0.2] |
| [4.7, 3.2, 1.3, 0.2] | [4.7, 3.2, 1.3, 0.2] |
| [4.6, 3.1, 1.5, 0.2] | [4.6, 3.1, 1.5, 0.2] |
| [5.0, 3.6, 1.4, 0.2] | [5.0, 3.6, 1.4, 0.2] |

+-----------------+-----------------+
only showing top 5 rows

Note: Since clustering algorithms including k-means use distance-based measurements to determine the similarity between data points, it’s strongly recommended to standardize the data to have a mean of zero and a standard deviation of one.

6. Elbow method to determine the optimal number of clusters for k-means clustering

```python
import numpy as np
cost = np.zeros(20)
for k in range(2, 20):
    kmeans = KMeans()
        .setK(k)
        .setSeed(1)
        .setFeaturesCol("indexedFeatures")
        .setPredictionCol("cluster")
    model = kmeans.fit(data)
    cost[k] = model.computeCost(data)  # requires Spark 2.0 or later
```

```python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots(1, 1, figsize=(8, 6))
ax.plot(range(2, 20), cost[2:20])
ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.show()
```

In my opinion, sometimes it’s hard to choose the optimal number of the clusters by using the elbow method. As shown in the following Figure, you can choose 3, 5 or even 8. I will choose 3 in this demo.

• Silhouette analysis

```python
# PySpark libraries
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col, percent_rank, lit
from pyspark.sql.window import Window
from pyspark.sql import DataFrame, Row
```

(continues on next page)
12.1. K-Means Model
from pyspark.sql.types import StructType
from functools import reduce  # For Python 3.x

from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

def optimal_k(df_in, index_col, k_min, k_max, num_runs):
    '''
    Determine optimal number of clusters by using Silhouette Score Analysis.
    :param df_in: the input dataframe
    :param index_col: the name of the index column
    :param k_min: the minmum number of the clusters
    :param k_max: the maxmum number of the clusters
    :param num_runs: the number of runs for each fixed clusters
    :return k: optimal number of the clusters
    :return silh_lst: Silhouette score
    :return r_table: the running results table
    
    :author: Wenqiang Feng
    :email: von198@gmail.com
    '''

    start = time.time()
    silh_lst = []
    k_lst = np.arange(k_min, k_max+1)

    r_table = df_in.select(index_col).toPandas()
    r_table = r_table.set_index(index_col)
    centers = pd.DataFrame()

    for k in k_lst:
        silh_val = []
        for run in np.arange(1, num_runs+1):

            # Trains a k-means model.
            kmeans = KMeans()
            .setK(k)\n            .setSeed(int(np.random.randint(100, size=1)))
            model = kmeans.fit(df_in)

            # Make predictions
            predictions = model.transform(df_in)
            r_table['cluster_{k}_{run}'.format(k=k, run=run)]= predictions.
                ->select('prediction').toPandas()

            # Evaluate clustering by computing Silhouette score
            evaluator = ClusteringEvaluator()
            silhouette = evaluator.evaluate(predictions)
            silh_val.append(silhouette)

    silh_lst.append(np.mean(silh_val))

    return k, np.argmax(silh_lst), silh_lst, r_table

(continues on next page)
silh_array = np.asanyarray(silh_val)
silh_lst.append(silh_array.mean())

elapsed = time.time() - start

silhouette = pd.DataFrame(list(zip(k_lst, silh_lst)), columns=['k', 'silhouette'])

print('+' * 80)
print('| The finding optimal k phase took %8.0f s. |' % elapsed)
print('+' * 80)

return k_lst[np.argmax(silh_lst, axis=0)], silhouette, r_table

k, silh_lst, r_table = optimal_k(scaledData, index_col, k_min, k_max, num_runs)

spark.createDataFrame(silh_lst).show()

+---+------------------+
| k | silhouette       |
+---+------------------+
| 3 | 0.8045154385557953 |
| 4 | 0.6993528775512052 |
| 5 | 0.6689286654221447 |
| 6 | 0.6356184024841809 |
| 7 | 0.7174102265711756 |
| 8 | 0.6720861758298997 |
| 9 | 0.601771359881241 |
| 10| 0.6292447334578428 |
+---+------------------+

From the silhouette list, we can choose 3 as the optimal number of the clusters.

**Warning:** ClusteringEvaluator in pyspark.ml.evaluation requires Spark 2.4 or later!!

7. Pipeline Architecture

```python
from pyspark.ml.clustering import KMeans, KMeansModel

kmeans = KMeans() \
    .setK(3) \
    .setFeaturesCol("indexedFeatures") \
    .setPredictionCol("cluster")
```

(continues on next page)
# Chain indexer and tree in a Pipeline

```python
pipeline = Pipeline(stages=[featureIndexer, kmeans])

model = pipeline.fit(transformed)

cluster = model.transform(transformed)
```

8. k-means clusters

```python
cluster = model.transform(transformed)
```

<table>
<thead>
<tr>
<th>features</th>
<th>indexedFeatures</th>
<th>cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>[5.1, 3.5, 1.4, 0.2]</td>
<td>[5.1, 3.5, 1.4, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.9, 3.0, 1.4, 0.2]</td>
<td>[4.9, 3.0, 1.4, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.7, 3.2, 1.3, 0.2]</td>
<td>[4.7, 3.2, 1.3, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.6, 3.1, 1.5, 0.2]</td>
<td>[4.6, 3.1, 1.5, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[5.0, 3.6, 1.4, 0.2]</td>
<td>[5.0, 3.6, 1.4, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[5.4, 3.9, 1.7, 0.4]</td>
<td>[5.4, 3.9, 1.7, 0.4]</td>
<td>1</td>
</tr>
<tr>
<td>[4.6, 3.4, 1.4, 0.3]</td>
<td>[4.6, 3.4, 1.4, 0.3]</td>
<td>1</td>
</tr>
<tr>
<td>[5.0, 3.4, 1.5, 0.2]</td>
<td>[5.0, 3.4, 1.5, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.4, 2.9, 1.4, 0.2]</td>
<td>[4.4, 2.9, 1.4, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.9, 3.1, 1.5, 0.1]</td>
<td>[4.9, 3.1, 1.5, 0.1]</td>
<td>1</td>
</tr>
<tr>
<td>[5.4, 3.7, 1.5, 0.2]</td>
<td>[5.4, 3.7, 1.5, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.8, 3.4, 1.6, 0.2]</td>
<td>[4.8, 3.4, 1.6, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[4.8, 3.0, 1.4, 0.1]</td>
<td>[4.8, 3.0, 1.4, 0.1]</td>
<td>1</td>
</tr>
<tr>
<td>[4.3, 3.0, 1.1, 0.1]</td>
<td>[4.3, 3.0, 1.1, 0.1]</td>
<td>1</td>
</tr>
<tr>
<td>[5.8, 4.0, 1.2, 0.2]</td>
<td>[5.8, 4.0, 1.2, 0.2]</td>
<td>1</td>
</tr>
<tr>
<td>[5.7, 4.4, 1.5, 0.4]</td>
<td>[5.7, 4.4, 1.5, 0.4]</td>
<td>1</td>
</tr>
<tr>
<td>[5.4, 3.9, 1.3, 0.4]</td>
<td>[5.4, 3.9, 1.3, 0.4]</td>
<td>1</td>
</tr>
<tr>
<td>[5.1, 3.5, 1.4, 0.3]</td>
<td>[5.1, 3.5, 1.4, 0.3]</td>
<td>1</td>
</tr>
<tr>
<td>[5.7, 3.8, 1.7, 0.3]</td>
<td>[5.7, 3.8, 1.7, 0.3]</td>
<td>1</td>
</tr>
<tr>
<td>[5.1, 3.8, 1.5, 0.3]</td>
<td>[5.1, 3.8, 1.5, 0.3]</td>
<td>1</td>
</tr>
</tbody>
</table>

only showing top 20 rows
RFM ANALYSIS

<table>
<thead>
<tr>
<th>Segment</th>
<th>RFM</th>
<th>Description</th>
<th>Marketing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Customers</td>
<td>111</td>
<td>Bought most recently and most often, and spend the most</td>
<td>No price incentives, new products, and loyalty programs</td>
</tr>
<tr>
<td>Loyal Customers</td>
<td>X1X</td>
<td>Buy most frequently</td>
<td>Use R and M to further segment</td>
</tr>
<tr>
<td>Big Spenders</td>
<td>XX1</td>
<td>Spend the most</td>
<td>Market your most expensive products</td>
</tr>
<tr>
<td>Almost Lost</td>
<td>311</td>
<td>Haven’t purchased for some time, but purchased frequently and spend the most</td>
<td>Aggressive price incentives</td>
</tr>
<tr>
<td>Lost Customers</td>
<td>411</td>
<td>Haven’t purchased for some time, but purchased frequently and spend the most</td>
<td>Aggressive price incentives</td>
</tr>
<tr>
<td>Lost Cheap Customers</td>
<td>444</td>
<td>Last purchased long ago, purchased few, and spent little</td>
<td>Don’t spend too much trying to re-acquire</td>
</tr>
</tbody>
</table>

The above figure source: Blast Analytics Marketing

RFM is a method used for analyzing customer value. It is commonly used in database marketing and direct marketing and has received particular attention in retail and professional services industries. More details
can be found at Wikipedia RFM_wikipedia.

RFM stands for the three dimensions:
- Recency – How recently did the customer purchase? i.e. Duration since last purchase
- Frequency – How often do they purchase? i.e. Total number of purchases
- Monetary Value – How much do they spend? i.e. Total money this customer spent

13.1 RFM Analysis Methodology

RFM Analysis contains three main steps:

13.1.1 1. Build the RFM features matrix for each customer

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
</tr>
</thead>
<tbody>
<tr>
<td>14911</td>
<td>1</td>
<td>248</td>
<td>132572.62</td>
</tr>
<tr>
<td>12748</td>
<td>0</td>
<td>224</td>
<td>29072.1</td>
</tr>
<tr>
<td>17841</td>
<td>1</td>
<td>169</td>
<td>40340.78</td>
</tr>
<tr>
<td>14606</td>
<td>1</td>
<td>128</td>
<td>11713.85</td>
</tr>
<tr>
<td>15311</td>
<td>0</td>
<td>118</td>
<td>59419.34</td>
</tr>
</tbody>
</table>

only showing top 5 rows

13.1.2 2. Determine cutting points for each feature

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
<th>r_seg</th>
<th>f_seg</th>
<th>m_seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>17420</td>
<td>50</td>
<td>3</td>
<td>598.83</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>16861</td>
<td>59</td>
<td>3</td>
<td>151.65</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>16503</td>
<td>106</td>
<td>5</td>
<td>1421.43</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15727</td>
<td>16</td>
<td>7</td>
<td>5178.96</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>17389</td>
<td>0</td>
<td>43</td>
<td>31300.08</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

only showing top 5 rows

13.1.3 3. Determine the RFM scores and summarize the corresponding business value

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
<th>r_seg</th>
<th>f_seg</th>
<th>m_seg</th>
<th>RFMScore</th>
</tr>
</thead>
</table>

(continues on next page)
The corresponding business description and marketing value:

<table>
<thead>
<tr>
<th>Segment</th>
<th>RFM</th>
<th>Description</th>
<th>Marketing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Customers</td>
<td>111</td>
<td>Bought most recently and most often, and spend the most</td>
<td>No price incentives, new products, and loyalty programs</td>
</tr>
<tr>
<td>Loyal Customers</td>
<td>X1X</td>
<td>Buy most frequently</td>
<td>Use R and M to further segment</td>
</tr>
<tr>
<td>Big Spenders</td>
<td>XX1</td>
<td>Spend the most</td>
<td>Market your most expensive products</td>
</tr>
<tr>
<td>Almost Lost</td>
<td>311</td>
<td>Haven't purchased for some time, but purchased frequently and spend the most</td>
<td>Aggressive price incentives</td>
</tr>
<tr>
<td>Lost Customers</td>
<td>411</td>
<td>Haven't purchased for some time, but purchased frequently and spend the most</td>
<td>Aggressive price incentives</td>
</tr>
<tr>
<td>Lost Cheap Customers</td>
<td>444</td>
<td>Last purchased long ago, purchased few, and spent little</td>
<td>Don't spend too much trying to re-acquire</td>
</tr>
</tbody>
</table>

Fig. 1: Source: Blast Analytics Marketing
13.2 Demo

- The Jupyter notebook can be download from Data Exploration.
- The data can be download from German Credit.

13.2.1 Load and clean data

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark RFM example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```python
df_raw = spark.read.format('com.databricks.spark.csv').
    options(header='true', \
            inferschema='true').
    load("Online Retail.csv",header=True);
```

check the data set

```python
df_raw.show(5)
df_raw.printSchema()
```

Then you will get

```
+---------+---------+--------------------+--------+------------+---------+----+
|InvoiceNo| StockCode| Description         |Quantity|
+---------+---------+--------------------+--------+------------+---------+----+
|536365   | 85123A  | WHITE HANGING HEA...| 6      |
|17850    |         |                     |        |
|536365   | 71053   | WHITE METAL LANTERN | 6      |
|17850    |         |                     |        |
|536365   | 84406B  | CREAM CUPID HEART...| 8      |
|17850    |         |                     |        |
|536365   | 84029G  | KNITTED UNION FLA...| 6      |
|17850    |         |                     |        |
|536365   | 84029E  | RED WOOLLY HOTTIE...| 6      |
|17850    |         |                     |        |
```

only showing top 5 rows
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: string (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: integer (nullable = true)
|-- Country: string (nullable = true)

3. Data clean and data manipulation
   • check and remove the null values

```python
from pyspark.sql.functions import count

def my_count(df_in):
    df_in.agg(*[count(c).alias(c) for c in df_in.columns]).show()

my_count(df_raw)
```

Since the count results are not the same, we have some null value in the `CustomerID` column. We can drop these records from the dataset.

```python
df = df_raw.dropna(how='any')
my_count(df)
```

• Deal with the `InvoiceDate`

13.2. Demo
```python
from pyspark.sql.functions import to_utc_timestamp, unix_timestamp, lit,
    datediff, col

timeFmt = "MM/dd/yy HH:mm"

df = df.withColumn('NewInvoiceDate',
    to_utc_timestamp(unix_timestamp(col('InvoiceDate'),
    timeFmt).cast('timestamp'), 'UTC'))

df.show(5)

<table>
<thead>
<tr>
<th>InvoiceNo</th>
<th>StockCode</th>
<th>Description</th>
<th>Quantity</th>
<th>InvoiceDate</th>
<th>UnitPrice</th>
<th>CustomerID</th>
<th>Country</th>
<th>NewInvoiceDate</th>
</tr>
</thead>
</table>
| 536365    | 85123A    | WHITE HANGING HEART BASKET             | 6        | 12/1/10 8:26| 2.55      | 17850      | United Kingdom| 2010-12-01 08:26:
| 536365    | 71053     | WHITE METAL LANTERN                     | 6        | 12/1/10 8:26| 3.39      | 17850      | United Kingdom| 2010-12-01 08:26:
| 536365    | 84406B    | CREAM CUPID HEART BASKET               | 8        | 12/1/10 8:26| 2.75      | 17850      | United Kingdom| 2010-12-01 08:26:
| 536365    | 84029G    | KNITTED UNION FLAME LANTERN            | 6        | 12/1/10 8:26| 3.39      | 17850      | United Kingdom| 2010-12-01 08:26:
| 536365    | 84029E    | RED WOOLLY HOTTIE BASKET               | 6        | 12/1/10 8:26| 3.39      | 17850      | United Kingdom| 2010-12-01 08:26:
```

**Warning:** The spark is pretty sensitive to the date format!

- calculate total price

```python
from pyspark.sql.functions import round

df = df.withColumn('TotalPrice', round( df.Quantity * df.UnitPrice, 2 ) )
```

- calculate the time difference

```python
from pyspark.sql.functions import mean, min, max, sum, datediff, to_date
date_max = df.select(max('NewInvoiceDate')).toPandas()
current = to_utc_timestamp( unix_timestamp(lit(str(date_max.iloc[0][0])), 
    'yy-MM-dd HH:mm').cast('timestamp'), 'UTC' )

# Calculate Duration
df = df.withColumn('Duration', datediff(lit(current), 'NewInvoiceDate'))
```
• build the Recency, Frequency and Monetary

```python
def groupBy(df, column, agg_func)
    return df.groupBy(column).agg(agg_func)

recency = df.groupBy('CustomerID').agg(min('Duration').alias('Recency'))
frequency = df.groupBy('CustomerID', 'InvoiceNo').count().
    .groupBy('CustomerID')
    .agg(count('*').alias("Frequency"))
monetary = df.groupBy('CustomerID').agg(round(sum('TotalPrice'), 2).alias('Monetary'))
rfm = recency.join(frequency, 'CustomerID', how = 'inner')
    .join(monetary, 'CustomerID', how = 'inner')
```

```python
table.show(5)
+
<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
</tr>
</thead>
<tbody>
<tr>
<td>17420</td>
<td>50</td>
<td>3</td>
<td>598.83</td>
</tr>
<tr>
<td>16861</td>
<td>59</td>
<td>3</td>
<td>151.65</td>
</tr>
<tr>
<td>16503</td>
<td>106</td>
<td>5</td>
<td>1421.43</td>
</tr>
<tr>
<td>15727</td>
<td>16</td>
<td>7</td>
<td>5178.96</td>
</tr>
<tr>
<td>17389</td>
<td>0</td>
<td>43</td>
<td>31300.08</td>
</tr>
</tbody>
</table>
+
only showing top 5 rows
```

### 13.2.2 RFM Segmentation

4. Determine cutting points

In this section, you can use the techniques (statistical results and visualizations) in Data Exploration section to help you determine the cutting points for each attribute. In my opinion, the cutting points are mainly depend on the business sense. You’s better talk to your market people and get feedback and suggestion from them. I will use the quantile as the cutting points in this demo.

```python
cols = ['Recency', 'Frequency', 'Monetary']
describe_pd(rfm, cols, 1)
```

```markdown
<table>
<thead>
<tr>
<th>summary</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4372.0</td>
<td>4372.0</td>
<td>4372.0</td>
</tr>
<tr>
<td>mean</td>
<td>91.5819853613907</td>
<td>5.07548032936871</td>
<td>1898.4597003659655</td>
</tr>
<tr>
<td>stddev</td>
<td>100.7721393138483</td>
<td>9.338754163574727</td>
<td>8219.345141139722</td>
</tr>
<tr>
<td>min</td>
<td>0.0</td>
<td>1.0</td>
<td>-4287.63</td>
</tr>
<tr>
<td>max</td>
<td>373.0</td>
<td>248.0</td>
<td>279489.02</td>
</tr>
<tr>
<td>25%</td>
<td>16.0</td>
<td>1.0</td>
<td>293.362499999999995</td>
</tr>
<tr>
<td>50%</td>
<td>50.0</td>
<td>3.0</td>
<td>648.075</td>
</tr>
<tr>
<td>75%</td>
<td>143.0</td>
<td>5.0</td>
<td>1611.725</td>
</tr>
</tbody>
</table>
```

The user defined function by using the cutting points:
def RScore(x):
    if x <= 16:
        return 1
    elif x <= 50:
        return 2
    elif x <= 143:
        return 3
    else:
        return 4

def FScore(x):
    if x <= 1:
        return 4
    elif x <= 3:
        return 3
    elif x <= 5:
        return 2
    else:
        return 1

def MScore(x):
    if x <= 293:
        return 4
    elif x <= 648:
        return 3
    elif x <= 1611:
        return 2
    else:
        return 1

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType

R_udf = udf(lambda x: RScore(x), StringType())
F_udf = udf(lambda x: FScore(x), StringType())
M_udf = udf(lambda x: MScore(x), StringType())

5. RFM Segmentation

rfm_seg = rfm.withColumn("r_seg", R_udf("Recency"))
rfm_seg = rfm_seg.withColumn("f_seg", F_udf("Frequency"))
rfm_seg = rfm_seg.withColumn("m_seg", M_udf("Monetary"))
rfm_seg.show(5)

+----------+-------+---------+--------+-----+-----+-----+
|CustomerID|Recency|Frequency|Monetary|r_seg|f_seg|m_seg|
|----------+-------+---------+--------+-----+-----+-----|
| 17420| 50| 3| 598.83| 2| 3| 2|
| 16861| 59| 3| 151.65| 3| 3| 1|
| 16503| 106| 5| 1421.43| 3| 2| 3|
| 15727| 16| 7| 5178.96| 1| 1| 4|
| 17389| 0| 43| 31300.08| 1| 1| 4|
(continues on next page)
only showing top 5 rows

```python
rfm_seg = rfm_seg.withColumn('RFMScore', F.concat(F.col('r_seg'), F.col('f_seg'), F.col('m_seg')))
rfm_seg.sort(F.col('RFMScore')).show(5)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Recency</th>
<th>Frequency</th>
<th>Monetary</th>
<th>r_seg</th>
<th>f_seg</th>
<th>m_seg</th>
<th>RFMScore</th>
</tr>
</thead>
<tbody>
<tr>
<td>17988</td>
<td>11</td>
<td>8</td>
<td>191.17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>16892</td>
<td>1</td>
<td>7</td>
<td>496.84</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>112</td>
</tr>
<tr>
<td>16668</td>
<td>15</td>
<td>6</td>
<td>306.72</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>112</td>
</tr>
<tr>
<td>16554</td>
<td>3</td>
<td>7</td>
<td>641.55</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>112</td>
</tr>
<tr>
<td>16500</td>
<td>4</td>
<td>6</td>
<td>400.86</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>112</td>
</tr>
</tbody>
</table>

only showing top 5 rows

<table>
<thead>
<tr>
<th>RFMScore</th>
<th>avg(Recency)</th>
<th>avg(Monetary)</th>
<th>avg(Frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>11.0</td>
<td>191.17</td>
<td>8.0</td>
</tr>
<tr>
<td>112</td>
<td>8.0</td>
<td>505.9775</td>
<td>7.5</td>
</tr>
<tr>
<td>113</td>
<td>7.237113402061856</td>
<td>1223.3604123711339</td>
<td>7.752577319587629</td>
</tr>
<tr>
<td>114</td>
<td>6.035123966942149</td>
<td>8828.888595041324</td>
<td>18.882231404958677</td>
</tr>
<tr>
<td>121</td>
<td>9.6</td>
<td>207.24</td>
<td>4.4</td>
</tr>
</tbody>
</table>

only showing top 5 rows

13.2.3 Statistical Summary

6. Statistical Summary

- simple summary

```python
rfm_seg.groupBy('RFMScore')
    .agg({'Recency': 'mean', 'Frequency': 'mean', 'Monetary': 'mean'}
    .sort(F.col('RFMScore')).show(5)
```

<table>
<thead>
<tr>
<th>RFMScore</th>
<th>avg(Recency)</th>
<th>avg(Monetary)</th>
<th>avg(Frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>11.0</td>
<td>191.17</td>
<td>8.0</td>
</tr>
<tr>
<td>112</td>
<td>8.0</td>
<td>505.9775</td>
<td>7.5</td>
</tr>
<tr>
<td>113</td>
<td>7.237113402061856</td>
<td>1223.3604123711339</td>
<td>7.752577319587629</td>
</tr>
<tr>
<td>114</td>
<td>6.035123966942149</td>
<td>8828.888595041324</td>
<td>18.882231404958677</td>
</tr>
<tr>
<td>121</td>
<td>9.6</td>
<td>207.24</td>
<td>4.4</td>
</tr>
</tbody>
</table>

- complex summary

```python
grp = 'RFMScore'
um_cols = ['Recency', 'Frequency', 'Monetary']
df_input = rfm_seg
quantile_grouped = quantile_agg(df_input, grp, num_cols)
```

(continues on next page)
quantile_grouped.toPandas().to_csv(output_dir+'quantile_grouped.csv')

deciles_grouped = deciles_agg(df_input,grp,num_cols)
deciles_grouped.toPandas().to_csv(output_dir+'deciles_grouped.csv')

13.3 Extension

You can also apply the K-means clustering in Clustering section to do the segmentation.

13.3.1 Build feature matrix

1. build dense feature matrix

```python
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

# method 1 (good for small feature):
def transData(row):
    return Row(label=row["Sales"],
               features=Vectors.dense([row["TV"],
                                       row["Radio"],
                                       row["Newspaper"]]))

# Method 2 (good for large features):
def transData(data):
    return data.rdd.map(lambda r: [r[0],Vectors.dense(r[1:]))).toDF(["CustomerID","rfm"])

transformed= transData(rfm)
transformed.show(5)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>rfm</th>
</tr>
</thead>
<tbody>
<tr>
<td>17420</td>
<td>[50.0,3.0,598.83]</td>
</tr>
<tr>
<td>16861</td>
<td>[59.0,3.0,151.65]</td>
</tr>
<tr>
<td>16503</td>
<td>[106.0,5.0,1421.43]</td>
</tr>
<tr>
<td>15727</td>
<td>[16.0,7.0,5178.96]</td>
</tr>
<tr>
<td>17389</td>
<td>[0.0,43.0,31300.08]</td>
</tr>
</tbody>
</table>

2. Scaler the feature matrix

```python
from pyspark.ml.feature import MinMaxScaler

scaler = MinMaxScaler(inputCol="rfm",
```

(continues on next page)
```python
outputCol="features")
scalerModel = scaler.fit(transformed)
scaledData = scalerModel.transform(transformed)
scaledData.show(5,False)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>rfm</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>17420</td>
<td>[50.0, 3.0, 598.83]</td>
<td>[0.13404825737265416, 0.008097165991902834, 0.01721938714830836]</td>
</tr>
<tr>
<td>16861</td>
<td>[59.0, 3.0, 151.65]</td>
<td>[0.1581769436997319, 0.008097165991902834, 0.01564357039241953]</td>
</tr>
<tr>
<td>16503</td>
<td>[106.0, 5.0, 1421.43]</td>
<td>[0.28418230563002683, 0.016194331983805668, 0.02011814573186342]</td>
</tr>
<tr>
<td>15727</td>
<td>[16.0, 7.0, 5178.96]</td>
<td>[0.04289544235924933, 0.024291497975708502, 0.033592858922501]</td>
</tr>
<tr>
<td>17389</td>
<td>[0.0, 43.0, 31300.08]</td>
<td>[0.0, 0.1700404858299595, 0.12540746393334334]</td>
</tr>
</tbody>
</table>

only showing top 5 rows

### 13.3.2 K-means clustering

3. Find optimal number of cluster

I will present two popular ways to determine the optimal number of the cluster.

- elbow analysis

```python
#PySpark libraries
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col, percent_rank, lit
from pyspark.sql.window import Window
from pyspark.sql import DataFrame, Row
from pyspark.sql.types import StructType
from functools import reduce  # For Python 3.x

from pyspark.ml.clustering import KMeans
# from pyspark.ml.evaluation import ClusteringEvaluator  # requires Spark 2.4 or later

import numpy as np
cost = np.zeros(20)
for k in range(2, 20):
    kmeans = KMeans()
```
...\setK(k)\ 
.\setSeed(1) \ 
.\setFeaturesCol("features")\ 
.\setPredictionCol("cluster")

model = kmeans.fit(scaledData)
cost[k] = model.computeCost(scaledData) # requires Spark 2.0 or later

import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import seaborn as sbs
from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots(1,1, figsize =(8,6))
ax.plot(range(2,20),cost[2:20], marker = "o")
ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.show()

In my opinion, sometimes it’s hard to choose the number of the clusters. As shown in Figure Cost v.s. the number of the clusters, you can choose 3, 5 or even 8. I will choose 3 in this demo.

- Silhouette analysis

#PySpark libraries
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.sql.functions import col, percent_rank, lit
from pyspark.sql.window import Window
from pyspark.sql import DataFrame, Row
from pyspark.sql.types import StructType
from functools import reduce # For Python 3.x

from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

def optimal_k(df_in,index_col,k_min, k_max,num_runs):
    '''
    Determine optimal number of clusters by using Silhouette Score Analysis.
    :param df_in: the input dataframe
    :param index_col: the name of the index column
    :param k_min: the train dataset
    :param k_min: the minimum number of the clusters
    :param k_max: the maximum number of the clusters
    :param num_runs: the number of runs for each fixed clusters
    :return k: optimal number of the clusters
    :return silh_lst: Silhouette score
    :return r_table: the running results table
    (continues on next page)
Fig. 2: Cost v.s. the number of the clusters
start = time.time()
silh_lst = []
k_lst = np.arange(k_min, k_max+1)

r_table = df_in.select(index_col).toPandas()
r_table = r_table.set_index(index_col)
centers = pd.DataFrame()

for k in k_lst:
    silh_val = []
    for run in np.arange(1, num_runs+1):

        # Trains a k-means model.
        kmeans = KMeans()
        .setK(k)
        .setSeed(int(np.random.randint(100, size=1)))
        model = kmeans.fit(df_in)

        # Make predictions
        predictions = model.transform(df_in)
        r_table['cluster_{k}_{run}'.format(k=k, run=run)] = predictions.

        # Evaluate clustering by computing Silhouette score
        evaluator = ClusteringEvaluator()
        silhouette = evaluator.evaluate(predictions)
        silh_val.append(silhouette)

    silh_array=np.asanyarray(silh_val)
    silh_lst.append(silh_array.mean())

elapsed = time.time() - start

silhouette = pd.DataFrame(list(zip(k_lst,silh_lst)),columns = ['k', 'silhouette'])

print('+------------------------------------------------------------+')
print('| The finding optimal k phase took %8.0f s. |' % elapsed)
print('+------------------------------------------------------------+')

return k_lst[np.argmax(silh_lst, axis=0)], silhouette, r_table

k, silh_lst, r_table = optimal_k(scaledData, index_col, k_min, k_max, num_runs)
The finding optimal k phase took 1783 s.

```python
spark.createDataFrame(silh_lst).show()
```

```
+---+------------------+
| k | silhouette       |
+---+------------------+
| 3 | 0.8045154385557953 |
| 4 | 0.6993528775512052 |
| 5 | 0.6689286654221447 |
| 6 | 0.6356184024841809 |
| 7 | 0.7174102265711756 |
| 8 | 0.6720861758298997 |
| 9 | 0.601771359881241 |
| 10| 0.6292447334578428 |
+---+------------------+
```

From the silhouette list, we can choose 3 as the optimal number of the clusters.

**Warning:** ClusteringEvaluator in pyspark.ml.evaluation requires Spark 2.4 or later!!

4. K-means clustering

```python
k = 3
kmeans = KMeans().setK(k).setSeed(1)
model = kmeans.fit(scaledData)
# Make predictions
predictions = model.transform(scaledData)
predictions.show(5,False)
```

```
+----------+-------------------+--------------------+----------+
|CustomerID| rfm| features|prediction|
+----------+-------------------+--------------------+----------+
| 17420    | [50.0, 3.0, 598.83] | [0.13404825737265...| 0|
| 16861    | [59.0, 3.0, 151.65] | [0.15817694369973...| 0|
| 16503    | [106.0, 5.0, 1421.43]| [0.28418230563002...| 2|
| 15727    | [16.0, 7.0, 5178.96] | [0.04289544235924...| 0|
| 17389    | [0.0, 43.0, 31300.08]| [0.0, 0.1700404858...| 0|
+----------+-------------------+--------------------+----------+
```

only showing top 5 rows

13.3.3 Statistical summary

5. statistical summary

13.3. Extension
results = rfm.join(predictions.select('CustomerID', 'prediction'), 'CustomerID', how='left')
results.show(5)

+----------+-------+---------+--------+----------+
|CustomerID|Recency|Frequency|Monetary|prediction|
+----------+-------+---------+--------+----------+
| 13098| 1| 41|28658.88| 0|
| 13248| 124| 2| 465.68| 2|
| 13452| 259| 2| 590.0| 1|
| 13460| 29| 2| 183.44| 0|
| 13518| 85| 1| 659.44| 0|
+----------+-------+---------+--------+----------+
only showing top 5 rows

• simple summary

results.groupBy('prediction')
.agg({'Recency': 'mean',
      'Frequency': 'mean',
      'Monetary': 'mean'} )
   .sort(F.col('prediction')).show(5)

+----------+------------------+------------------+------------------+
|prediction| avg(Recency)| avg(Monetary)| avg(Frequency)|
+----------+------------------+------------------+------------------+
| 0|30.966337980278816|2543.0355321319284| 6.514450867052023|
| 1|296.02403846153845|407.16831730769206|1.5592948717948718|
| 2|154.40148698884758| 702.5096406443623| 2.550185873605948|
+----------+------------------+------------------+------------------+

• complex summary

grp = 'RFMScore'
num_cols = ['Recency', 'Frequency', 'Monetary']
df_input = results
quantile_grouped = quantile_agg(df_input, grp, num_cols)
quantile_grouped.toPandas().to_csv(output_dir+'quantile_grouped.csv')

deciles_grouped = deciles_agg(df_input, grp, num_cols)
deciles_grouped.toPandas().to_csv(output_dir+'deciles_grouped.csv')
Chinese proverb

Articles showed more than intended. – Xianglong Shen

14.1 Text Collection

14.1.1 Image to text

- My `img2txt` function

```python
def img2txt(img_dir):
    
    ""
    convert images to text
    ""
    import os, PythonMagick
```

(continues on next page)
from datetime import datetime
import PyPDF2

from PIL import Image
import pytesseract

f = open('doc4img.txt', 'wa')
for img in [img_file for img_file in os.listdir(img_dir) if (img_file.endswith(".png") or img_file.endswith(".jpg") or img_file.endswith(".jpeg"))]:
    start_time = datetime.now()
    input_img = img_dir + '/' + img

    print('---------------------------------------------------------------
   --->
   --->
   --->
   --->
   --->
   --->
   --->

    print(img)
    print('Converting ' + img + '......')
    print('--------------------------------------------------------------------

    # extract the text information from images
    text = pytesseract.image_to_string(Image.open(input_img))
    print(text)

    # output text file
    f.write( img + "\n")
    f.write(text.encode('utf-8'))

    print "CPU Time for converting" + img +":"+ str(datetime.now() - start_time) +"\n"
    f.write( "\n--------------------------------------------------------------------
   --->\n   --->\n   --->\n   --->\n   --->\n   --->\n   --->"
    f.close()

• Demo

I applied my img2txt function to the image in Image folder.

image_dir = r"Image"

img2txt(image_dir)

Then I got the following results:

--------------------------------------------------------------------
feng.pdf_0.png
Converting feng.pdf_0.png......
Wenqiang Feng is Data Scientist for DST’s Applied Analytics Group. Dr. Feng’s responsibilities include providing DST clients with access to cutting-edge skills and technologies, including Big Data analytic solutions, advanced analytic and data enhancement techniques, and modeling.

Dr. Feng has deep analytic expertise in data mining, analytic systems, machine learning algorithms, business intelligence, and applying Big Data tools to strategically solve industry problems in a cross-functional business. Before joining the DST Applied Analytics Group, Dr. Feng holds a MA Data Science Fellow at The Institute for Mathematics and Its Applications (IMA) at the University of Minnesota. While there, he helped startup companies make marketing decisions based on deep predictive analytics.

Dr. Feng graduated from University of Tennessee, Knoxville with PhD in Computational mathematics and Master’s degree in Statistics. He also holds Master’s degree in Computational Mathematics at Missouri University of Science and Technology (MST) and Master’s degree in Applied Mathematics at University of science and technology of China (USTC).

**14.1.2 Image Enhanced to text**

- My `img2txt_enhance` function

```python
def img2txt_enhance(img_dir, scaler):
    ""
    convert images files to text
    ""

    import numpy as np
    import os, PythonMagick
    from datetime import datetime
    import PyPDF2
```

(continues on next page)
from PIL import Image, ImageEnhance, ImageFilter
import pytesseract

f = open('doc4img.txt', 'wa')
for img in [img_file for img_file in os.listdir(img_dir) if (img_file.endswith(".png") or img_file.endswith(".jpg") or img_file.endswith(".jpeg"))]:

    start_time = datetime.now()

    input_img = img_dir + "/" + img
    enhanced_img = img_dir + "/" + "Enhanced" + "/" + img

    im = Image.open(input_img) # the second one
    im = im.filter(ImageFilter.MedianFilter())
    enhancer = ImageEnhance.Contrast(im)
    im = enhancer.enhance(1)
    im = im.convert('1')
    im.save(enhanced_img)

    for scale in np.ones(scaler):
        im = Image.open(enhanced_img) # the second one
        im = im.filter(ImageFilter.MedianFilter())
        enhancer = ImageEnhance.Contrast(im)
        im = enhancer.enhance(scale)
        im = im.convert('1')
        im.save(enhanced_img)

    print('---------------------------------------------------------------
       ----
    print(img)
    print('Converting ' + img + '.......')
    print('---------------------------------------------------------------
       ----

# extract the text information from images
    text = pytesseract.image_to_string(Image.open(enhanced_img))
    print(text)

# output text file
    f.write(img + "\n")
    f.write(text.encode('utf-8'))

    print "CPU Time for converting" + img + ":" + str(datetime.now() - start_time) + "\n"
    f.write("\n")

(continues on next page)
f.close()

• Demo

I applied my `img2txt_enhance` function to the following noised image in Enhance folder.

![Image](image)

```python
image_dir = r"Enhance"
pdf2txt_enhance(image_dir)
```

Then I got the following results:

```
noised.jpg
Converting noised.jpg.......  
zHHH
CPU Time for convertingnoised.jpg:0:00:00.135465
```

while the result from `img2txt` function is

```
noised.jpg
Converting noised.jpg.......  
,2 WW
CPU Time for convertingnoised.jpg:0:00:00.133508
```

which is not correct.

### 14.1.3 PDF to text

• My `pdf2txt` function

```python
def pdf2txt(pdf_dir, image_dir):
    
    """
    convert PDF to text
    """

    import os, PythonMagick
    from datetime import datetime
    import PyPDF2
```

(continues on next page)
from PIL import Image
import pytesseract
f = open('doc.txt', 'wa')
for pdf in [pdf_file for pdf_file in os.listdir(pdf_dir) if pdf_file.endswith(".pdf")]:
    start_time = datetime.now()
    input_pdf = pdf_dir + '/' + pdf
    pdf_im = PyPDF2.PdfFileReader(file(input_pdf, "rb"))
    npage = pdf_im.getNumPages()
    print('---------------------------------------------------------------
    -----')
    print(pdf)
    print('Converting %d pages.' % npage)
    print('---------------------------------------------------------------
    -----
    f.write("\n----------------------------------------------------------
    ----------")
    for p in range(npage):
        pdf_file = input_pdf + '][' + str(p) +']'
        image_file = image_dir + '/' + pdf+ '_' + str(p)+'.png'

        # convert PDF files to Images
        im = PythonMagick.Image()
        im.density('300')
        im.read(pdf_file)
        im.write(image_file)

        # extract the text information from images
        text = pytesseract.image_to_string(Image.open(image_file))

        #print(text)

        # output text file
        f.write(pdf + "\n")
        f.write(text.encode('utf-8'))

    print "CPU Time for converting" + pdf +":" + str(datetime.now() - start_time) +"\n"
    f.close()

• Demo
I applied my pdf2txt function to my scanned bio pdf file in pdf folder.
Then I got the following results:

```
 Wenqiang Feng
 Data Scientist
 DST APPLIED ANALYTICS GROUP

Wenqiang Feng is Data Scientist for DST’s Applied Analytics Group. Dr. Feng’s responsibilities include providing DST clients with access to cutting–edge skills and technologies, including Big Data analytic solutions, advanced analytic and data enhancement techniques and modeling.

Dr. Feng has deep analytic expertise in data mining, analytic systems, machine learning algorithms, business intelligence, and applying Big Data tools to strategically solve industry problems in a cross–functional business. Before joining the DST Applied Analytics Group, Dr. Feng holds a MA Data Science Fellow at The Institute for Mathematics and Its Applications (IMA) at the University of Minnesota. While there, he helped startup companies make marketing decisions based on deep predictive analytics.

Dr. Feng graduated from University of Tennessee, Knoxville with PhD in Computational mathematics and Master’s degree in Statistics. He also holds Master’s degree in Computational Mathematics at Missouri University of Science and Technology (MST) and Master’s degree in Applied Mathematics at University of science and technology of China (USTC).

CPU Time for convertingfeng.pdf:0:00:03.143800
```

14.1.4 Audio to text

- My audio2txt function
```python
def audio2txt(audio_dir):
    """ convert audio to text""

    import speech_recognition as sr
    r = sr.Recognizer()

    f = open('doc.txt','wa')
    for audio_n in [audio_file for audio_file in os.listdir(audio_dir) if audio_file.endswith(".wav")]:

        filename = audio_dir + "/" + audio_n

        # Read audio data
        with sr.AudioFile(filename) as source:
            audio = r.record(source)  # read the entire audio file

        # Google Speech Recognition
        text = r.recognize_google(audio)

        # output text file
        f.write( audio_n + ": ")
        f.write(text.encode('utf-8'))
        f.write(\n"

        print('You said: ' + text)

    f.close()```

- Demo

I applied my audio2txt function to my audio records in audio folder.

```python
audio_dir = r"audio"

audio2txt(audio_dir)
```

Then I got the following results:

You said: hello this is George welcome to my tutorial
You said: mathematics is important in daily life
You said: call me tomorrow
You said: do you want something to eat
You said: I want to speak with him
You said: nice to see you
You said: can you speak slowly
You said: have a good day

By the way, you can use my following python code to record your own audio and play with audio2txt function in Command-line python record.py "demo2.wav":

```python
import sys, getopt
```

(continues on next page)
import speech_recognition as sr

audio_filename = sys.argv[1]

r = sr.Recognizer()
with sr.Microphone() as source:
    r.adjust_for_ambient_noise(source)
    print("Hey there, say something, I am recording!")
    audio = r.listen(source)
    print("Done listening!")

with open(audio_filename, "wb") as f:
    f.write(audio.get_wav_data())

14.2 Text Preprocessing

- check to see if a row only contains whitespace

```python
def check_blanks(data_str):
    is_blank = str(data_str.isspace())
    return is_blank
```

- Determine whether the language of the text content is English or not: Use langid module to classify the language to make sure we are applying the correct cleanup actions for English

```python
def check_lang(data_str):
    predict_lang = langid.classify(data_str)
    if predict_lang[1] >= .9:
        language = predict_lang[0]
    else:
        language = 'NA'
    return language
```

- Remove features

```python
def remove_features(data_str):
    # compile regex
    url_re = re.compile('https?://(www.)?\w+\./\w+/(\w+)*/?')
    punc_re = re.compile('[%s]' % re.escape(string.punctuation))
    num_re = re.compile('(^\d+)')
    mention_re = re.compile('@[\w]+')
    alpha_num_re = re.compile("^[a-zA-Z0-9_.]+$")
    # convert to lowercase
    data_str = data_str.lower()
    # remove hyperlinks
    data_str = url_re.sub('', data_str)
    # remove @mentions
    data_str = mention_re.sub('', data_str)
    # remove punctuation
```
data_str = punc_re.sub(' ', data_str)
# remove numeric 'words'
data_str = num_re.sub(' ', data_str)
# remove non a-z 0-9 characters and words shorter than 3 characters
list_pos = 0
cleaned_str = ''
for word in data_str.split():
    if list_pos == 0:
        if alpha_num_re.match(word) and len(word) > 2:
            cleaned_str = word
        else:
            cleaned_str = ''
    else:
        if alpha_num_re.match(word) and len(word) > 2:
            cleaned_str = cleaned_str + ' ' + word
        else:
            cleaned_str += ' '
    list_pos += 1
return cleaned_str

• removes stop words

def remove_stops(data_str):
    # expects a string
    stops = set(stopwords.words("english"))
    list_pos = 0
    cleaned_str = ''
text = data_str.split()
for word in text:
    if word not in stops:
        # rebuild cleaned_str
        if list_pos == 0:
            cleaned_str = word
        else:
            cleaned_str = cleaned_str + ' ' + word
    list_pos += 1
return cleaned_str

• tagging text

def tag_and_remove(data_str):
    cleaned_str = ''
    # noun tags
    nn_tags = ['NN', 'NNP', 'NNPS', 'NNS']
    # adjectives
    jj_tags = ['JJ', 'JJR', 'JJS']
    # verbs
    vb_tags = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
    nltk_tags = nn_tags + jj_tags + vb_tags
    # break string into 'words'
text = data_str.split()

# tag the text and keep only those with the right tags
tagged_text = pos_tag(text)
for tagged_word in tagged_text:
    if tagged_word[1] in nltk_tags:
        cleaned_str += tagged_word[0] + ' '

return cleaned_str

• lemmatization

```python
def lemmatize(data_str):
    # expects a string
    list_pos = 0
    cleaned_str = ''
    lmtzr = WordNetLemmatizer()
text = data_str.split()
tagged_words = pos_tag(text)
for word in tagged_words:
    if 'v' in word[1].lower():
        lemma = lmtzr.lemmatize(word[0], pos='v')
    else:
        lemma = lmtzr.lemmatize(word[0], pos='n')
    if list_pos == 0:
        cleaned_str = lemma
    else:
        cleaned_str = cleaned_str + ' ' + lemma
    list_pos += 1
return cleaned_str
```

define the preprocessing function in PySpark

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
import preproc as pp

check_lang_udf = udf(pp.check_lang, StringType())
remove_stops_udf = udf(pp.remove_stops, StringType())
remove_features_udf = udf(pp.remove_features, StringType())
tag_and_remove_udf = udf(pp.tag_and_remove, StringType())
lemmatize_udf = udf(pp.lemmatize, StringType())
check_blanks_udf = udf(pp.check_blanks, StringType())
```

### 14.3 Text Classification

Theoretically speaking, you may apply any classification algorithms to do classification. I will only present Naive Bayes method is the following.
14.3.1 Introduction

14.3.2 Demo

1. create spark contexts

```python
import pyspark
from pyspark.sql import SQLContext

# create spark contexts
sc = pyspark.SparkContext()
sqlContext = SQLContext(sc)
```

2. load dataset

```python
# Load a text file and convert each line to a Row.
data_rdd = sc.textFile("../data/raw_data.txt")
parts_rdd = data_rdd.map(lambda l: l.split("\t"))

# Filter bad rows out
garantee_col_rdd = parts_rdd.filter(lambda l: len(l) == 3)
typed_rdd = guarantee_col_rdd.map(lambda p: (p[0], p[1], float(p[2])))

# Create DataFrame
data_df = sqlContext.createDataFrame(typed_rdd, ["text", "id", "label"])

# get the raw columns
raw_cols = data_df.columns

# data_df.show()
data_df.printSchema()
```

```
root
|-- text: string (nullable = true)
  |-- id: string (nullable = true)
  |-- label: double (nullable = true)
```
3. setup pyspark udf function

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
import preproc as pp

# Register all the functions in Preproc with Spark Context
check_lang_udf = udf(pp.check_lang, StringType())
remove_stops_udf = udf(pp.remove_stops, StringType())
remove_features_udf = udf(pp.remove_features, StringType())
tag_and_remove_udf = udf(pp.tag_and_remove, StringType())
lemmatize_udf = udf(pp.lemmatize, StringType())
check_blanks_udf = udf(pp.check_blanks, StringType())
```

4. language identification

```python
lang_df = data_df.withColumn("lang", check_lang_udf(data_df["text"]))
en_df = lang_df.filter(lang_df["lang"] == "en")
en_df.show(4)
```

5. remove stop words

```python
rm_stops_df = en_df.select(raw_cols)\n    .withColumn("stop_text", remove_stops_udf(en_df["text"]))
rm_stops_df.show(4)
```

(continues on next page)
6. remove irrelevant features

```python
rm_features_df = rm_stops_df.select(raw_cols+"["stop_text"]")
  .withColumn("feat_text", 
    remove_features_udf(rm_stops_df["stop_text"]))
rm_features_df.show(4)
```

7. tag the words

```python
tagged_df = rm_features_df.select(raw_cols+"["feat_text"]")
  .withColumn("tagged_text", 
    tag_and_remove_udf(rm_features_df.feat_text))
tagged_df.show(4)
```
8. lemmatization of words

```python
lemm_df = tagged_df.select(raw_cols+"tagged_text")
      .withColumn("lemm_text", lemmatize_udf(tagged_df["tagged_text"]))
lemm_df.show(4)
```

9. remove blank rows and drop duplicates

```python
check_blanks_df = lemm_df.select(raw_cols+"lemm_text")
      .withColumn("is_blank", check_blanks_udf(lemm_df["lemm_text"]))
# remove blanks
no_blanks_df = check_blanks_df.filter(check_blanks_df["is_blank"] == "False")
# drop duplicates
dedup_df = no_blanks_df.dropDuplicates(['text', 'label'])
dedup_df.show(4)
```
10. add unigue ID

```python
from pyspark.sql.functions import monotonically_increasing_id

# Create Unique ID
dedup_df = dedup_df.withColumn("uid", monotonically_increasing_id())
```

```
+--------------------+------------------+-----+--------------------+--------+
| text| id|label| lemm_text|is_blank|
+--------------------+------------------+-----+--------------------+--------+
| dragon| 1546813742| 1.0| dragon| False|
| hurt much| 1558492525| 1.0| hurt much| False|
| seth blog word se...| 383221484023709697| 1.0| seth blog word se...| False|
| teforia use machi...| 660668007975268352| 1.0| teforia use machi...| False|
```

11. create final dataset

```
data = dedup_df.select('uid','id', 'text','label')
data.show(4)
```

```
+------------+------------------+--------------------+-----+
| uid| id| text|label|
+------------+------------------+--------------------+-----+
| 85899345920| 1546813742| dragon| 1.0|
| 111669149696| 1558492525| hurt much| 1.0|
| 128849018880|383221484023709697| seth blog word se...| 1.0|
| 137438953472|660668007975268352| teforia use machi...| 1.0|
```

12. Create training and test sets

```
# Split the data into training and test sets (40% held out for testing)
(trainingData, testData) = data.randomSplit([0.6, 0.4])
```

13. NaiveBayes Pipeline
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
from pyspark.ml import Pipeline
from pyspark.ml.classification import NaiveBayes, RandomForestClassifier
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.feature import CountVectorizer

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and nb.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="rawFeatures")
# vectorizer = CountVectorizer(inputCol= "words", outputCol="rawFeatures")
idf = IDF(minDocFreq=3, inputCol="rawFeatures", outputCol="features")

# Naive Bayes model
nb = NaiveBayes()

# Pipeline Architecture
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, nb])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)

14. Make predictions
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("text", "label", "prediction").show(5,False)

+-----------------------------------------------+-----+----------+
|text |label|prediction|
+-----------------------------------------------+-----+----------+
|finish |1.0 |1.0 |
|meet rolo dogsofthinkgeek happy nationaldogday |1.0 |1.0 |
|pumpkin family |1.0 |1.0 |
|meet jet dogsofthinkgeek happy nationaldogday |1.0 |1.0 |
|meet vixie dogsofthinkgeek happy nationaldogday|1.0 |1.0 |
+-----------------------------------------------+-----+----------+
only showing top 5 rows

15. evaluation

from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
evaluator.evaluate(predictions)

14.3. Text Classification
14.4 Sentiment analysis

14.4.1 Introduction

Sentiment analysis (sometimes known as opinion mining or emotion AI) refers to the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. Sentiment analysis is widely applied to voice of the customer materials such as reviews and survey responses, online and social media, and healthcare materials for applications that range from marketing to customer service to clinical medicine.

Generally speaking, sentiment analysis aims to determine the attitude of a speaker, writer, or other subject with respect to some topic or the overall contextual polarity or emotional reaction to a document, interaction, or event. The attitude may be a judgment or evaluation (see appraisal theory), affective state (that is to say, the emotional state of the author or speaker), or the intended emotional communication (that is to say, the emotional effect intended by the author or interlocutor).

Sentiment analysis in business, also known as opinion mining is a process of identifying and cataloging a piece of text according to the tone conveyed by it. It has broad application:

- Sentiment Analysis in Business Intelligence Build up
- Sentiment Analysis in Business for Competitive Advantage
- Enhancing the Customer Experience through Sentiment Analysis in Business
14.4.2 Pipeline

14.4.3 Demo

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark Sentiment Analysis example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```python
df = spark.read.format('com.databricks.spark.csv').
    options(header='true', 
    inferSchema='true').
    load("../data/newtwitter.csv",header=True);
```

```
+--------------------+----------+-------+
| text | id | pubdate |
+--------------------+----------+-------+
|10 Things Missing...|2602860537| 18536|
|RT @ _NATURALBWINN...|2602850443| 18536|
|RT @ HBO24 yo the ...|2602761852| 18535|
|Aaaaaaaand I have...|2602738438| 18535|
|can I please have...|2602684185| 18535|
+--------------------+----------+-------+
only showing top 5 rows
```

3. Text Preprocessing

- remove non ASCII characters

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk import pos_tag
```

(continues on next page)
import string
import re

# remove non ASCII characters

def strip_non_ascii(data_str):
    ''' Returns the string without non ASCII characters'''
    stripped = (c for c in data_str if 0 < ord(c) < 127)
    return ''.join(stripped)

# setup pyspark udf function
strip_non_ascii_udf = udf(strip_non_ascii, StringType())

check:

df = df.withColumn('text_non_asci', strip_non_ascii_udf(df['text']))
df.show(5, True)

output:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>id</td>
<td>pubdate</td>
<td>text_non_asci</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
<td>-------</td>
<td>--------------------</td>
</tr>
<tr>
<td>10 Things Missing...</td>
<td>2602860537</td>
<td>18536</td>
<td>10 Things Missing...</td>
</tr>
<tr>
<td>RT @_NATURALBWINN...</td>
<td>2602850443</td>
<td>18536</td>
<td>RT @_NATURALBWINN...</td>
</tr>
<tr>
<td>RT @HBO24 yo the ...</td>
<td>2602761852</td>
<td>18535</td>
<td>RT @HBO24 yo the ...</td>
</tr>
<tr>
<td>Aaaaaaaaaand I have...</td>
<td>2602738438</td>
<td>18535</td>
<td>Aaaaaaaaaand I have...</td>
</tr>
<tr>
<td>can I please have...</td>
<td>2602684185</td>
<td>18535</td>
<td>can I please have...</td>
</tr>
</tbody>
</table>

only showing top 5 rows

• fixed abbreviation

# fixed abbreviation

def fix_abbreviation(data_str):
    data_str = data_str.lower()
    data_str = re.sub(r'\bthats\b', 'that is', data_str)
    data_str = re.sub(r'\bive\b', 'i have', data_str)
    data_str = re.sub(r'\bim\b', 'i am', data_str)
    data_str = re.sub(r'\bya\b', 'yeah', data_str)
    data_str = re.sub(r'\bcant\b', 'can not', data_str)
    data_str = re.sub(r'\bdont\b', 'do not', data_str)
    data_str = re.sub(r'\bwont\b', 'will not', data_str)
    data_str = re.sub(r'\bid\b', 'i would', data_str)
    data_str = re.sub(r'\bwtf\b', 'what the fuck', data_str)
    data_str = re.sub(r'\bwth\b', 'what the hell', data_str)
    data_str = re.sub(r'\bbr\b', 'are', data_str)
    data_str = re.sub(r'\bbu\b', 'you', data_str)
    data_str = re.sub(r'\bbk\b', 'ok', data_str)
    data_str = re.sub(r'\bsux\b', 'sucks', data_str)
    data_str = re.sub(r'\bno\b', 'no', data_str)
    data_str = re.sub(r'\bcoo\b', 'cool', data_str)
    data_str = re.sub(r'\brt\b', '', data_str)
data_str = data_str.strip()
return data_str

fix_abbreviation_udf = udf(fix_abbreviation, StringType())

check:

```python
df = df.withColumn('fixed_abbrev', fix_abbreviation_udf(df['text_non_ascii']))
df.show(5, True)
```

output:

```
+--------------------+----------+-------+--------------------+----------------+
 |                   |         |       |                   |               |
 |--------------------+----------+-------+--------------------+----------------|
 | 10 Things Missing...| 2602860537| 18536|10 Things Missing...|10 things missing...|
 | RT @_NATURALBWINN...| 2602850443| 18536| RT @_NATURALBWINN...| @ naturalbwinner ...
 | RT @HBO24 yo the ...| 2602761852| 18535| RT @HBO24 yo the ...| @hbo24 yo the #ne...
 | Aaaaaaaaand I have...| 2602738438| 18535| Aaaaaaaaand I have...| aaaaaaaand i have...
 | can I please have...| 2602684185| 18535| can I please have...| can i please have...

only showing top 5 rows
```

- remove irrelevant features

```python
def remove_features(data_str):
    # compile regex
    url_re = re.compile('https?://(www.)?\w+\./\w+(/\w+)*/?')
    punc_re = re.compile('[%s]' % re.escape(string.punctuation))
    num_re = re.compile('([\d]+)')
    mention_re = re.compile('@([\w]+)')
    alpha_num_re = re.compile('^[a-z0-9_.]+$')
    # convert to lowercase
    data_str = data_str.lower()
    # remove hyperlinks
    data_str = url_re.sub('', data_str)
    # remove @mentions
    data_str = mention_re.sub('', data_str)
    # remove punctuation
    data_str = punc_re.sub('', data_str)
    # remove numeric 'words'
```

(continues on next page)
data_str = num_re.sub(' ', data_str)
# remove non a-z 0-9 characters and words shorter than 1 characters
list_pos = 0
cleaned_str = ''
for word in data_str.split():
    if list_pos == 0:
        if alpha_num_re.match(word) and len(word) > 1:
            cleaned_str = word
        else:
            cleaned_str = ' '
    else:
        if alpha_num_re.match(word) and len(word) > 1:
            cleaned_str = cleaned_str + ' ' + word
        else:
            cleaned_str += ' '
    list_pos += 1
# remove unwanted space, *.split() will automatically split on
# whitespace and discard duplicates, the " ".join() joins the
# resulting list into one string.
return " ".join(cleaned_str.split())

# setup pyspark udf function
remove_features_udf = udf(remove_features, StringType())

check:

def = df.withColumn('removed', remove_features_udf(df['fixed_abbrev']))
def.show(5, True)

output:

<table>
<thead>
<tr>
<th>text</th>
<th>id</th>
<th>pubdate</th>
<th>text_non_ascii</th>
<th>fixed_abbrev</th>
<th>removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Things Missing...</td>
<td>260260537</td>
<td>18536</td>
<td>10 Things Missing...</td>
<td>10 things missing...</td>
<td>10 things missing...</td>
</tr>
<tr>
<td>RT @NATURALBWINN...</td>
<td>260285043</td>
<td>18536</td>
<td>RT @NATURALBWINN...</td>
<td>@naturalbwinner...</td>
<td>oh and do not lik...</td>
</tr>
<tr>
<td>RT @HBO24 yo the...</td>
<td>2602761852</td>
<td>18535</td>
<td>RT @HBO24 yo the...</td>
<td>@hbo24 yo the #ne.../yo the newtwitter...</td>
<td></td>
</tr>
<tr>
<td>Aaaaaaand I have...</td>
<td>2602738438</td>
<td>18535</td>
<td>Aaaaaaand I have...</td>
<td>aaaaaaand i have.../aaaaaaand have t...</td>
<td></td>
</tr>
<tr>
<td>can I please have...</td>
<td>2602684185</td>
<td>18535</td>
<td>can I please have...</td>
<td>can i please have.../can please have t...</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>---------</td>
<td>----------------</td>
<td>--------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
only showing top 5 rows

4. Sentiment Analysis main function
from pyspark.sql.types import FloatType

from textblob import TextBlob

def sentiment_analysis(text):
    return TextBlob(text).sentiment.polarity

sentiment_analysis_udf = udf(sentiment_analysis, FloatType())

df = df.withColumn("sentiment_score", sentiment_analysis_udf(df['removed']))
df.show(5, True)

• Sentiment score

<table>
<thead>
<tr>
<th>removed</th>
<th>sentiment_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>things missing in...</td>
<td>-0.03181818</td>
</tr>
<tr>
<td>oh and do not lik...</td>
<td>-0.03181818</td>
</tr>
<tr>
<td>yo the newtwitter...</td>
<td>0.3181818</td>
</tr>
<tr>
<td>aaaaaaaaand have t...</td>
<td>0.11818182</td>
</tr>
<tr>
<td>can please have t...</td>
<td>0.13636364</td>
</tr>
</tbody>
</table>

only showing top 5 rows

• Words frequency

• Sentiment Classification

```python
def condition(r):
    if r >= 0.1:
        # (continues on next page)
```
5. Output

- Sentiment Class

<table>
<thead>
<tr>
<th>text</th>
<th>sentiment_score</th>
<th>sentiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and this #newtwitter...</td>
<td>1.0</td>
<td>positive</td>
</tr>
<tr>
<td>&quot;RT @SarahsJokes:...</td>
<td>1.0</td>
<td>positive</td>
</tr>
<tr>
<td>#newtwitter using...</td>
<td>1.0</td>
<td>positive</td>
</tr>
<tr>
<td>The #NewTwitter h...</td>
<td>1.0</td>
<td>positive</td>
</tr>
<tr>
<td>You can now undo ...</td>
<td>1.0</td>
<td>positive</td>
</tr>
</tbody>
</table>

only showing top 5 rows

<table>
<thead>
<tr>
<th>text</th>
<th>sentiment_score</th>
<th>sentiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists on #NewTwitter...</td>
<td>-0.1</td>
<td>neutral</td>
</tr>
<tr>
<td>Too bad most of m...</td>
<td>-0.1</td>
<td>neutral</td>
</tr>
<tr>
<td>the #newtwitter i...</td>
<td>-0.1</td>
<td>neutral</td>
</tr>
</tbody>
</table>

(continues on next page)
14.5 N-grams and Correlations

14.6 Topic Model: Latent Dirichlet Allocation

14.6.1 Introduction

In text mining, a topic model is a unsupervised model for discovering the abstract “topics” that occur in a collection of documents.

Latent Dirichlet Allocation (LDA) is a mathematical method for estimating both of these at the same time: finding the mixture of words that is associated with each topic, while also determining the mixture of topics that describes each document.
14.6.2 Demo

1. Load data

```python
rawdata = spark.read.load("../data/airlines.csv", format="csv", header=True)
rawdata.show(5)
```

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
<th>recommended</th>
<th>review</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Delta Air Lines</td>
<td>21-Jun-14</td>
<td>Thailand</td>
<td>7</td>
<td>Economy</td>
<td>4</td>
<td>YES</td>
<td>Flew Mar 30 NRT t...</td>
</tr>
<tr>
<td>10002</td>
<td>Delta Air Lines</td>
<td>19-Jun-14</td>
<td>USA</td>
<td>0</td>
<td>Economy</td>
<td>2</td>
<td>NO</td>
<td>Flight 2463 leave...</td>
</tr>
<tr>
<td>10003</td>
<td>Delta Air Lines</td>
<td>18-Jun-14</td>
<td>USA</td>
<td>0</td>
<td>Economy</td>
<td>1</td>
<td>NO</td>
<td>Delta Website fro...</td>
</tr>
<tr>
<td>10004</td>
<td>Delta Air Lines</td>
<td>17-Jun-14</td>
<td>USA</td>
<td>9</td>
<td>Business</td>
<td>4</td>
<td>YES</td>
<td>&quot;I just returned ...</td>
</tr>
<tr>
<td>10005</td>
<td>Delta Air Lines</td>
<td>17-Jun-14</td>
<td>Ecuador</td>
<td>7</td>
<td>Economy</td>
<td>3</td>
<td>YES</td>
<td>&quot;Round-trip fligh...</td>
</tr>
</tbody>
</table>

only showing top 5 rows

1. Text preprocessing

I will use the following raw column names to keep my table concise:

```python
raw_cols = rawdata.columns
raw_cols
```

```
['id', 'airline', 'date', 'location', 'rating', 'cabin', 'value', 'recommended', 'review']
```

```python
rawdata = rawdata.dropDuplicates(['review'])
```

```python
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, DoubleType, DateType

from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk import pos_tag
import langid
import string
import re
```

- remove non ASCII characters
# remove non ASCII characters

```python
def strip_non_ascii(data_str):
    ''' Returns the string without non ASCII characters'''
    stripped = (c for c in data_str if 0 < ord(c) < 127)
    return ''.join(stripped)
```

- check it blank line or not

```python
# check to see if a row only contains whitespace
def check_blanks(data_str):
    is_blank = str(data_str.isspace())
    return is_blank
```

- check the language (a little bit slow, I skited this step)

```python
# check the language (only apply to english)
def check_lang(data_str):
    from langid.langid import LanguageIdentifier, model
    identifier = LanguageIdentifier.from_modelstring(model, norm_probs=True)
    predict_lang = identifier.classify(data_str)

    if predict_lang[1] >= .9:
        language = predict_lang[0]
    else:
        language = predict_lang[0]
    return language
```

- fixed abbreviation

```python
# fixed abbreviation
def fix_abbreviation(data_str):
    data_str = data_str.lower()
    data_str = re.sub(r'\bthats\b', 'that is', data_str)
    data_str = re.sub(r'\bhive\b', 'i have', data_str)
    data_str = re.sub(r'\bim\b', 'i am', data_str)
    data_str = re.sub(r'\bya\b', 'yeah', data_str)
    data_str = re.sub(r'\bcant\b', 'can not', data_str)
    data_str = re.sub(r'\bdont\b', 'do not', data_str)
    data_str = re.sub(r'\bwont\b', 'will not', data_str)
    data_str = re.sub(r'\bid\b', 'i would', data_str)
    data_str = re.sub(r'wtf', 'what the fuck', data_str)
    data_str = re.sub(r'\bwth\b', 'what the hell', data_str)
    data_str = re.sub(r'\br\b', 'OK', data_str)
    data_str = re.sub(r'\bsux\b', 'sucks', data_str)
    data_str = re.sub(r'\bno+\b', 'no', data_str)
    data_str = re.sub(r'\bcool\b', 'cool', data_str)
    data_str = re.sub(r'\r\n\b', ' ', data_str)
    data_str = data_str.strip()
    return data_str
```
• remove irrelevant features

```python
# remove irrelevant features

def remove_features(data_str):
    # compile regex
    url_re = re.compile('https?://(www.)?\w+.(\.\w+)/?')
    punc_re = re.compile('[' + re.escape(string.punctuation) + ']')
    num_re = re.compile('((' + re.escape(string.digits) + ')>')
    mention_re = re.compile('@\w+')
    alpha_num_re = re.compile('^[a-z0-9_.]+$')
    # convert to lowercase
    data_str = data_str.lower()
    # remove hyperlinks
    data_str = url_re.sub(' ', data_str)
    # remove @mentions
    data_str = mention_re.sub(' ', data_str)
    # remove punctuation
    data_str = punc_re.sub(' ', data_str)
    # remove numeric 'words'
    data_str = num_re.sub(' ', data_str)
    # remove non a-z 0-9 characters and words shorter than 1 character
    list_pos = 0
    cleaned_str = ''
    for word in data_str.split():
        if list_pos == 0:
            if alpha_num_re.match(word) and len(word) > 1:
                cleaned_str = word
            else:
                cleaned_str = ''
        else:
            if alpha_num_re.match(word) and len(word) > 1:
                cleaned_str = cleaned_str + ' ' + word
            else:
                cleaned_str += ' ' 
        list_pos += 1
    return ' '.join(cleaned_str.split())
```

• removes stop words

```python
# removes stop words

def remove_stops(data_str):
    # expects a string
    stops = set(stopwords.words("english"))
    list_pos = 0
    cleaned_str = ''
    text = data_str.split()
    for word in text:
        if word not in stops:
            # rebuild cleaned_str
            cleaned_str += word
    return cleaned_str
```

(continues on next page)
if list_pos == 0:
cleaned_str = word
else:
cleaned_str = cleaned_str + ' ' + word
list_pos += 1
return cleaned_str

• Part-of-Speech Tagging

# Part-of-Speech Tagging
def tag_and_remove(data_str):
cleaned_str = ''
# noun tags
nn_tags = ['NN', 'NNP', 'NNPS', 'NNS']
# adjectives
jj_tags = ['JJ', 'JJR', 'JJS']
# verbs
vb_tags = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
nltk_tags = nn_tags + jj_tags + vb_tags

# break string into 'words'
text = data_str.split()

# tag the text and keep only those with the right tags
tagged_text = pos_tag(text)
for tagged_word in tagged_text:
    if tagged_word[1] in nltk_tags:
cleaned_str += tagged_word[0] + ' ' 
return cleaned_str

• lemmatization

# lemmatization
def lemmatize(data_str):
    # expects a string
list_pos = 0
cleaned_str = ''
lmtzr = WordNetLemmatizer()
text = data_str.split()
tagged_words = pos_tag(text)
for word in tagged_words:
    if 'v' in word[1].lower():
        lemma = lmtzr.lemmatize(word[0], pos='v')
    else:
        lemma = lmtzr.lemmatize(word[0], pos='n')
    if list_pos == 0:
cleaned_str = lemma
else:
cleaned_str = cleaned_str + ' ' + lemma
list_pos += 1
return cleaned_str
• setup pyspark udf function

```python
# setup pyspark udf function
strip_non_ascii_udf = udf(strip_non_ascii, StringType())
check_blanks_udf = udf(check_blanks, StringType())
check_lang_udf = udf(check_lang, StringType())
fix_abbreviation_udf = udf(fix_abbreviation, StringType())
remove_stops_udf = udf(remove_stops, StringType())
remove_features_udf = udf(remove_features, StringType())
tag_and_remove_udf = udf(tag_and_remove, StringType())
lemmatize_udf = udf(lemmatize, StringType())
```

1. Text processing

• correct the data schema

```python
rawdata = rawdata.withColumn('rating', rawdata.rating.cast('float'))
```

```python
rawdata.printSchema()
```

```plaintext
root
|-- id: string (nullable = true)
|-- airline: string (nullable = true)
|-- date: string (nullable = true)
|-- location: string (nullable = true)
|-- rating: float (nullable = true)
|-- cabin: string (nullable = true)
|-- value: string (nullable = true)
|-- recommended: string (nullable = true)
|-- review: string (nullable = true)
```

```python
from datetime import datetime
from pyspark.sql.functions import col
# https://docs.python.org/2/library/datetime.html#strftime-and-
˓→strptime-behavior
# 21-Jun-14 <----> %d-%b-%y
to_date = udf (lambda x: datetime.strptime(x, '%d-%b-%y'),
˓→DateType())
rawdata = rawdata.withColumn('date', to_date(col('date')))

rawdata.printSchema()
```

```plaintext
root
|-- id: string (nullable = true)
|-- airline: string (nullable = true)
|-- date: date (nullable = true)
|-- location: string (nullable = true)
|-- rating: float (nullable = true)
|-- cabin: string (nullable = true)
```

(continues on next page)
rawdata.show(5)

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
<th>recommended</th>
<th>review</th>
<th>non_asci</th>
</tr>
</thead>
<tbody>
<tr>
<td>10551</td>
<td>Southwest Airlines</td>
<td>2013-11-06</td>
<td>USA</td>
<td>1.0</td>
<td>Business</td>
<td>2</td>
<td>Flight 3246 from ...</td>
<td>Flight 3246 from ...</td>
<td>Flight 3246 from ...</td>
</tr>
<tr>
<td>10298</td>
<td>US Airways</td>
<td>2014-03-31</td>
<td>UK</td>
<td>1.0</td>
<td>Business</td>
<td>0</td>
<td>Flight from Manch</td>
<td>Flight from Manch</td>
<td>Flight from Manch</td>
</tr>
<tr>
<td>10564</td>
<td>Southwest Airlines</td>
<td>2013-09-06</td>
<td>USA</td>
<td>10.0</td>
<td>Economy</td>
<td>5</td>
<td>YES I'm Executive Pla...</td>
<td>YES I'm Executive Pla...</td>
<td>YES I'm Executive Pla...</td>
</tr>
<tr>
<td>10134</td>
<td>Delta Air Lines</td>
<td>2013-12-10</td>
<td>USA</td>
<td>8.0</td>
<td>Economy</td>
<td>4</td>
<td>YES MSP-JFK-MXP and r...</td>
<td>YES MSP-JFK-MXP and r...</td>
<td>YES MSP-JFK-MXP and r...</td>
</tr>
<tr>
<td>10912</td>
<td>United Airlines</td>
<td>2014-04-07</td>
<td>USA</td>
<td>3.0</td>
<td>Economy</td>
<td>1</td>
<td>NO Worst airline I h...</td>
<td>NO Worst airline I h...</td>
<td>NO Worst airline I h...</td>
</tr>
</tbody>
</table>

only showing top 5 rows

rawdata = rawdata.withColumn('non_asci', strip_non_ascii_udf(rawdata['review']))

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
<th>recommended</th>
<th>review</th>
<th>non_asci</th>
</tr>
</thead>
<tbody>
<tr>
<td>10551</td>
<td>Southwest Airlines</td>
<td>2013-11-06</td>
<td>USA</td>
<td>1.0</td>
<td>Business</td>
<td>2</td>
<td>Flight 3246 from ...</td>
<td>Flight 3246 from ...</td>
<td>Flight 3246 from ...</td>
</tr>
<tr>
<td>10298</td>
<td>US Airways</td>
<td>2014-03-31</td>
<td>UK</td>
<td>1.0</td>
<td>Business</td>
<td>0</td>
<td>Flight from Manch</td>
<td>Flight from Manch</td>
<td>Flight from Manch</td>
</tr>
<tr>
<td>10564</td>
<td>Southwest Airlines</td>
<td>2013-09-06</td>
<td>USA</td>
<td>10.0</td>
<td>Economy</td>
<td>5</td>
<td>YES I'm Executive Pla...</td>
<td>YES I'm Executive Pla...</td>
<td>YES I'm Executive Pla...</td>
</tr>
<tr>
<td>10134</td>
<td>Delta Air Lines</td>
<td>2013-12-10</td>
<td>USA</td>
<td>8.0</td>
<td>Economy</td>
<td>4</td>
<td>YES MSP-JFK-MXP and r...</td>
<td>YES MSP-JFK-MXP and r...</td>
<td>YES MSP-JFK-MXP and r...</td>
</tr>
<tr>
<td>10912</td>
<td>United Airlines</td>
<td>2014-04-07</td>
<td>USA</td>
<td>3.0</td>
<td>Economy</td>
<td>1</td>
<td>NO Worst airline I h...</td>
<td>NO Worst airline I h...</td>
<td>NO Worst airline I h...</td>
</tr>
</tbody>
</table>

only showing top 5 rows
```python
rawdata = rawdata.select(raw_cols+[\'non_asci\'])
    .withColumn(\'fixed_abbrev\', fix_abbreviation_udf(rawdata[\'non_asci\']))

+--------+------------------+----------+--------+------+--------+-----+
<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
</tr>
</thead>
</table>
+--------+------------------+----------+---------+-------+--------+-------|
| 10551  | Southwest Airlines| 2013-11-06| USA     | 1.0   | Business| 2     |
|        | NO| Flight 3246 from ...| Flight 3246 from ...| flight 3246 | 
|        | from ...| 
| 10298  | US Airways| 2014-03-31| UK      | 1.0   | Business| 0     |
|        | NO| Flight from Manch...| Flight from Manch...| flight from... | 
|        | from manchester...| (continues on next page)

only showing top 5 rows
```

```python
rawdata = rawdata.select(raw_cols+[\'fixed_abbrev\'])
    .withColumn(\'stop_text\', remove_stops_udf(rawdata[\'fixed_abbrev\']))

+--------+------------------+----------+--------+-------+--------+-----+
<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>recommended</th>
<th>review</th>
</tr>
</thead>
</table>
+--------+------------------+----------+---------+-------+-------------+--------|
| 10551  | Southwest Airlines| 2013-11-06| USA     | 1.0   | Business| 2     |
|        | NO| Flight 3246 from ...| flight 3246 | 
|        | from ...| 
| 10298  | US Airways| 2014-03-31| UK      | 1.0   | Business| 0     |
|        | NO| Flight from Manch...| flight from manch... | 
|        | from manchester...| 
```

(continues on next page)
Learning Apache Spark with Python

(continued from previous page)

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
<th>recommended</th>
<th>review</th>
<th>feat_text</th>
</tr>
</thead>
<tbody>
<tr>
<td>10564</td>
<td>Southwest Airlines</td>
<td>2013-09-06</td>
<td>USA</td>
<td>10.0</td>
<td>Economy</td>
<td>5</td>
<td>YES</td>
<td>I'm Executive Pla...</td>
<td>i'm executive pl...</td>
</tr>
<tr>
<td>10134</td>
<td>Delta Air Lines</td>
<td>2013-12-10</td>
<td>USA</td>
<td>8.0</td>
<td>Economy</td>
<td>4</td>
<td>YES</td>
<td>MSP-JFK-MXP and r...</td>
<td>msp-jfk-mxp and r...</td>
</tr>
<tr>
<td>10912</td>
<td>United Airlines</td>
<td>2014-04-07</td>
<td>USA</td>
<td>3.0</td>
<td>Economy</td>
<td>1</td>
<td>NO</td>
<td>Worst airline I h...</td>
<td>worst airline i h...</td>
</tr>
</tbody>
</table>

---

only showing top 5 rows

rawdata = rawdata.select(raw_cols+['stop_text'])
.withColumn('feat_text',remove_features_udf(rawdata['stop_text']))

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>location</th>
<th>rating</th>
<th>cabin</th>
<th>value</th>
<th>recommended</th>
<th>review</th>
<th>feat_text</th>
</tr>
</thead>
<tbody>
<tr>
<td>10551</td>
<td>Southwest Airlines</td>
<td>2013-11-06</td>
<td>USA</td>
<td>1.0</td>
<td>Business</td>
<td>2</td>
<td>NO</td>
<td>Flight 3246 from ...</td>
<td>flight 3246 from ...</td>
</tr>
<tr>
<td>10298</td>
<td>US Airways</td>
<td>2014-03-31</td>
<td>UK</td>
<td>1.0</td>
<td>Business</td>
<td>0</td>
<td>NO</td>
<td>Flight from Manch...</td>
<td>flight from manch...</td>
</tr>
<tr>
<td>10564</td>
<td>Southwest Airlines</td>
<td>2013-09-06</td>
<td>USA</td>
<td>10.0</td>
<td>Economy</td>
<td>5</td>
<td>YES</td>
<td>I'm Executive Pla...</td>
<td>i'm executive pl...</td>
</tr>
<tr>
<td>10134</td>
<td>Delta Air Lines</td>
<td>2013-12-10</td>
<td>USA</td>
<td>8.0</td>
<td>Economy</td>
<td>4</td>
<td>YES</td>
<td>MSP-JFK-MXP and r...</td>
<td>msp-jfk-mxp and r...</td>
</tr>
<tr>
<td>10912</td>
<td>United Airlines</td>
<td>2014-04-07</td>
<td>USA</td>
<td>3.0</td>
<td>Economy</td>
<td>1</td>
<td>NO</td>
<td>Worst airline I h...</td>
<td>worst airline i h...</td>
</tr>
</tbody>
</table>

---

only showing top 5 rows

rawdata = rawdata.select(raw_cols+['feat_text'])
.withColumn('tagged_text',tag_and_remove_udf(rawdata['feat_text']))

(continues on next page)
rawdata = rawdata.withColumn('lemm_text', lemmatize_udf(rawdata['tagged_text']))

rawdata = rawdata.select(raw_cols+['tagged_text']) \
   .withColumn('lemm_text', lemmatize_udf(rawdata['tagged_text']))
Learning Apache Spark with Python

(continued from previous page)

```python
rawdata = rawdata.withColumn("is_blank", check_blanks_udf(rawdata[
"lemm_text"]))
```

```
+-----+------------------+----------+--------+------+--------+-----+-
| id  | airline          | date     | location| rating| cabin  | value| recommended| review| lemm_|
+-----+------------------+----------+--------+-------+-------+------+-----------+-------+-----+----------+--------+--------+--------+-------+--------+-----+|
|     |                  |          |         |       |       |      |            |       |      |          |        |        |        |       |        |      |
|10551|Southwest Airlines|2013-11-06| USA| 1.0|Business| 2 | NO|Flight 3246 from ...|flight chicago mi...| False| 0 |
|10298|US Airways|2014-03-31| UK| 1.0|Business| 0 | NO|Flight from Manch...|flight manchester...| False| 0 |
|10564|Southwest Airlines|2013-09-06| USA| 10.0|Economy| 5 | YES|I'm Executive Pla...|executive platiniu...| False| 0 |
|10134|Delta Air Lines|2013-12-10| USA| 8.0|Economy| 4 | YES|MSP-JFK-MXP and r...|msp jfk mxp retu...| False| 0 |
|10912|United Airlines|2014-04-07| USA| 3.0|Economy| 1 | NO|Worst airline I h...|worst airline ual...| False| 0 |
```

(continues on next page)

Learning Apache Spark with Python

(continued from previous page)

| 10298 | US Airways | 2014-03-31 | UK | 1.0 | Business | 0 |
| 10564 | Southwest Airlines | 2013-09-06 | USA | 10.0 | Economy | 5 |
| 10134 | Delta Air Lines | 2013-12-10 | USA | 8.0 | Economy | 2 |
| 10912 | United Airlines | 2014-04-07 | USA | 3.0 | Economy | 1 |

only showing top 5 rows

# Pipeline for LDA model

```python
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
from pyspark.ml import Pipeline
from pyspark.ml.classification import NaiveBayes,
from pyspark.ml.clustering import LDA
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator
from pyspark.ml.feature import IndexToString, StringIndexer,
from pyspark.ml.feature import CountVectorizer

# Configure an ML pipeline, which consists of tree stages: tokenizer, hashingTF, and nb.
tokenizer = Tokenizer(inputCol="lemm_text", outputCol="words")
data = tokenizer.transform(data)
vectorizer = CountVectorizer(inputCol= "words", outputCol= "rawFeatures")
idf = IDF(inputCol="rawFeatures", outputCol="features")
#idfModel = idf.fit(data)
lda = LDA(k=20, seed=1, optimizer="em")

pipeline = Pipeline(stages=[tokenizer, vectorizer, idf, lda])

model = pipeline.fit.fit(data)
```

1. Results presentation

- Topics

<table>
<thead>
<tr>
<th>topic</th>
<th>termIndices</th>
<th>termWeights</th>
</tr>
</thead>
</table>
| 0     | [60, 7, 12, 483, ...] | [0.01349507958269... |}
| 1     | [363, 29, 187, 55... | [0.0124725014447... |}

(continues on next page)

- Topic terms

```python
from pyspark.sql.types import ArrayType, StringType

def termsIdx2Term(vocabulary):
    def termsIdx2Term(termIndices):
        return [vocabulary[int(index)] for index in termIndices]
    return udf(termsIdx2Term, ArrayType(StringType()))

vectorizerModel = model.stages[1]
vocabList = vectorizerModel.vocabulary
final = ldatopics.withColumn("Terms", termsIdx2Term(vocabList)("termIndices"))
```

<table>
<thead>
<tr>
<th>topic</th>
<th>termIndices</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[60, 7, 12, 483, 292, 326, 88, 4, 808, 32]</td>
<td>[pm, plane, board, kid, online, lga, schedule, get, memphis, arrive]</td>
</tr>
<tr>
<td>1</td>
<td>[363, 29, 187, 55, 48, 647, 30, 9, 204, 457]</td>
<td>[dublin, class, th, sit, entertainment, express, say, delay, dl, son]</td>
</tr>
<tr>
<td>2</td>
<td>[46, 107, 672, 274, 92, 539, 23, 27, 279, 8]</td>
<td>[economy, sfo, milwaukee, decent, comfortable, iad, return, united, average, airline]</td>
</tr>
</tbody>
</table>

(continues on next page)
Learning Apache Spark with Python

(continued from previous page)

| 3 | [76, 43, 285, 152, 102, 34, 300, 113, 24, 31] | [didn, pay, lose, different, extra, bag, mile, baggage, leave, day] |
|   | [201, 13, 372, 692, 248, 62, 211, 187, 105, 110] | [houston, crew, heathrow, louisville, london, great, denver, th, land, jfk] |
| 5 | [122, 103, 181, 48, 434, 10, 121, 147, 934, 169] | [lhr, serve, screen, entertainment, ny, delta, excellent, atl, sin, newark] |
| 6 | [14, 270, 18, 74, 70, 37, 16, 450, 3, 20] | [check, employee, gate, line, change, wait, take, fill, time, tell] |
| 7 | [111, 36, 341, 10, 320, 528, 844, 19, 195, 524] | [atlanta, first, toilet, delta, washington, card, global, staff, route, amsterdam] |
| 8 | [477, 266, 297, 185, 1, 33, 22, 783, 17, 908] | [fuel, group, pas, boarding, seat, trip, minute, orleans, make, select] |
| 9 | [10, 73, 46, 1, 248, 302, 213, 659, 48, 228] | [delta, lax, economy, seat, london, detroit, comfo, weren, entertainment, wife] |
| 10 | [57, 29, 411, 10, 221, 121, 661, 19, 805, 733] | [business, class, fra, delta, lounge, excellent, syd, staff, nov, mexico] |
| 11 | [293, 119, 385, 481, 503, 69, 13, 87, 176, 545] | [march, ua, manchester, phx, envoy, drink, crew, american, aa, canada] |
| 12 | [116, 218, 256, 156, 639, 20, 365, 18, 22, 136] | [san, cht, francisco, second, text, tell, captain, gate, minute, available] |
| 13 | [433, 171, 176, 339, 429, 575, 10, 26, 474, 796] | [daughter, small, aa, ba, segment, proceed, delta, passenger, size, similar] |
| 14 | [74, 84, 45, 108, 342, 111, 315, 87, 52, 4] | [line, agent, next, hotel, standby, atlanta, dallas, american, book, get] |
| 15 | [669, 215, 14, 58, 561, 59, 125, 179, 93, 5] | [fit, carry, check, people, bathroom, ask, thing, row, don, fly] |
| 16 | [198, 21, 98, 164, 57, 141, 345, 62, 121, 174] | [ife, good, nice, much, business, lot, dfw, great, excellent, carrier] |
| 17 | [96, 29, 569, 444, 15, 568, 21, 103, 657, 505] | [phl, class, diego, lady, food, wheelchair, good, serve, miami, mia] |
| 18 | [18, 60, 140, 64, 47, 40, 31, 35, 2, 123] | [gate, pm, phoenix, connection, cancel, connect, day, airpo, hour, charlotte] |
| 19 | [33, 178, 95, 2, 9, 284, 42, 4, 89, 31] | [trip, counter, philadelphia, hour, delay, stay, way, get, southwest, day] |
- **LDA results**

<table>
<thead>
<tr>
<th>id</th>
<th>airline</th>
<th>date</th>
<th>cabin</th>
<th>rating</th>
<th>words</th>
<th>features</th>
<th>topicDistribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>10551</td>
<td>Southwest Airlines</td>
<td>2013-11-06</td>
<td>Business</td>
<td>1.0</td>
<td>flight, chicago,...</td>
<td>(4695,[0,2,3,6,11,...]</td>
<td>0.03640342580508...</td>
</tr>
<tr>
<td>10298</td>
<td>US Airways</td>
<td>2014-03-31</td>
<td>Business</td>
<td>1.0</td>
<td>flight, manchest,...</td>
<td>(4695,[0,1,2,6,7,...]</td>
<td>0.01381306271470...</td>
</tr>
<tr>
<td>10564</td>
<td>Southwest Airlines</td>
<td>2013-09-06</td>
<td>Economy</td>
<td>10.0</td>
<td>executive, plati,...</td>
<td>(4695,[0,1,6,7,11,11]</td>
<td>0.05063554352934...</td>
</tr>
<tr>
<td>10134</td>
<td>Delta Air Lines</td>
<td>2013-12-10</td>
<td>Economy</td>
<td>8.0</td>
<td>msp, jfk, mxp, r,...</td>
<td>(4695,[0,1,3,10,1,...]</td>
<td>0.01494708959842...</td>
</tr>
<tr>
<td>10912</td>
<td>United Airlines</td>
<td>2014-04-07</td>
<td>Economy</td>
<td>3.0</td>
<td>worst, airline,...</td>
<td>(4695,[0,1,7,8,13,...]</td>
<td>0.04421751181232...</td>
</tr>
<tr>
<td>10089</td>
<td>Delta Air Lines</td>
<td>2014-02-18</td>
<td>Economy</td>
<td>2.0</td>
<td>dl, mia, lax, im,...</td>
<td>(4695,[2,4,5,7,8,...]</td>
<td>0.02158861273876...</td>
</tr>
<tr>
<td>10385</td>
<td>US Airways</td>
<td>2013-10-21</td>
<td>Economy</td>
<td>10.0</td>
<td>flew, gla, phl,...</td>
<td>(4695,[0,1,3,5,14,...]</td>
<td>0.03343845991816...</td>
</tr>
<tr>
<td>10249</td>
<td>US Airways</td>
<td>2014-06-17</td>
<td>Economy</td>
<td>1.0</td>
<td>friend, book, fl,...</td>
<td>(4695,[0,2,3,4,5,...]</td>
<td>0.02362432562165...</td>
</tr>
<tr>
<td>10289</td>
<td>US Airways</td>
<td>2014-04-12</td>
<td>Economy</td>
<td>10.0</td>
<td>flew, air, rome,...</td>
<td>(4695,[0,1,5,8,13,...]</td>
<td>0.01664012816210...</td>
</tr>
<tr>
<td>10654</td>
<td>Southwest Airlines</td>
<td>2012-07-10</td>
<td>Economy</td>
<td>8.0</td>
<td>lhr, jfk, think,...</td>
<td>(4695,[0,4,5,6,8,...]</td>
<td>0.01526072330297...</td>
</tr>
<tr>
<td>10754</td>
<td>American Airlines</td>
<td>2014-05-04</td>
<td>Economy</td>
<td>10.0</td>
<td>san, diego, moli,...</td>
<td>(4695,[0,2,8,15,2,...]</td>
<td>0.03571177612496...</td>
</tr>
<tr>
<td>10646</td>
<td>Southwest Airlines</td>
<td>2012-08-17</td>
<td>Economy</td>
<td>7.0</td>
<td>toledo, co, stop,...</td>
<td>(4695,[0,2,3,4,7,...]</td>
<td>0.02394775146271...</td>
</tr>
<tr>
<td>10097</td>
<td>Delta Air Lines</td>
<td>2014-02-03</td>
<td>First Class</td>
<td>10.0</td>
<td>honolulu, la, fi,...</td>
<td>(4695,[0,4,6,7,13,...]</td>
<td>0.02008375619661...</td>
</tr>
<tr>
<td>10132</td>
<td>Delta Air Lines</td>
<td>2013-12-16</td>
<td>Economy</td>
<td>7.0</td>
<td>manchester, uk,...</td>
<td>(4695,[0,1,2,3,5,...]</td>
<td>0.01463126146601...</td>
</tr>
<tr>
<td>10560</td>
<td>Southwest Airlines</td>
<td>2013-09-20</td>
<td>Economy</td>
<td>9.0</td>
<td>first, time, sou,...</td>
<td>(4695,[0,3,7,8,9,...]</td>
<td>0.04934836469896...</td>
</tr>
<tr>
<td>10579</td>
<td>Southwest Airlines</td>
<td>2013-07-25</td>
<td>Economy</td>
<td>0.0</td>
<td>plane, land, pm,...</td>
<td>(4695,[2,3,4,5,7,...]</td>
<td>0.0610695241722...</td>
</tr>
<tr>
<td>10425</td>
<td>US Airways</td>
<td>2013-08-06</td>
<td>Economy</td>
<td>3.0</td>
<td>airway, bad, pro,...</td>
<td>(4695,[2,3,4,7,8,...]</td>
<td>0.017704717322...</td>
</tr>
<tr>
<td>10650</td>
<td>Southwest Airlines</td>
<td>2012-07-27</td>
<td>Economy</td>
<td>9.0</td>
<td>flew, jfk, lhr,...</td>
<td>(4695,[0,1,6,13,1,...]</td>
<td>0.02676226245086...</td>
</tr>
<tr>
<td>10260</td>
<td>US Airways</td>
<td>2014-06-03</td>
<td>Economy</td>
<td>1.0</td>
<td>february, air, u,...</td>
<td>(4695,[0,2,4,17,2,...]</td>
<td>0.02887390875079...</td>
</tr>
<tr>
<td>10202</td>
<td>Delta Air Lines</td>
<td>2013-09-14</td>
<td>Economy</td>
<td>10.0</td>
<td>aug, lhr, jfk, b,...</td>
<td>(4695,[1,2,4,7,10,...]</td>
<td>0.02377704988307...</td>
</tr>
</tbody>
</table>

(continues on next page)
• Average rating and airlines for each day

• Average rating and airlines for each month

• Topic 1 corresponding to time line

• reviews (documents) relate to topic 1
SELECT id, airline, date, year_month, rating, topic_1, topic_2
FROM enrichedData
WHERE date = '2014-01-01'
ORDER BY date

US Airways  Southwest Airlines  Delta Air Lines  United Airlines  American Airlines

```
10263  US Airways  2014-05-25  "Delays on all booked flights. Outward bound - Dublin to Philadelphia Philadelphia to Vegas. Vegas to LA. Baggage did not arrive some hours later. Cabin staff were unfriendly and quite rude. From Dublin We were sitting at the back of the plane and we were told "that's what you get when you travel at the back". I opened my little tub of butter which had been heated in hot liquid and went all over my hand. I said to the stewardess who was passing by who could have brought me a napkin to sped by saying "oh I know that happens!". Only that I had a tray of food on my lap she would have had more to deal with! W baggage was to be stored in the overhead lockers or underneath the seat in front. This obviously does not apply to cabin staf back centre aisle seats of the plane it was not assured in any way. Their baggage was a danger to all the passengers in that what they preash and put staff baggage in the hold. Cabin crew were unpronounced in appearance. Homeward bound to Dublin delayed resulting in us overnighting in Orlando very grateful for the overnight accommodation provided at the Hyatt. Flew to ( Boarding for Dublin at Charlotte was very confused and inefficiently exercised by Gate staff - resulting in delay in take-off. En quality on those flights that had the facility. General impression overall. Very Disappointing."
```
15.1 Introduction

15.2 Co-occurrence Network

Co-occurrence networks are generally used to provide a graphic visualization of potential relationships.
between people, organizations, concepts or other entities represented within written material. The generation and visualization of co-occurrence networks has become practical with the advent of electronically stored text amenable to text mining.

### 15.2.1 Methodology

- Build Corpus C
- Build Document-Term matrix D based on Corpus C
- Compute Term-Document matrix $D^T$
- Adjacency Matrix $A = D^T \cdot D$

There are four main components in this algorithm in the algorithm: Corpus C, Document-Term matrix D, Term-Document matrix $D^T$ and Adjacency Matrix A. In this demo part, I will show how to build those four main components.

Given that we have three groups of friends, they are

```
+-------------------------------------+
| words                              |
+-------------------------------------+
| [[george] [jimmy] [john] [peter]]  |
| [[vincent] [george] [stefan] [james]] |
| [[emma] [james] [olivia] [george]] |
+-------------------------------------+
```

1. Corpus C

Then we can build the following corpus based on the unique elements in the given group data:

```
[u'george', u'james', u'jimmy', u'peter', u'stefan', u'vincent', u'
˓→olivia', u'john', u'emma']
```

The corresponding elements frequency:

```python
from pyspark.ml.feature import CountVectorizer
count_vectorizer_wo = CountVectorizer(inputCol='term', outputCol='features')
# with total unique vocabulary
countVectorizer_mod_wo = count_vectorizer_wo.fit(df)
countVectorizer_twitter_wo = countVectorizer_mod_wo.transform(df)
# with truncated unique vocabulary (99%)
count_vectorizer = CountVectorizer(vocabSize=48, inputCol='term',
˓→outputCol='features')
countVectorizer_mod = count_vectorizer.fit(df)
countVectorizer_twitter = countVectorizer_mod.transform(df)
```
Term-Document matrix $D^T$

RDD:

```python
array([1.0, 1.0, 1.0], array([0.0, 1.0, 1.0]), array([1.0, 0.0, 0.0]), array([1.0, 0.0, 0.0]), array([0.0, 1.0, 0.0]), array([0.0, 1.0, 0.0]), array([0.0, 0.0, 1.0]), array([1.0, 0.0, 0.0]), array([0.0, 0.0, 1.0])]
```

Matrix:

```python
array([[1.0, 1.0, 1.0],
       [0.0, 1.0, 1.0],
       [1.0, 0.0, 0.0],
       [1.0, 0.0, 0.0],
       [0.0, 1.0, 0.0],
       [0.0, 1.0, 0.0],
       [0.0, 0.0, 1.0],
       [1.0, 0.0, 0.0],
       [0.0, 0.0, 1.0]])
```

3. Adjacency Matrix $A = D^T \cdot D$

RDD:
15.2.2 Coding Puzzle from my interview

- Problem

The attached utf-8 encoded text file contains the tags associated with an online biomedical scientific article formatted as follows (size: 100000). Each Scientific article is represented by a line in the file delimited by carriage return.

Write a program that, using this file as input, produces a list of pairs of tags which appear TOGETHER in any order and position in at least fifty different Scientific articles. For example, in the above sample, [Female] and [Humans] appear together twice, but every other pair appears only once. Your program should output the pair list to stdout in the same form as the input (eg tag 1, tag 2n).

- My solution

The corresponding words frequency:

Output:
### Fig. 1: Word frequency

<table>
<thead>
<tr>
<th>term.x</th>
<th>term.y</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Female</td>
<td>Humans</td>
<td>16741.0</td>
</tr>
<tr>
<td>Male</td>
<td>Humans</td>
<td>13883.0</td>
</tr>
<tr>
<td>Adult</td>
<td>Humans</td>
<td>10391.0</td>
</tr>
<tr>
<td>Male</td>
<td>Female</td>
<td>9806.0</td>
</tr>
<tr>
<td>MiddleAged</td>
<td>Humans</td>
<td>8181.0</td>
</tr>
<tr>
<td>Adult</td>
<td>Female</td>
<td>7411.0</td>
</tr>
<tr>
<td>Adult</td>
<td>Male</td>
<td>7240.0</td>
</tr>
<tr>
<td>MiddleAged</td>
<td>Male</td>
<td>6328.0</td>
</tr>
<tr>
<td>MiddleAged</td>
<td>Female</td>
<td>6002.0</td>
</tr>
<tr>
<td>MiddleAged</td>
<td>Adult</td>
<td>5944.0</td>
</tr>
</tbody>
</table>

only showing top 10 rows

The corresponding Co-occurrence network:

Then you will get Figure *Co-occurrence network*

### 15.3 Appendix: matrix multiplication in PySpark

1. load test matrix

    ```python
    df = spark.read.csv("matrix1.txt",sep="","inferSchema=True")
    df.show()
    ```

15.3. Appendix: matrix multiplication in PySpark
Fig. 2: Co-occurrence network
2. main function for matrix multiplication in PySpark

```python
from pyspark.sql import functions as F
from functools import reduce

# reference: https://stackoverflow.com/questions/44348527/matrix-
˓→multiplication-at-a-in-pyspark

# do the sum of the multiplication that we want, and get
# one data frame for each column

colDFs = []

for c2 in df.columns:
    colDFs.append(df.select(F.sum(df[c1] * df[c2]).alias("op_{0}".format(i))
˓→for i, c1 in enumerate(df.columns) )

# now union those separate data frames to build the "matrix"
mtxDF = reduce(lambda a, b: a.select(a.columns).union(b.select(a.columns)),
˓→colDFs)

mtxDF.show()
```

3. Validation with python version

```python
import numpy as np

a = np.genfromtxt("matrix1.txt", delimiter="",

np.dot(a.T, a)

array([[152.45, 118.89, 57.15, 121.44],
       [118.89, 104.95, 38.93, 94.71],
       [ 57.15, 38.93, 52.54, 55.99],
       [121.44, 94.71, 55.99, 110.11]])
```
15.4 Correlation Network

TODO ..
Chinese proverb

Don’t put all your eggs in one basket.

Code for the above figure:

```python
import numpy as np
import matplotlib.pyplot as plt
```
fig, ax = plt.subplots(figsize=(10, 8), subplot_kw=dict(aspect="equal"))

recipe = ["375 k U.S. Large Cap Blend", "300 k U.S. Large Cap Value", "75 k U.S. Short-Term Bonds", "50 k U.S. Small Cap Blend", "55 k U.S. Small Cap Value", "95 k U.S. Real Estate", "250 k Intermediate-Term Bonds"]

data = [float(x.split()[0]) for x in recipe]
ingredients = [' '.join(x.split()[2:]) for x in recipe]

print(data)
print(ingredients)
def func(pct, allvals):
    absolute = int(pct/100.*np.sum(allvals))
    return "{:1f}%
    ({:d} k)".format(pct, absolute)

explode = np.empty(len(data))
explode.fill(0.1)

wedges, texts, autotexts = ax.pie(data, explode=explode, autopct=lambda pct:
    func(pct, data),
    textprops=dict(color="w"))

ax.legend(wedges, ingredients,
    #title="Stock portfolio",
    loc="center left",
    bbox_to_anchor=(1, 0, 0.5, 1))

plt.setp(autotexts, size=8, weight="bold")

#ax.set_title("Stock portfolio")
plt.show()
16.2 Alternating Least Squares

Apache Spark ML implements ALS for collaborative filtering, a very popular algorithm for making recommendations.

ALS recommender is a matrix factorization algorithm that uses Alternating Least Squares with Weighted-Lamda-Regularization (ALS-WR). It factors the user to item matrix $A$ into the user-to-feature matrix $U$ and the item-to-feature matrix $M$: It runs the ALS algorithm in a parallel fashion. The ALS algorithm should uncover the latent factors that explain the observed user to item ratings and tries to find optimal factor weights to minimize the least squares between predicted and actual ratings.

https://www.elenacuoco.com/2016/12/22/alternating-least-squares-als-spark-ml/

16.3 Demo

- The Jupyter notebook can be download from ALS Recommender systems.
- The data can be download from German Credit.

16.3.1 Load and clean data

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark RFM example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

2. Load dataset

```python
df_raw = spark.read.format('com.databricks.spark.csv').
     options(header='true', \
             inferschema='true').
     load("Online Retail.csv",header=True);
```

check the data set

```python
df_raw.show(5)
df_raw.printSchema()
```

Then you will get

(continues on next page)
3. Data clean and data manipulation

   • check and remove the null values

```python
from pyspark.sql.functions import count

def my_count(df_in):
    df_in.agg( *[ count(c).alias(c) for c in df_in.columns ] ).show()
```

```python
import pyspark.sql.functions as F
from pyspark.sql.functions import round

df_raw = df_raw.withColumn('Asset', round( F.col('Quantity') * F.col('UnitPrice'), 2 ))
df = df_raw.withColumnRenamed('StockCode', 'Cusip')
    .select('CustomerID','Cusip','Quantity','UnitPrice','Asset')

my_count(df)
```

Since the count results are not the same, we have some null value in the CustomerID column. We can
drop these records from the dataset.

```python
df = df.filter(F.col('Asset')>=0)
df = df.dropna(how='any')
my_count(df)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Cusip</th>
<th>Quantity</th>
<th>UnitPrice</th>
<th>Asset</th>
</tr>
</thead>
<tbody>
<tr>
<td>397924</td>
<td>397924</td>
<td>397924</td>
<td>397924</td>
<td>397924</td>
</tr>
</tbody>
</table>

```
df.show(3)
```

```
<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Cusip</th>
<th>Quantity</th>
<th>UnitPrice</th>
<th>Asset</th>
</tr>
</thead>
<tbody>
<tr>
<td>17850</td>
<td>85123A</td>
<td>6</td>
<td>2.55</td>
<td>15.3</td>
</tr>
<tr>
<td>17850</td>
<td>71053</td>
<td>6</td>
<td>3.39</td>
<td>20.34</td>
</tr>
<tr>
<td>17850</td>
<td>84406B</td>
<td>8</td>
<td>2.75</td>
<td>22.0</td>
</tr>
</tbody>
</table>
```

- Convert the Cusip to consistent format

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType
def toUpper(s):
    return s.upper()
upper_udf = udf(lambda x: toUpper(x), StringType())
```

- Find the most top n stockes

```python
pop = df.groupBy('Cusip')
    .agg(F.count('CustomerID').alias('Customers'), F.round(F.sum('Asset'), 2).alias('TotalAsset'))
    .sort([F.col('Customers'), F.col('TotalAsset')], ascending=[0, 0])
pop.show(5)
```

<table>
<thead>
<tr>
<th>Cusip</th>
<th>Customers</th>
<th>TotalAsset</th>
</tr>
</thead>
<tbody>
<tr>
<td>85123A</td>
<td>2035</td>
<td>100603.5</td>
</tr>
<tr>
<td>22423</td>
<td>1724</td>
<td>142592.95</td>
</tr>
<tr>
<td>85099B</td>
<td>1618</td>
<td>85220.78</td>
</tr>
<tr>
<td>84879</td>
<td>1408</td>
<td>56580.34</td>
</tr>
<tr>
<td>47566</td>
<td>1397</td>
<td>68844.33</td>
</tr>
</tbody>
</table>
```

16.3. Demo
### 16.3.2 Build feature matrix

- Fetch the top \( n \) cusip list

```python
top = 10
cusip_lst = pd.DataFrame(pop.select('Cusip').head(top)).astype('str').iloc[:, 0].tolist()
cusip_lst.insert(0, 'CustomerID')
```

- Create the portfolio table for each customer

```python
pivot_tab = df.groupBy('CustomerID').pivot('Cusip').sum('Asset')
pivot_tab = pivot_tab.fillna(0)
```

- Fetch the most \( n \) stock’s portfolio table for each customer

```python
selected_tab = pivot_tab.select(cusip_lst)
selected_tab.show(4)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>85123A</th>
<th>22423</th>
<th>85099B</th>
<th>84879</th>
<th>47566</th>
<th>20725</th>
<th>22720</th>
<th>20727</th>
<th>POST</th>
<th>23203</th>
</tr>
</thead>
<tbody>
<tr>
<td>16503</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>33.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>15727</td>
<td>123.9</td>
<td>25.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>33.0</td>
<td>99.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>14570</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>14450</td>
<td>0.0</td>
<td>0.0</td>
<td>8.32</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>49.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

(continues on next page)

- Build the rating matrix

```python
def elemwiseDiv(df_in):
    num = len(df_in.columns)
    temp = df_in.rdd.map(lambda x: list(flatten([x[0], [x[i]/float(sum(x[1:])) if sum(x[1:])>0 else x[i] for i in range(1, num)]])))
    return spark.createDataFrame(temp, df_in.columns)

ratings = elemwiseDiv(selected_tab)
```

```python
ratings.show(4)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>85123A</th>
<th>22423</th>
<th>85099B</th>
<th>84879</th>
<th>47566</th>
<th>20725</th>
<th>22720</th>
<th>20727</th>
<th>POST</th>
<th>23203</th>
</tr>
</thead>
<tbody>
<tr>
<td>16503</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>15727</td>
<td>0.44</td>
<td>0.09</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

(continues on next page)
• Convert rating matrix to long table

```python
from pyspark.sql.functions import array, col, explode, struct, lit

def to_long(df, by):
    
    """
    """

    # Filter dtypes and split into column names and type description
    cols, dtypes = zip(*((c, t) for (c, t) in df.dtypes if c not in by))

    # Spark SQL supports only homogeneous columns
    assert len(set(dtypes)) == 1, "All columns have to be of the same type"

    # Create and explode an array of (column_name, column_value) structs
    kvs = explode(array([struct(lit(c).alias("Cusip"), col(c).alias("rating")) for c in cols ])).alias("kvs")

    df_all = to_long(ratings, ['CustomerID'])
    df_all.show(5)
```

```
+----------+------+-----+------+-----+-----+-----+-----+-----+----+-----+
|CustomerID| Cusip|rating|
+----------+------+-----+------+-----+-----+-----+-----+-----+----+-----+
| 16503|85123A| 0.0|
| 16503| 22423| 0.0|
| 16503|85099B| 0.0|
| 16503| 84879| 0.0|
| 16503| 47566| 0.0|
+----------+------+-----+------+-----+-----+-----+-----+-----+----+-----+
only showing top 5 rows
```

• Convert the string Cusip to numerical index

```python
from pyspark.ml.feature import StringIndexer

# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='Cusip',
                              outputCol='indexedCusip').fit(df_all)

df_all = labelIndexer.transform(df_all)

df_all.show(5, True)

df_all.printSchema()
```

```
+----------+------+------+------------+
|CustomerID| Cusip|rating|indexedCusip|
+----------+------+------+------------+
| 16503|85123A| 0.0|          |
| 16503| 22423| 0.0|          |
| 16503|85099B| 0.0|          |
| 16503| 84879| 0.0|          |
| 16503| 47566| 0.0|          |
+----------+------+------+------------+
```
<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Cusip</th>
<th>rating</th>
<th>indexedCusip</th>
</tr>
</thead>
<tbody>
<tr>
<td>16503</td>
<td>85123A</td>
<td>0.0</td>
<td>6.0</td>
</tr>
<tr>
<td>16503</td>
<td>22423</td>
<td>0.0</td>
<td>9.0</td>
</tr>
<tr>
<td>16503</td>
<td>85099B</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>16503</td>
<td>84879</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>16503</td>
<td>47566</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

only showing top 5 rows

root
|-- CustomerID: long (nullable = true)
|-- Cusip: string (nullable = false)
|-- rating: double (nullable = true)
|-- indexedCusip: double (nullable = true)

16.3.3 Train model

- build train and test dataset

```
train, test = df_all.randomSplit([0.8, 0.2])
```

```
train.show(5)
test.show(5)
```

```
+----------+------|------------+------------------+
|CustomerID|Cusip|indexedCusip| rating|
+----------+------|------------+------------------+
| 12940    |20725 | 2.0         | 0.0     |
| 12940    |20727 | 4.0         | 0.0     |
| 12940    |22423 | 9.0|0.49990198000392083|
| 12940    |22720 | 3.0         | 0.0     |
| 12940    |23203 | 7.0         | 0.0     |
+----------+------|------------+------------------+ only showing top 5 rows

```

```+----------+------|------------+------------------+
|CustomerID|Cusip|indexedCusip| rating|
+----------+------|------------+------------------+
| 12940    |84879 | 1.0|0.1325230346990786|
| 13285    |20725 | 2.0|0.2054154995331466|
| 13285    |20727 | 4.0|0.2054154995331466|
| 13285    |47566 | 0.0 | 0.0     |
| 13623    |23203 | 7.0         | 0.0     |
+----------+------|------------+------------------+ only showing top 5 rows```

- train model
```python
import itertools
from math import sqrt
from operator import add
import sys
from pyspark.ml.recommendation import ALS

# Import necessary libraries

from pyspark.ml.evaluation import RegressionEvaluator

evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
predictionCol="prediction")

def computeRmse(model, data):
    """
    Compute RMSE (Root mean Squared Error).
    """
    predictions = model.transform(data)
    rmse = evaluator.evaluate(predictions)
    print("Root-mean-square error = " + str(rmse))
    return rmse

# Train models and evaluate them on the validation set

ranks = [4, 5]
lambdas = [0.05]
numIters = [30]
bestModel = None
bestValidationRmse = float("inf")
bestRank = 0
bestLambda = -1.0
bestNumIter = -1

val = test.na.drop()
for rank, lmbda, numIter in itertools.product(ranks, lambdas, numIters):
    als = ALS(rank=rank, maxIter=numIter, regParam=lmbda, numUserBlocks=10,
              numItemBlocks=10, implicitPrefs=False,
              alpha=1.0,
              userCol="CustomerID", itemCol="indexedCusip", seed=1, ratingCol="rating", nonnegative=True)
    model=als.fit(train)

    validationRmse = computeRmse(model, val)
    print("RMSE (validation) = %f for the model trained with " % validationRmse +
          "rank = %d, lambda = %.1f, and numIter = %d." % (rank, lmbda,
          numIter))
    if (validationRmse, bestValidationRmse):
        bestModel = model
        bestValidationRmse = validationRmse
        bestRank = rank
        bestLambda = lmbda
        bestNumIter = numIter

model = bestModel
```

16.3. Demo
16.3.4 Make prediction

- make prediction

```python
topredict = test[test['rating'] == 0]
predictions = model.transform(topredict)
predictions.filter(predictions.prediction > 0)\
  .sort([F.col('CustomerID'), F.col('Cusip')], ascending=[0, 0]).show(5)
```

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>Cusip</th>
<th>indexedCusip</th>
<th>rating</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>18283</td>
<td>47566</td>
<td>0.0</td>
<td>0.0</td>
<td>0.01625076</td>
</tr>
<tr>
<td>18282</td>
<td>85123A</td>
<td>6.0</td>
<td>0.0</td>
<td>0.057172246</td>
</tr>
<tr>
<td>18282</td>
<td>84879</td>
<td>1.0</td>
<td>0.0</td>
<td>0.059531752</td>
</tr>
<tr>
<td>18282</td>
<td>23203</td>
<td>7.0</td>
<td>0.0</td>
<td>0.010502596</td>
</tr>
<tr>
<td>18282</td>
<td>22720</td>
<td>3.0</td>
<td>0.0</td>
<td>0.053893942</td>
</tr>
</tbody>
</table>

only showing top 5 rows
Monte Carlo simulations are just a way of estimating a fixed parameter by repeatedly generating random numbers. More details can be found at A Zero Math Introduction to Markov Chain Monte Carlo Methods.

Monte Carlo simulation is a technique used to understand the impact of risk and uncertainty in financial, project management, cost, and other forecasting models. A Monte Carlo simulator helps one visualize most or all of the potential outcomes to have a better idea regarding the risk of a decision. More details can be found at The house always wins.

### 17.1 Simulating Casino Win

We assume that the player John has the 49% chance to win the game and the wager will be $5 per game.
Learning Apache Spark with Python

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

start_m = 100
wager = 5
bets = 100
trials = 1000

trans = np.vectorize(lambda t: -wager if t <= 0.51 else wager)

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
end_m = []

for i in range(trials):
    money = reduce(lambda c, x: c + [c[-1] + x], trans(np.random.randint(bets)), [start_m])
    end_m.append(money[-1])
    plt.plot(money)
plt.ylabel('Player Money in $')
plt.xlabel('Number of bets')
plt.title(('John starts the game with $%.2f and ends with $%.2f')%(start_m, sum(end_m)/len(end_m))
plt.show()
17.2 Simulating a Random Walk

17.2.1 Fetch the histirical stock price

1. Fetch the data. If you need the code for this piece, you can contact with me.

```
stock.tail(4)
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Adj Close</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-12-07</td>
<td>155.399994</td>
<td>158.050003</td>
<td>151.729996</td>
<td>153.059998</td>
<td>153.059998</td>
<td>17447900</td>
</tr>
<tr>
<td>2018-12-10</td>
<td>150.389999</td>
<td>152.809998</td>
<td>147.479996</td>
<td>151.429993</td>
<td>151.429993</td>
<td>15525500</td>
</tr>
<tr>
<td>2018-12-11</td>
<td>155.259995</td>
<td>156.240005</td>
<td>150.899994</td>
<td>151.830002</td>
<td>151.830002</td>
<td>13651900</td>
</tr>
<tr>
<td>2018-12-12</td>
<td>155.240005</td>
<td>156.169998</td>
<td>151.429993</td>
<td>151.5</td>
<td>151.5</td>
<td>16597900</td>
</tr>
</tbody>
</table>

2. Convert the str type date to date type

```
stock['Date'] = pd.to_datetime(stock['Date'])
```

3. Data visualization

```
# Plot everything by leveraging the very powerful matplotlib package
width = 10
```

(continues on next page)
height = 6
data = stock
fig = plt.figure(figsize=(width, height))
ax = fig.add_subplot(1,1,1)
ax.plot(data.Date, data.Close, label='Close'
ax.plot(data.Date, data.High, label='High')
# ax.plot(data.Date, data.Low, label='Low')
ax.set_xlabel('Date')
ax.set_ylabel('price (\$)')
ax.legend()
ax.set_title('Stock price: ' + ticker, y=1.01)
# plt.xticks(rotation=70)
plt.show()

# Plot everything by leveraging the very powerful matplotlib package
fig = plt.figure(figsize=(width, height))
ax = fig.add_subplot(1,1,1)
ax.plot(data.Date, data.Volume, label='Volume'
# ax.plot(data.Date, data.High, label='High')
# ax.plot(data.Date, data.Low, label='Low')
ax.set_xlabel('Date')
ax.set_ylabel('Volume')
ax.legend()
ax.set_title('Stock volume: ' + ticker, y=1.01)
# plt.xticks(rotation=70)
plt.show()

17.2.2 Calculate the Compound Annual Growth Rate

The formula for Compound Annual Growth Rate (CAGR) is very useful for investment analysis. It may also
be referred to as the annualized rate of return or annual percent yield or effective annual rate, depending on
the algebraic form of the equation. Many investments such as stocks have returns that can vary wildly. The
CAGR formula allows you to calculate a “smoothed” rate of return that you can use to compare to other
investments. The formula is defined as (more details can be found at CAGR Calculator and Formula)

\[
\text{CAGR} = \left( \frac{\text{End Value}}{\text{Start Value}} \right)^{\frac{365}{\text{days}}} - 1
\]

days = (stock.Date.iloc[-1] - stock.Date.iloc[0]).days
cagr = (((stock['Adj Close'].iloc[-1]) / stock['Adj Close'].iloc[0])) ** (365.0/days)) - 1
print ("CAGR =",str(round(cagr,4)*100)+"%")
mu = cagr

17.2.3 Calculate the annual volatility

A stock’s volatility is the variation in its price over a period of time. For example, one stock may have a
tendency to swing wildly higher and lower, while another stock may move in much steadier, less turbulent
Both stocks may end up at the same price at the end of day, but their path to that point can vary wildly. First, we create a series of percentage returns and calculate the annual volatility of returns. Annualizing volatility. To present this volatility in annualized terms, we simply need to multiply our daily standard deviation by the square root of 252. This assumes there are 252 trading days in a given year. More details can be found at How to Calculate Annualized Volatility.

```python
stock['Returns'] = stock['Adj Close'].pct_change()
vol = stock['Returns'].std() * np.sqrt(252)
```

### 17.2.4 Create matrix of daily returns

1. Create matrix of daily returns using random normal distribution. Generates an RDD matrix comprised of i.i.d. samples from the uniform distribution $U(0.0, 1.0)$.

```python
S = stock['Adj Close'].iloc[-1]  # starting stock price (i.e. last available, real stock price)
T = 5  # Number of trading days
mu = cagr  # Return
vol = vol  # Volatility
trials = 10000
mat = RandomRDDs.normalVectorRDD(sc, trials, T, seed=1)
```
Fig. 2: Historical Stock Volume
2. Transform the distribution in the generated RDD from U(0.0, 1.0) to U(a, b), use RandomRDDs.uniformRDD(sc, n, p, seed) .map(lambda v: a + (b - a) * v)

\[
a = \mu / T \\
b = \text{vol} / \text{math.sqrt}(T) \\
v = \text{mat}.map(\lambda x: a + (b - a) \times x)
\]

3. Convert Rdd matrix to dataframe

```python
df = v.map(\lambda x: [round(i, 6) + 1 for i in x]).toDF()
df.show(5)
```

```
+----------+----------+----------+----------+----------+
| _1| _2| _3| _4| _5|
|----------+----------+----------+----------+----------+
| 0.935234| 1.162894| 1.07972 | 1.238257| 1.066136 |
| 0.878456| 1.045922| 0.990071| 1.045552| 0.854516 |
| 0.186472| 0.947777| 0.742247| 0.940032| 1.220934 |
| 0.872928| 1.030882| 1.248644| 1.142622| 1.063762 |
| 1.09742 | 1.188537| 1.137283| 1.162548| 1.024612 |
+----------+----------+----------+----------+----------+
only showing top 5 rows
```

```python
from pyspark.sql.functions import lit
S = stock['Adj Close'].iloc[-1]
price = df.withColumn('init_price', lit(S))
```

```
price.show(5)
```

```
+----------+----------+----------+----------+----------+----------+
| _1| _2| _3| _4| _5| init_price|
|----------+----------+----------+----------+----------+----------+
| 0.935234| 1.162894| 1.07972 | 1.238257| 1.066136| 151.5 |
| 0.878456| 1.045922| 0.990071| 1.045552| 0.854516| 151.5 |
| 0.186472| 0.947777| 0.742247| 0.940032| 1.220934| 151.5 |
| 0.872928| 1.030882| 1.248644| 1.142622| 1.063762| 151.5 |
| 1.09742 | 1.188537| 1.137283| 1.162548| 1.024612| 151.5 |
+----------+----------+----------+----------+----------+----------+
only showing top 5 rows
```

```python
price = price.withColumn('day_0', col('init_price'))
```

```
price.show(5)
```

```
+----------+----------+----------+----------+----------+----------+
| _1| _2| _3| _4| _5| init_price| day_0 |
|----------+----------+----------+----------+----------+----------+----------+
| 0.935234| 1.162894| 1.07972 | 1.238257| 1.066136| 151.5 | 151.5 |
| 0.878456| 1.045922| 0.990071| 1.045552| 0.854516| 151.5 | 151.5 |
| 0.186472| 0.947777| 0.742247| 0.940032| 1.220934| 151.5 | 151.5 |
| 0.872928| 1.030882| 1.248644| 1.142622| 1.063762| 151.5 | 151.5 |
| 1.09742 | 1.188537| 1.137283| 1.162548| 1.024612| 151.5 | 151.5 |
+----------+----------+----------+----------+----------+----------+----------+
(continues on next page)

17.2. Simulating a Random Walk
17.2.5 Monte Carlo Simulation

```python
from pyspark.sql.functions import round
for name in price.columns[:-2]:
    price = price.withColumn('day' + name, round(col(name) + col('init_price'), 2))
    price = price.withColumn('init_price', col('day' + name))
price.show(5)
```

```
only showing top 5 rows
```

17.2.6 Summary

```python
selected_col = [name for name in price.columns if 'day_' in name]
simulated = price.select(selected_col)
simulated.describe().show()
```

```
only showing top 5 rows
```
### Learning Apache Spark with Python

(continued from previous page)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>151.5</td>
<td>155.1</td>
<td>158.5</td>
<td>162.2</td>
<td>166.0</td>
</tr>
<tr>
<td></td>
<td>170.0</td>
<td>170.1</td>
<td>173.0</td>
<td>176.2</td>
<td>180.6</td>
</tr>
<tr>
<td>std</td>
<td>0.0</td>
<td>18.3</td>
<td>26.5</td>
<td>33.4</td>
<td>41.3</td>
</tr>
<tr>
<td>min</td>
<td>151.5</td>
<td>88.2</td>
<td>74.5</td>
<td>65.9</td>
<td>68.2</td>
</tr>
<tr>
<td></td>
<td>58.8</td>
<td>58.2</td>
<td>58.7</td>
<td>58.2</td>
<td>58.3</td>
</tr>
<tr>
<td>25%</td>
<td>151.5</td>
<td>142.5</td>
<td>140.1</td>
<td>138.8</td>
<td>138.8</td>
</tr>
<tr>
<td></td>
<td>137.3</td>
<td>137.3</td>
<td>137.3</td>
<td>137.3</td>
<td>137.3</td>
</tr>
<tr>
<td>50%</td>
<td>151.5</td>
<td>152.0</td>
<td>157.2</td>
<td>159.8</td>
<td>162.6</td>
</tr>
<tr>
<td></td>
<td>165.0</td>
<td>165.0</td>
<td>165.0</td>
<td>165.0</td>
<td>165.0</td>
</tr>
<tr>
<td>75%</td>
<td>151.5</td>
<td>167.4</td>
<td>175.5</td>
<td>182.9</td>
<td>189.7</td>
</tr>
<tr>
<td></td>
<td>196.9</td>
<td>196.9</td>
<td>196.9</td>
<td>196.9</td>
<td>196.9</td>
</tr>
<tr>
<td>max</td>
<td>151.5</td>
<td>227.5</td>
<td>275.9</td>
<td>319.2</td>
<td>353.6</td>
</tr>
<tr>
<td></td>
<td>403.7</td>
<td>403.7</td>
<td>403.7</td>
<td>403.7</td>
<td>403.7</td>
</tr>
</tbody>
</table>

```python
data_plt = simulated.toPandas()
days = pd.date_range(stock['Date'].iloc[-1], periods= T+1, freq='B').date
width = 10
height = 6
fig = plt.figure(figsize=(width, height))
ax = fig.add_subplot(1,1,1)
days = pd.date_range(stock['Date'].iloc[-1], periods= T+1, freq='B').date

for i in range(trials):
    plt.plot(days, data_plt.iloc[i])
ax.set_xlabel('Date')
ax.set_ylabel('price ($)')
ax.set_title('Simulated Stock price: ' + ticker, y=1.01)
plt.show()
```

#### 17.2.7 One-year Stock price simulation
Fig. 3: Simulated Stock Price
Fig. 4: Simulated Stock Price distribution
Chinese proverb

A book is known in time of need.

Monte Carlo simulations are just a way of estimating a fixed parameter by repeatedly generating random numbers. More details can be found at A Zero Math Introduction to Markov Chain Monte Carlo Methods.

Markov Chain Monte Carlo (MCMC) methods are used to approximate the posterior distribution of a parameter of interest by random sampling in a probabilistic space. More details can be found at A Zero Math Introduction to Markov Chain Monte Carlo Methods.

The following theory and demo are from Dr. Rebecca C. Steorts’s Intro to Markov Chain Monte Carlo. More details can be found at Dr. Rebecca C. Steorts’s STA 360/601: Bayesian Methods and Modern Statistics class at Duke.

18.1 Metropolis algorithm

The Metropolis algorithm takes three main steps:

1. Sample $\theta^* \sim J(\theta|\theta^{(s)})$
2. Compute the acceptance ratio \( r \)

\[
r = \frac{p(\theta^* | y)}{p(\theta^{(s)} | y)} = \frac{p(y | \theta^*) p(\theta^*)}{p(y | \theta^{(s)}) p(\theta^{(s)})}
\]

3. Let

\[
\theta^{(s+1)} = \begin{cases} 
\theta^* & \text{with prob } \min(r, 1) \\
\theta^{(s)} & \text{otherwise}
\end{cases}
\]

(18.1)

Note: Actually, the (18.1) in Step 3 can be replaced by sampling \( u \sim \text{Uniform}(0, 1) \) and setting \( \theta^{(s+1)} = \theta^* \) if \( u < r \) and setting \( \theta^{(s+1)} = \theta^{(s)} \) otherwise.

### 18.2 A Toy Example of Metropolis

The following example is going to test out the Metropolis algorithm for the conjugate Normal-Normal model with a known variance situation.

#### 18.2.1 Conjugate Normal-Normal model

\( X_1, \ldots, X_n \) \( \sim \text{iid Normal}(\theta, \sigma^2) \)

\( \theta \sim \text{Normal}(\mu, \tau^2) \)

Recall that the posterior of \( \theta \) is Normal\((\mu_n, \tau_n^2)\), where

\[
\mu_n = \bar{x} \frac{n/\sigma^2}{n/\sigma^2 + 1/\tau^2} + \mu \frac{1/\tau^2}{n/\sigma^2 + 1/\tau^2}
\]

and

\[
\tau_n^2 = \frac{1}{n/\sigma^2 + 1/\tau^2}
\]
18.2.2 Example setup

The rest of the parameters are $\sigma^2 = 1$, $\tau^2 = 10$, $\mu = 5$, $n = 5$ and

$$y = [9.37, 10.18, 9.16, 11.60, 10.33]$$

For this setup, we get that $\mu_n = 10.02745$ and $\tau_n^2 = 0.1960784$.

18.2.3 Essential mathematical derivation

In the Metropolis algorithm, we need to compute the acceptance ratio $r$, i.e.

$$r = \frac{p(\theta^*|x)}{p(\theta^{(s)}|x)} = \frac{p(x|\theta^*)p(\theta^*)}{p(x|\theta^{(s)})p(\theta^{(s)})} = \left( \frac{\prod_i \text{dnorm}(x_i, \theta^*, \sigma)}{\prod_i \text{dnorm}(x_i, \theta^{(s)}, \sigma)} \right) \left( \frac{\text{dnorm}(\theta^*, \mu, \tau)}{\text{dnorm}(\theta^{(s)}, \mu, \tau)} \right)$$

In many cases, computing the ratio $r$ directly can be numerically unstable, however, this can be modified by taking $\log r$. i.e.

$$\log r = \sum_i \left( \log[\text{dnorm}(x_i, \theta^*, \sigma)] - \log[\text{dnorm}(x_i, \theta^{(s)}, \sigma)] \right) + \sum_i \left( \log[\text{dnorm}(\theta^*, \mu, \tau)] - \log[\text{dnorm}(\theta^{(s)}, \mu, \tau)] \right)$$

Then the criteria of the acceptance becomes: if $\log u < \log r$, where $u$ is sample from the Uniform$(0, 1)$.

18.3 Demos

Now, we generate $S$ iterations of the Metropolis algorithm starting at $\theta^{(0)} = 0$ and using a normal proposal distribution, where

$$\theta^{(s+1)} \sim \text{Normal}(\theta^{(s)}, 2).$$

18.3.1 R results
# setting values
set.seed(1)
s2<-1
t2<-10
mu<-5; n<-5

# rounding the rnorm to 2 decimal places
y<-round(rnorm(n,10,1),2)

# mean of the normal posterior
mu.n<-( mean(y)*n/s2 + mu/t2 )/( n/s2+1/t2)

# variance of the normal posterior
t2.n<-1/(n/s2+1/t2)

# defining the data
y<-c(9.37, 10.18, 9.16, 11.60, 10.33)

####metropolis part####
##S = total num of simulations
theta<-0 ; delta<-2 ; S<-10000 ; THETA<-NULL ; set.seed(1)
for(s in 1:S){
  ## simulating our proposal
  theta.star<-rnorm(1,theta,sqrt(delta))
  ## taking the log of the ratio r
  log.r<-( sum(dnorm(y,theta.star,sqrt(s2),log=TRUE))+
          dnorm(theta.star,mu,sqrt(t2),log=TRUE))-
          ( sum(dnorm(y,theta,sqrt(s2),log=TRUE))+
            dnorm(theta,mu,sqrt(t2),log=TRUE))
  #print(log.r)
  if(log(runif(1))<log.r) { theta<-theta.star }
  ## updating THETA
  #print(log(runif(1)))
  THETA<-c(THETA,theta)
}

## two plots: trace of theta and comparing the empirical distribution
## of simulated values to the true posterior
par(mar=c(3,3,1,1),mgp=c(1.75,.75,0))
par(mfrow=c(1,2))

# creating a sequence
skeep<-seq(10,S,by=10)
# making a trace place
plot(skeep,THETA[skeep],type="l",
     xlab="iteration",ylab=expression(theta))
# making a histogram
hist(THETA[-(1:50)],prob=TRUE,main="",
     xlab=expression(theta),ylab="density")

th<-seq(min(THETA),max(THETA),length=100)
lines(th,dnorm(th,mu.n,sqrt(t2.n)))

Figure. Histogram for the Metropolis algorithm with r shows a trace plot for this run as well as a histogram for the Metropolis algorithm compared with a draw from the true normal density.
18.3.2 Python results

```python
# coding: utf-8

# In[1]:
import numpy as np

# In[2]:
from scipy.stats import norm

def rnorm(n, mean, sd):
    
    """
    same functions as rnorm in r
    r: rnorm(n, mean=0, sd=1)
    py: rvs(loc=0, scale=1, size=1, random_state=None)
    """
    return norm.rvs(loc=mean, scale=sd, size=n)

def dnorm(x, mean, sd, log=False):
    
    """
    same functions as dnorm in r
    """
```

(continues on next page)
```python
dnorm(x, mean=0, sd=1, log=FALSE)
pdf(x, loc=0, scale=1)

""
if log:
    return np.log(norm.pdf(x=x, loc=mean, scale=sd))
else:
    return norm.pdf(x=x, loc=mean, scale=sd)

def runif(n, min=0, max=1):
    ""
    r: runif(n, min = 0, max = 1)
    py: random.uniform(low=0.0, high=1.0, size=None)
    ""
    return np.random.uniform(min, max, size=n)

# In[3]:
s2 = 1
t2 = 10
mu = 5
n = 5

# In[4]:
y = rnorm(n, 10, 1)
y

# In[5]:
# mean of the normal posterior
mu_n = (np.mean(y)*n/s2 + mu/float(t2))/(n/float(s2)+1/float(t2))
mu_n

# In[6]:
# variance of the normal posterior
# t2.n<-1/(n/s2+1/t2)
t2_n = 1.0/(n/float(s2)+1.0/t2)
t2_n

# In[7]:
# defining the data
# y<-c(9.37, 10.18, 9.16, 11.60, 10.33)
```

(continues on next page)
Learning Apache Spark with Python

y = [9.37, 10.18, 9.16, 11.60, 10.33]

# In[8]:

mu_n = (np.mean(y)*n/s2 + mu/float(t2))/(n/float(s2)+1/float(t2))
mu_n

# In[9]:

###metropolis part####

##S = total num of simulations
# theta<-0 ; delta<-2 ; S<-10000 ; THETA<-NULL ; set.seed(1)

theta = 0
delta = 2
S = 10000
theta_v = []

# In[ ]:

for s in range(S):
    theta_star = norm.rvs(theta,np.sqrt(delta),1)
    logr = (sum(dnorm(y,theta_star,np.sqrt(s2),log=True)) +
            sum(dnorm(theta_star,mu,np.sqrt(t2),log=True))-
            (sum(dnorm(y,theta,np.sqrt(s2),log=True)) +
             sum(dnorm([theta],mu,np.sqrt(t2),log=True))))
    #print(logr)
    if np.log(runif(1))<logr:
        theta = theta_star
    #print(theta)
    theta_v.append(theta)

# In[ ]:

import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 8))

plt.subplot(1, 2, 1)
plt.plot(theta_v,'b-.')
plt.subplot(1, 2, 2)
#bins = np.arange(0, S, 2)
plt.hist(theta_v, density=True,bins='auto')
x = np.linspace(min(theta_v),max(theta_v),100)

(continues on next page)
Fig. 2: Histogram for the Metropolis algorithm with python

Figure. *Histogram for the Metropolis algorithm with python* shows a trace plot for this run as well as a histogram for the Metropolis algorithm compared with a draw from the true normal density.

### 18.3.3 PySpark results

TODO...

Figure. *Histogram for the Metropolis algorithm with PySpark* shows a trace plot for this run as well as a histogram for the Metropolis algorithm compared with a draw from the true normal density.
Fig. 3: Histogram for the Metropolis algorithm with PySpark
Chinese proverb
Sharpening the knife longer can make it easier to hack the firewood – old Chinese proverb

19.1 Feedforward Neural Network

19.1.1 Introduction
A feedforward neural network is an artificial neural network wherein connections between the units do not form a cycle. As such, it is different from recurrent neural networks.

The feedforward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward (see Fig. MultiLayer Neural Network), from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

Fig. 1: MultiLayer Neural Network
19.1.2 Demo

1. Set up spark context and SparkSession

```python
from pyspark.sql import SparkSession

spark = SparkSession.
    builder.
    appName("Python Spark Feedforward neural network example")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
```

2. Load dataset

```
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
|fixed|volatile|citric|sugar|chlorides|free|total|density|
+-----+--------+------+-----+---------+----+-----+-------+----+---------+-----+
|  7.4|  0.7|  0.0|  1.9|  0.076|11.0| 34.0|  0.9978|3.51|  0.56|
|  7.4|  0.7|  0.0|  1.9|  0.076|11.0| 34.0|  0.9978|3.51|  0.56|
|  7.4|  0.7|  0.0|  1.9|  0.076|11.0| 34.0|  0.9978|3.51|  0.56|
|  7.4|  0.7|  0.0|  1.9|  0.076|11.0| 34.0|  0.9978|3.51|  0.56|
|  7.4|  0.7|  0.0|  1.9|  0.076|11.0| 34.0|  0.9978|3.51|  0.56|
```

only showing top 5 rows

3. change categorical variable size

```python
# Convert to float format

def string_to_float(x):
    return float(x)

# def condition(r):
#     if (0<= r <= 4):
#         label = "low"
#     elif (4< r <= 6):
#         label = "medium"
#     else:
#         label = "high"
#     return label

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, DoubleType
```

(continues on next page)
string_to_float_udf = udf(string_to_float, DoubleType())
quality_udf = udf(lambda x: condition(x), StringType())
df = df.withColumn("quality", quality_udf("quality"))

4. Convert the data to dense vector

```python
# convert the data to dense vector
def transData(data):
    return data.rdd.map(lambda r: [r[-1], Vectors.dense(r[:-1])]).toDF(["label", "features"])
```

```python
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

data = transData(df)
data.show()
```

5. Split the data into training and test sets (40% held out for testing)

```python
# Split the data into train and test
(trainingData, testData) = data.randomSplit([0.6, 0.4])
```

6. Train neural network

```python
# specify layers for the neural network:
# input layer of size 11 (features), two intermediate of size 5 and 4
# and output of size 7 (classes)
layers = [11, 5, 4, 4, 3, 7]

# create the trainer and set its parameters
FNN = MultilayerPerceptronClassifier(labelCol="indexedLabel", 
                                      featuresCol="indexedFeatures",
                                      maxIter=100, layers=layers, 
                                      blockSize=128, seed=1234)

# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", 
                               labels=labelIndexer.labels)

# Chain indexers and forest in a Pipeline
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, FNN, labelConverter])

# train the model
# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)
```

7. Make predictions

```python
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

19.1. Feedforward Neural Network
8. Evaluation

```python
# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Predictions accuracy = \$g, Test Error = \$g \% ") % (accuracy, (1.0 - accuracy))
```
CDH (Cloudera Distribution Hadoop) is the most complete, tested, and widely deployed distribution of Apache Hadoop. A lot of small or middle size companies are using CHD. While Cloudera does not support IPython or Jupyter notebooks on CDH and the Cloudera Data Science Workbench is expensive, many companies are using CDH+zeppelin or CDH+jupyterhub infrastructure. This infrastructure works pretty well, but it’s inconvenient for Data Engineer or Data Scientist to do automation during the production process. This chapter will cover how to use Jinja2, spark sql and ML Pipelines to implement the automation for Cloudera Distribution Hadoop.

20.1 Automation Pipeline

The automation pipeline mainly contains two parts:

1. Jinja2 + spark sql for data clean and manipulation automation
2. ML Pipelines for Machine Learning automation

20.2 Data Clean and Manipulation Automation

20.2.1 Jinja 2

Jinja is a modern and designer-friendly templating language for Python, modelled after Django’s templates. Use Jinja2 to generate SQL query will need two steps:

1. Get template
temp = ""
    SELECT project, timesheet, hours
    FROM timesheet
    WHERE user_id = {{ user_id }}
    (if project_id %)
    AND project_id = {{ project_id }}
    (endif %)
""

2. render the template

```python
args = {"user_id": u"runawayhorse",
        "project_id": 123}
query = Template(temp).render(args)
print(query)
```

Then, you will get the following SQL query:

```sql
SELECT project, timesheet, hours
FROM timesheet
WHERE user_id = runawayhorse
AND project_id = 123
```

Note

The Jinja is smart then you think. If you try this

```python
args = {"user_id": u"runawayhorse"}
query = Template(temp).render(args)
print(query)
```

Then, you will get the following SQL query:

```sql
SELECT project, timesheet, hours
FROM timesheet
WHERE user_id = runawayhorse
```

If you have a long query, you can use Jinja `get_template` to read the template:

```python
import os
from jinja2 import Template
from jinja2 import Environment, FileSystemLoader

path = os.path.abspath(os.path.join(sys.path[0]))
```
try:
    os.mkdir(path)
except OSError:
    pass
os.chdir(path)
print(path)

jinja_env = Environment(loader=FileSystemLoader(path))
template = jinja_env.get_template('test.sql')
query = template.render(states=states)
print(query)

with test.sql file is as follows:

```sql
select id
{% for var in states %}
, (CASE WHEN (off_st =('{{var}}')) THEN 1 ELSE 0 END) AS off_st_{{var}}
{% endfor %}
FROM table1
```

Then you will get the following query:

```sql
select id
, (CASE WHEN (off_st = 'MO') THEN 1 ELSE 0 END) AS off_st_MO
, (CASE WHEN (off_st = 'KS') THEN 1 ELSE 0 END) AS off_st_KS
, (CASE WHEN (off_st = 'KY') THEN 1 ELSE 0 END) AS off_st_KY
, (CASE WHEN (off_st = 'OH') THEN 1 ELSE 0 END) AS off_st_OH
FROM table1
```

### 20.2.2 Spark SQL

Spark SQL at here will be called to execute SQL or HiveQL queries which generated by Jinjia2 on existing warehouses.

```python
# without output
spark.sql(query)

# with output
df = spark.sql(query)
```
20.3 ML Pipeline Automation

I will not cover the details of the ML Pipeline at here, the interested reader is referred to ML Pipelines. The main steps for defining the stages are as follows:

```python
scaler = 'Standard'

from pyspark.ml.feature import Normalizer, StandardScaler, MinMaxScaler
if scaler == 'Normal':
    scaler = Normalizer(inputCol="features", outputCol="scaledFeatures", p=1.0)
elif scaler == 'Standard':
    scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                             withStd=True, withMean=False)
else:
    scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")

from pyspark.ml.feature import StringIndexer
# Index labels, adding metadata to the label column
labelIndexer = StringIndexer(inputCol='label',
                             outputCol='label').fit(transformed)

from pyspark.ml.feature import IndexToString
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)

from pyspark.ml.classification import LogisticRegression
ml = LogisticRegression(featuresCol='scaledFeatures', labelCol='label',
                         maxIter=100, regParam=0.01, elasticNetParam=0.6)

# Chain indexers and tree in a Pipeline
pipeline_model = Pipeline(stages=[scaler, labelIndexer, ml, labelConverter])

# Train model. This also runs the indexers.
model = pipeline_model.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)
```

20.4 Save and Load PipelineModel

```python
# save PipelineModel
model.write().overwrite().save(out_path)

# load PipelineModel
from pyspark.ml import PipelineModel
```

(continues on next page)
model = PipelineModel.load(out_path)

## 20.5 Ingest Results Back into Hadoop

```python
df.createOrReplaceTempView("temp_table")

query = '''
create table database_name.prediction_{{dt}} AS
SELECT *
FROM temp_table
'''

output = Template(query).render(dt=dt)
spark.sql(output)
```
WRAP PYSPARK PACKAGE

It’s super easy to wrap your own package in Python. I packed some functions which I frequently used in my daily work. You can download and install it from My PySpark Package. The hierarchical structure and the directory structure of this package are as follows.

21.1 Package Wrapper

21.1.1 Hierarchical Structure

```
|-- build
 | |-- bdist.linux-x86_64
 | |-- lib.linux-x86_64-2.7
 | | |-- PySparkTools
 | | | |-- __init__.py
 | | | |-- Manipulation
 | | | | |-- DataManipulation.py
 | | | | | |-- __init__.py
 | | | | | |-- Visualization
 | | | | | | |-- __init__.py
 | | | | | | |-- PyPlots.py
 ||-- dist
 | | |-- PySparkTools-1.0-py2.7.egg
 ||-- __init__.py
 |-- PySparkTools
 | | |-- __init__.py
 | | |-- Manipulation
 | | | |-- DataManipulation.py
 | | | | |-- __init__.py
 | | | |-- Visualization
 | | | | |-- __init__.py
 | | | | |-- PyPlots.py
 | | | |-- PyPlots.pyc
 |-- PySparkTools.egg-info
 | | |-- dependency_links.txt
 | | |-- PKG-INFO
 | | |-- requires.txt
 | | |-- SOURCES.txt
```
From the above hierarchical structure, you will find that you have to have `__init__.py` in each directory. I will explain the `__init__.py` file with the example below:

### 21.1.2 Set Up

```python
from setuptools import setup, find_packages

try:
    with open("README.md") as f:
        long_description = f.read()
except IOError:
    long_description = ""

try:
    with open("requirements.txt") as f:
        requirements = [x.strip() for x in f.read().splitlines() if x.strip()]
except IOError:
    requirements = []

setup(name='PySparkTools',
      install_requires=requirements,
      version='1.0',
      description='Python Spark Tools',
      author='Wenqiang Feng',
      author_email='von198@gmail.com',
      url='https://github.com/runawayhorse001/PySparkTools',
      packages=find_packages(),
      long_description=long_description)
```

### 21.1.3 ReadMe

```bash
# PySparkTools

This is my PySpark Tools. If you want to clone and install it, you can use

```
- clone
```

```bash
(continues on next page)
```
git clone git@github.com:runawayhorse001/PySparkTools.git
```
```{bash}
cd PySparkTools
pip install -r requirements.txt
python setup.py install
```
```
```{bash}
cd PySparkTools/test
python test1.py
```

21.2 Package Publishing on PyPI

21.2.1 Install `twine`

```bash
pip install twine
```

21.2.2 Build Your Package

```bash
python setup.py sdist bdist_wheel
```

Then you will get a new folder `dist`:

```
- PySparkAudit-1.0.0-py2.7.egg
- PySparkAudit-1.0.0-py2-none-any.whl
- PySparkAudit-1.0.0.tar.gz
```

21.2.3 Upload Your Package

```bash
twine upload dist/*
```

During the uploading processing, you need to provide your PyPI account username and password:

```
Enter your username: runawayhorse001
Enter your password: **************
```
21.2.4 Package at PyPI

Here is my PySparkAudit package at [PyPI](https://pypi.org/project/PySparkAudit). You can install PySparkAudit using:

```
pip install PySparkAudit
```
**PySparkAudit**: PySpark Data Audit Library. The PDF version can be downloaded from HERE. The python version **PyAudit**: Python Data Audit Library API can be found at PyAudit.

### 22.1 Install with `pip`

You can install the PySparkAudit from [PyPI](https://pypi.org/project/PySparkAudit):

```
pip install PySparkAudit
```

### 22.2 Install from Repo

#### 22.2.1 Clone the Repository

```
git clone https://github.com/runawayhorse001/PySparkAudit.git
```

#### 22.2.2 Install

```
cd PySparkAudit
pip install -r requirements.txt
python setup.py install
```

### 22.3 Uninstall

```
pip uninstall statspy
```
Learning Apache Spark with Python

22.4 Test

22.4.1 Run test code

```python
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark regression example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

# from PySparkAudit import dtypes_class, hist_plot, bar_plot, freq_items, feature_len
# from PySparkAudit import dataset_summary, rates, trend_plot

# path = '/home/feng/Desktop'
# import PySpark Audit function
from PySparkAudit import auditing

# load dataset
data = spark.read.csv(path='Heart.csv', 
                       sep=',', encoding='UTF-8', comment=None, header=True, 
                       inferSchema=True)

# auditing in one function
print(auditing(data, display=True))
```

22.4.2 Audited Results

The files in 00-audited_results.xlsx:

1. Dataset_summary

2. Numeric_summary

3. Category_summary
4. Correlation_matrix

5. Histograms in Histograms.pdf


22.5 Auditing on Big Dataset

In this section, we will demonstrate the auditing performance and audited results on the big data set. The data set is Spanish High Speed Rail tickets pricing. It is available at: https://www.kaggle.com/thegurus/spanish-high-speed-rail-system-ticket-pricing. This data set has 2579771 samples and 10 features.

From the following CPU time, you will see most of the time was spent on plotting the histograms. If your time and memory are limited, we will suggest you to use sample_size to generate the subset of the dataset to plot histograms.

For example:
## Chapter 22. PySpark Data Audit Library

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>summary</td>
<td>value</td>
</tr>
<tr>
<td>2</td>
<td>sample_size</td>
<td>303</td>
</tr>
<tr>
<td>3</td>
<td>feature_size</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>single_unique_feature</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>row_w_null</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>row_w_empty</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>row_w_zero</td>
<td>299</td>
</tr>
<tr>
<td>8</td>
<td>row_avg_null_count</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>row_avg_empty_count</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>row_avg_zero_count</td>
<td>3.25083</td>
</tr>
<tr>
<td>11</td>
<td>numerical_fields</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>categorical_fields</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>date_fields</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>unsupported_fields</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>double</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>int</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>string</td>
<td>4</td>
</tr>
</tbody>
</table>
### 22.5. Auditing on Big Dataset

- **Feature**: Features under audit.
- **Dtypes**: Data types of the features.
- **N Count**: Number of records counted.
- **N Null**: Number of null values.
- **N Distinct**: Number of distinct values.
- **Mean**: Mean value of the feature.
- **Std Dev**: Standard deviation of the feature.
- **Min**: Minimum value of the feature.
- **Max**: Maximum value of the feature.
- **Q1**: First quartile.
- **Q3**: Third quartile.
- **Med**: Median of the feature.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Dtypes</th>
<th>N Count</th>
<th>N Null</th>
<th>N Distinct</th>
<th>Mean</th>
<th>Std Dev</th>
<th>Min</th>
<th>Max</th>
<th>Q1</th>
<th>Q3</th>
<th>Med</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>4154.4389</td>
<td>9.038662</td>
<td>29</td>
<td>77</td>
<td>48</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Sex</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>20.67986</td>
<td>0.467290</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RestBP</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>50.131.689</td>
<td>17.5997</td>
<td>94</td>
<td>200</td>
<td>120</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>Chol</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>152.246.693</td>
<td>51.7769</td>
<td>126</td>
<td>564</td>
<td>211</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>Fbs</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>20.14851</td>
<td>0.356190</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>RestECG</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>30.990099</td>
<td>0.994979</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MaxHR</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>91.149.607</td>
<td>22.87506</td>
<td>71</td>
<td>202</td>
<td>134</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>ExAng</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>20.32673</td>
<td>0.469790</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Oldpeak</td>
<td>double</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>40.101360</td>
<td>1.161075</td>
<td>0.0</td>
<td>6.2</td>
<td>0</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>Slope</td>
<td>int</td>
<td>303</td>
<td>0</td>
<td>303</td>
<td>31.600660</td>
<td>0.616226</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
### Data Audit Library

<table>
<thead>
<tr>
<th>feature</th>
<th>dtypes</th>
<th>low</th>
<th>count</th>
<th>null</th>
<th>count</th>
<th>unique</th>
<th>values</th>
<th>type</th>
<th>var</th>
<th>rate</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChestPain</td>
<td>string</td>
<td>303</td>
<td>303</td>
<td>4</td>
<td>7</td>
<td>10.7228</td>
<td>['asympt', 0.0132]</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>string</td>
<td>303</td>
<td>303</td>
<td>5</td>
<td>1</td>
<td>1.0132</td>
<td>['0', 176]</td>
<td>0.0165</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thal</td>
<td>string</td>
<td>303</td>
<td>303</td>
<td>4</td>
<td>2</td>
<td>7.45875</td>
<td>['normal']</td>
<td>0.0132</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AHD</td>
<td>string</td>
<td>303</td>
<td>303</td>
<td>2</td>
<td>2</td>
<td>2.45875</td>
<td>['No', 16]</td>
<td>0.0066</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 22.5. Auditing on Big Dataset

```excel
<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>RestBP</th>
<th>Chol</th>
<th>Fbs</th>
<th>RestECG</th>
<th>MaxHR</th>
<th>ExAng</th>
<th>Oldpeak</th>
<th>Slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-0.09754</td>
<td>0.28495</td>
<td>0.20895</td>
<td>0.11853</td>
<td>0.14887</td>
<td>-0.39381</td>
<td>0.09166</td>
<td>0.20381</td>
<td>0.16177</td>
</tr>
<tr>
<td>3</td>
<td>-0.06446</td>
<td>-0.19991</td>
<td>0.04786</td>
<td>0.02165</td>
<td>-0.04866</td>
<td>0.1462</td>
<td>0.10217</td>
<td>0.03753</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.28495</td>
<td>-0.06446</td>
<td>0.13012</td>
<td>0.17534</td>
<td>0.14656</td>
<td>-0.04535</td>
<td>0.06476</td>
<td>0.18917</td>
<td>0.11738</td>
</tr>
<tr>
<td>5</td>
<td>0.20895</td>
<td>-0.19991</td>
<td>0.13012</td>
<td>0.00984</td>
<td>0.17104</td>
<td>-0.00343</td>
<td>0.06131</td>
<td>0.04656</td>
<td>-0.00406</td>
</tr>
<tr>
<td>6</td>
<td>0.11853</td>
<td>0.04786</td>
<td>0.17534</td>
<td>0.00984</td>
<td>0.06956</td>
<td>-0.00785</td>
<td>0.02567</td>
<td>0.00575</td>
<td>0.05989</td>
</tr>
<tr>
<td>7</td>
<td>0.14887</td>
<td>0.02165</td>
<td>0.14656</td>
<td>0.17104</td>
<td>0.06956</td>
<td>-0.08339</td>
<td>0.08487</td>
<td>0.11413</td>
<td>0.13395</td>
</tr>
<tr>
<td>8</td>
<td>-0.39381</td>
<td>-0.04866</td>
<td>-0.04535</td>
<td>-0.00343</td>
<td>-0.00785</td>
<td>-0.08339</td>
<td>1</td>
<td>-0.3781</td>
<td>-0.34309</td>
</tr>
<tr>
<td>9</td>
<td>0.09166</td>
<td>0.1462</td>
<td>0.06476</td>
<td>0.06131</td>
<td>0.02567</td>
<td>0.08487</td>
<td>-0.03781</td>
<td>1</td>
<td>0.28822</td>
</tr>
<tr>
<td>10</td>
<td>0.20381</td>
<td>0.10217</td>
<td>0.18917</td>
<td>0.04656</td>
<td>0.00575</td>
<td>0.11413</td>
<td>-0.34309</td>
<td>0.28822</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0.16177</td>
<td>0.03753</td>
<td>0.11738</td>
<td>-0.00406</td>
<td>0.05989</td>
<td>0.13395</td>
<td>-0.3856</td>
<td>0.25775</td>
<td>0.57754</td>
</tr>
</tbody>
</table>
```

---

For a detailed analysis of the dataset, we can use correlation matrices to identify relationships between variables. The correlation matrix shows the correlation coefficients between each pair of variables. In this example, the correlation matrix indicates the strength and direction of the relationship between different health indicators such as age, sex, blood pressure, cholesterol levels, etc. High positive correlations suggest that as one variable increases, the other variable also tends to increase, while high negative correlations indicate that as one variable increases, the other tends to decrease.
Chapter 22. PySpark Data Audit Library
from pyspark.sql import SparkSession

spark = SparkSession.
  .builder.
  .appName("Python Spark regression example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

# from PySparkAudit import dtypes_class, hist_plot, bar_plot, freq_items,
  # feature_len
# from PySparkAudit import dataset_summary, rates, trend_plot

# Audited results output path
out_path = '/home/feng/Desktop'

# import PySpark Audit function
from PySparkAudit import auditing

# load dataset
# Spanish High Speed Rail tickets pricing - Renfe (~2.58M)
# https://www.kaggle.com/thegurus/spanish-high-speed-rail-system-ticket-pricing
data = spark.read.csv(path='/home/feng/Downloads/renfe.csv',
  sep=',', encoding='UTF-8', comment=None, header=True,
  inferSchema=True)

# auditing in one function
auditing(data, output_dir=out_path, tracking=True)

Result:

22.5.1 print in bash

================================================================
The audited results summary audited_results.xlsx was located at:
/home/feng/Desktop/Audited
Generate data set summary took = 60.535009145736694 s
================================================================
Collecting data types.... Please be patient!
Generate counts took = 0.0016515254974365234 s
================================================================
Collecting features' counts.... Please be patient!
Generate counts took = 6.502962350845337 s
================================================================
Collecting data frame description.... Please be patient!
Generate data frame description took = 1.5562639236450195 s
================================================================
Calculating percentiles.... Please be patient!
Generate percentiles took = 19.76785445213318 s

(continues on next page)
Calculating features' length.... Please be patient!
Generate features' length took = 4.953453540802002 s

Calculating top 5 frequent items.... Please be patient!
Generate rates took: 4.761325359344482 s

Calculating rates.... Please be patient!
Generate rates took: 17.201056718826294 s
Auditing numerical data took = 54.77840781211853 s

Collecting data types.... Please be patient!
Generate counts took = 0.001623392105102539 s

Collecting features' counts.... Please be patient!
Generate counts took = 12.59226107597351 s

Calculating features' length.... Please be patient!
Generate features' length took = 5.332952976226807 s
Calculating top 5 frequent items.... Please be patient!
Generate rates took: 6.832213878631592 s

Calculating rates.... Please be patient!
Generate rates took: 23.704302072525024 s
Auditing categorical data took = 48.484763622283936 s

The correlation matrix plot Corr.png was located at:
/home/feng/Desktop/Audited
Calculating correlation matrix... Please be patient!
Generate correlation matrix took = 19.61273431777954 s

The Histograms plots *.png were located at:
/home/feng/Desktop/Audited/02-hist
Plotting histograms of _c0.... Please be patient!
Plotting histograms of price.... Please be patient!
Histograms plots are DONE!!!
Generate histograms plots took = 160.3421311378479 s

The Bar plot Bar_plots.pdf was located at:
/home/feng/Desktop/Audited
Plotting barplot of origin.... Please be patient!
Plotting barplot of destination.... Please be patient!
Plotting barplot of train_type.... Please be patient!
Plotting barplot of train_class.... Please be patient!
Plotting barplot of fare.... Please be patient!
Plotting barplot of insert_date.... Please be patient!
Plotting barplot of start_date.... Please be patient!
Plotting barplot of end_date.... Please be patient!
Bar plots are DONE!!!
Generate bar plots took = 24.17994236946106 s
The Trend plot Trend_plots.pdf was located at:
/home/feng/Desktop/Audited
Plotting trend plot of _c0.... Please be patient!
Plotting trend plot of price.... Please be patient!
Trend plots are DONE!!!
Generate trend plots took = 11.697550296783447 s
Generate all the figures took = 196.25823402404785 s
Generate all audited results took = 379.73954820632935 s
================================================================
The auditing processes are DONE!!!

22.5.2 Audited results folder

![Audited results folder]

22.5. Auditing on Big Dataset
The Zeppelin users may have same problem with me that the Zeppelin .json notebook is hard to open and read. **ze2nb**: A piece of code to convert Zeppelin .json notebook to .ipynb Jupyter notebook, .py and .html file. This library is based on Ryan Blue’s Jupyter/Zeppelin conversion: [jupyter-zeppelin]. The API book can be found at ze2nb API or [zeppelin2nb]. You may download and distribute it. Please be aware, however, that the note contains typos as well as inaccurate or incorrect description.

### 23.1 How to Install

#### 23.1.1 Install with pip

You can install the ze2nb from [PyPI](https://pypi.org/project/ze2nb):

```
pip install ze2nb
```

#### 23.1.2 Install from Repo

1. Clone the Repository
   
   ```
git clone https://github.com/runawayhorse001/ze2nb.git
   ```

2. Install
   
   ```
cd zeppelin2nb
pip install -r requirements.txt
python setup.py install
   ```

#### 23.1.3 Uninstall

```
pip uninstall ze2nb
```
23.2 Converting Demos

The following demos are designed to show how to use zeppelin2nb to convert the .json to .ipynb, .py and .html.

23.2.1 Converting in one function

For example:

```python
# import python library
import os, sys

# import zeppelin2nb module
from ze2nb import ze2nb

# scenario 1
# file and output at the current directory
# output path, the default output path will be the current directory
ze2nb('H2o_Sparking.json')

# scenario 2
output = os.path.abspath(os.path.join(sys.path[0])) + '/output'
ze2nb('H2o_Sparking.json', out_path=output, to_html=True, to_py=True)

# scenario 3
# with load and output path
load_path = '/Users/dt216661/Documents/MyJson/'
output = os.path.abspath(os.path.join(sys.path[0])) + '/output1'
ze2nb('H2o_GBM.json', load_path=load_path, out_path=output, to_html=True, to_py=True)
```

23.2.2 Converted results

Result:

Results in output:

Results in output1:
23.2. Converting Demos
Chapter 23. Zeppelin to jupyter notebook
23.2. Converting Demos
You can download the PDF version: PySpark Cheat Sheet and pdDataBase vs rddDataFrame.
Those APIs are automatically generated from PySpark package, so all the CopyRights belong to Spark.

25.1 Stat API

class pyspark.ml.stat.ChiSquareTest

Note: Experimental

Conduct Pearson’s independence test for every feature against the label. For each feature, the (feature, label) pairs are converted into a contingency matrix for which the Chi-squared statistic is computed. All label and feature values must be categorical.

The null hypothesis is that the occurrence of the outcomes is statistically independent.

New in version 2.2.0.

static test(dataset, featuresCol, labelCol)

Perform a Pearson’s independence test using dataset.

Parameters

• dataset – DataFrame of categorical labels and categorical features. Real-valued features will be treated as categorical for each distinct value.

• featuresCol – Name of features column in dataset, of type Vector (VectorUDT).

• labelCol – Name of label column in dataset, of any numerical type.

Returns DataFrame containing the test result for every feature against the label. This DataFrame will contain a single Row with the following fields: - pValues : Vector - degreesOfFreedom : Array[Int] - statistics : Vector Each of these fields has one value per feature.
New in version 2.2.0.

```python
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.stat import ChiSquareTest
>>> dataset = [[0, Vectors.dense([0, 0, 1])],
...             [0, Vectors.dense([1, 0, 1])],
...             [1, Vectors.dense([2, 1, 1])],
...             [1, Vectors.dense([3, 1, 1])])
>>> dataset = spark.createDataFrame(dataset, ['label', 'features'])
>>> chiSqResult = ChiSquareTest.test(dataset, 'features', 'label')
>>> chiSqResult.select('degreesOfFreedom').collect()[0]
Row(degreesOfFreedom=[3, 1, 0])
```

New in version 2.2.0.

### pyspark.ml.stat.Correlation

**Note:** Experimental

Compute the correlation matrix for the input dataset of Vectors using the specified method. Methods currently supported: *pearson* (default), *spearman*.

**Note:** For Spearman, a rank correlation, we need to create an RDD[Double] for each column and sort it in order to retrieve the ranks and then join the columns back into an RDD[Vector], which is fairly costly. Cache the input Dataset before calling *corr* with `method = 'spearman'` to avoid recomputing the common lineage.

New in version 2.2.0.

#### static corr(dataset, column, method='pearson')

Compute the correlation matrix with specified method using dataset.

**Parameters**

- **dataset** – A Dataset or a DataFrame.
- **column** – The name of the column of vectors for which the correlation coefficient needs to be computed. This must be a column of the dataset, and it must contain Vector objects.
- **method** – String specifying the method to use for computing correlation. Supported: *pearson* (default), *spearman*.

**Returns** A DataFrame that contains the correlation matrix of the column of vectors. This DataFrame contains a single row and a single column of name ‘SMETHOD-NAME($COLUMN)’.

```python
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.stat import Correlation
>>> dataset = [[Vectors.dense([1, 0, 0, -2])],
...             [Vectors.dense([1, 0, 0, -2])],
...             [Vectors.dense([1, 0, 0, -2])]]
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.stat import Correlation
>>> dataset = [[Vectors.dense([1, 0, 0, -2])],
...             [Vectors.dense([1, 0, 0, -2])],
...             [Vectors.dense([1, 0, 0, -2])]]
```
New in version 2.2.0.

class pyspark.ml.stat.KolmogorovSmirnovTest

**Note:** Experimental

Conduct the two-sided Kolmogorov Smirnov (KS) test for data sampled from a continuous distribution.

By comparing the largest difference between the empirical cumulative distribution of the sample data and the theoretical distribution we can provide a test for the the null hypothesis that the sample data comes from that theoretical distribution.

New in version 2.4.0.

**static test** *(dataset, sampleCol, distName, *params)*

Conduct a one-sample, two-sided Kolmogorov-Smirnov test for probability distribution equality. Currently supports the normal distribution, taking as parameters the mean and standard deviation.

**Parameters**

- **dataset** – a Dataset or a DataFrame containing the sample of data to test.
- **sampleCol** – Name of sample column in dataset, of any numerical type.
- **distName** – a string name for a theoretical distribution, currently only support “norm”.
- **params** – a list of Double values specifying the parameters to be used for the theoretical distribution. For “norm” distribution, the parameters includes mean and variance.
Returns A DataFrame that contains the Kolmogorov-Smirnov test result for the input sampled data. This DataFrame will contain a single Row with the following fields:
- `pValue`: Double - `statistic`: Double

```python
>>> from pyspark.ml.stat import KolmogorovSmirnovTest
>>> dataset = [[-1.0], [0.0], [1.0]]
>>> dataset = spark.createDataFrame(dataset, ['sample'])
>>> ksResult = KolmogorovSmirnovTest.test(dataset, 'sample', 'norm', 0.0, 1.0).first()
>>> round(ksResult.pValue, 3)
1.0
>>> round(ksResult.statistic, 3)
0.175
>>> dataset = [[2.0], [3.0], [4.0]]
>>> dataset = spark.createDataFrame(dataset, ['sample'])
>>> ksResult = KolmogorovSmirnovTest.test(dataset, 'sample', 'norm', 3.0, 1.0).first()
>>> round(ksResult.pValue, 3)
1.0
>>> round(ksResult.statistic, 3)
0.175
```

New in version 2.4.0.

```python
class pyspark.ml.stat.Summarizer
```

Note: Experimental

Tools for vectorized statistics on MLlib Vectors. The methods in this package provide various statistics for Vectors contained inside DataFrames. This class lets users pick the statistics they would like to extract for a given column.

```python
>>> from pyspark.ml.stat import Summarizer
>>> from pyspark.sql import Row
>>> from pyspark.ml.linalg import Vectors

>>> summarizer = Summarizer.metrics("mean", "count")
>>> df = sc.parallelize([Row(weight=1.0, features=Vectors.dense(1.0, 1.0, 1.0))])
... .toDF()
>>> df.select(summarizer.summary(df.features, df.weight)).show(truncate=False)
```

(continues on next page)
<table>
<thead>
<tr>
<th>aggregate_metrics(features, 1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[1.0,1.5,2.0], 2]</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
</tbody>
</table>

```python
>>> df.select(Summarizer.mean(df.features, df.weight)).show(truncate=False)
```

```text
+--------------+  
|mean(features) |  
+--------------+  
|[1.0,1.0,1.0] |  
+--------------+  
```

```python
>>> df.select(Summarizer.mean(df.features)).show(truncate=False)
```

```text
+--------------+  
|mean(features) |  
+--------------+  
|[1.0,1.5,2.0] |  
+--------------+  
```

New in version 2.4.0.

**static count** *(col, weightCol=None)*

return a column of count summary

New in version 2.4.0.

**static max** *(col, weightCol=None)*

return a column of max summary

New in version 2.4.0.

**static mean** *(col, weightCol=None)*

return a column of mean summary

New in version 2.4.0.

**static metrics(**metrics**)*

Given a list of metrics, provides a builder that it turns computes metrics from a column.

See the documentation of [[Summarizer]] for an example.

The following metrics are accepted (case sensitive):

- mean: a vector that contains the coefficient-wise mean.
- variance: a vector that contains the coefficient-wise variance.
- count: the count of all vectors seen.
- numNonzeros: a vector with the number of non-zeros for each coefficients
- max: the maximum for each coefficient.
- min: the minimum for each coefficient.
Learning Apache Spark with Python

- normL2: the Euclidian norm for each coefficient.
- normL1: the L1 norm of each coefficient (sum of the absolute values).

**Parameters** `metrics` – metrics that can be provided.

**Returns** an object of `pyspark.ml.stat.SummaryBuilder`

Note: Currently, the performance of this interface is about 2x~3x slower then using the RDD interface.

New in version 2.4.0.

```python
static min(col, weightCol=None)
return a column of min summary
New in version 2.4.0.
```

```python
static normL1(col, weightCol=None)
return a column of normL1 summary
New in version 2.4.0.
```

```python
static normL2(col, weightCol=None)
return a column of normL2 summary
New in version 2.4.0.
```

```python
static numNonZeros(col, weightCol=None)
return a column of numNonZero summary
New in version 2.4.0.
```

```python
static variance(col, weightCol=None)
return a column of variance summary
New in version 2.4.0.
```

```python
class pyspark.ml.stat.SummaryBuilder(jSummaryBuilder)
```

**Note:** Experimental

A builder object that provides summary statistics about a given column.

Users should not directly create such builders, but instead use one of the methods in `pyspark.ml.stat.Summarizer`

New in version 2.4.0.

```python
summary(featuresCol, weightCol=None)
Returns an aggregate object that contains the summary of the column with the requested metrics.
```

**Parameters**

- `featuresCol` – a column that contains features Vector object.
• **weightCol** – a column that contains weight value. Default weight is 1.0.

**Returns** an aggregate column that contains the statistics. The exact content of this structure is determined during the creation of the builder.

New in version 2.4.0.

### 25.2 Regression API

```python
class pyspark.ml.regression.AFTSurvivalRegression (featuresCol='features', labelCol='label', predictionCol='prediction', fitIntercept=True, maxIter=100, tol=1e-06, censorCol='censor', quantileProbabilities=[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99], quantilesCol=None, aggregationDepth=2)
```

**Note:** Experimental

Accelerated Failure Time (AFT) Model Survival Regression

Fit a parametric AFT survival regression model based on the Weibull distribution of the survival time.

**See also:**

AFT Model

```python
>>> from pyspark.ml.linalg import Vectors
>>> df = spark.createDataFrame([
... (1.0, Vectors.dense(1.0), 1.0),
... (1e-40, Vectors.sparse(1, [], []), 0.0)], ["label", "features", 
... "censor"])
>>> aftsr = AFTSurvivalRegression()
>>> model = aftsr.fit(df)
>>> model.predict(Vectors.dense(6.3))
1.0
>>> model.predictQuantiles(Vectors.dense(6.3))
DenseVector([0.0101, 0.0513, 0.1054, 0.2877, 0.6931, 1.3863, 2.3026, 2.9957, 4.6052])
>>> model.transform(df).show()
+-------+---------+------+----------+
| label | features|censor|prediction|
|-------+---------+------+----------+
| 1.0   | [1.0]   | 1.0  | 1.0       |
| 1.0E-40| (1,[],[])| 0.0  | 1.0       |
(continues on next page)
```
```python
>>> aftsr_path = temp_path + "/aftsr"
>>> aftsr.save(aftsr_path)
>>> aftsr2 = AFTSurvivalRegression.load(aftsr_path)
>>> aftsr2.getMaxIter()
100
>>> model_path = temp_path + "/aftsr_model"
>>> model.save(model_path)
>>> model2 = AFTSurvivalRegressionModel.load(model_path)
>>> model.coefficients == model2.coefficients
True
>>> model.intercept == model2.intercept
True
>>> model.scale == model2.scale
True
```

New in version 1.6.0.

**getCensorCol()**

Gets the value of `censorCol` or its default value.

New in version 1.6.0.

**getQuantileProbabilities()**

Gets the value of `quantileProbabilities` or its default value.

New in version 1.6.0.

**getQuantilesCol()**

Gets the value of `quantilesCol` or its default value.

New in version 1.6.0.

**setCensorCol(value)**

Sets the value of `censorCol`.

New in version 1.6.0.

**setParams**

```
setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction", fitIntercept=True, maxIter=100, tol=1e-06, censorCol="censor", quantileProbabilities=[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99], quantilesCol=None, aggregationDepth=2):
```

New in version 1.6.0.

**setQuantileProbabilities(value)**

Sets the value of `quantileProbabilities`.

New in version 1.6.0.
**setQuantilesCol**(*value*)
Sets the value of quantilesCol.
New in version 1.6.0.

```python
class pyspark.ml.regression.AFTSurvivalRegressionModel (java_model=None)
```

**Note:** Experimental

Model fitted by `AFTSurvivalRegression`.
New in version 1.6.0.

**coefficients**
Model coefficients.
New in version 2.0.0.

**intercept**
Model intercept.
New in version 1.6.0.

**predict**(*features*)
Predicted value
New in version 2.0.0.

**predictQuantiles**(*features*)
Predicted Quantiles
New in version 2.0.0.

**scale**
Model scale parameter.
New in version 1.6.0.

```python
class pyspark.ml.regression.DecisionTreeRegressor(featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, checkpointInterval=10, impurity='variance', seed=None, varianceCol=None)
```

Decision tree learning algorithm for regression. It supports both continuous and categorical features.
>>> from pyspark.ml.linalg import Vectors
>>> df = spark.createDataFrame([
...     (1.0, Vectors.dense(1.0)),
...     (0.0, Vectors.sparse(1, [], []))], ["label", "features"])
>>> dt = DecisionTreeRegressor(maxDepth=2, varianceCol="variance")
>>> model = dt.fit(df)
>>> model.depth
1
>>> model.numNodes
3
>>> model.featureImportances
SparseVector(1, {0: 1.0})
>>> model.numFeatures
1
>>> test0 = spark.createDataFrame([Vectors.dense(-1.0)], ["features"])
>>> model.transform(test0).head().prediction
0.0
>>> test1 = spark.createDataFrame([Vectors.sparse(1, [0], [1.0])], ["features"])
>>> model.transform(test1).head().prediction
1.0
>>> dtr_path = temp_path + "/dtr"
>>> dt.save(dtr_path)
>>> dt2 = DecisionTreeRegressor.load(dtr_path)
>>> dt2.getMaxDepth()
2
>>> model_path = temp_path + "/dtr_model"
>>> model.save(model_path)
>>> model2 = DecisionTreeRegressionModel.load(model_path)
>>> model.numNodes == model2.numNodes
True
>>> model.depth == model2.depth
True
>>> model.transform(test1).head().variance
0.0

New in version 1.4.0.

setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction",
maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0,
maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity="variance", seed=None, varianceCol=None)

Sets params for the DecisionTreeRegressor.

New in version 1.4.0.

class pyspark.ml.regression.DecisionTreeRegressionModel(java_model=None)

Model fitted by DecisionTreeRegressor.

New in version 1.4.0.

featureImportances

Estimate of the importance of each feature.
This generalizes the idea of “Gini” importance to other losses, following the explanation of Gini importance from “Random Forests” documentation by Leo Breiman and Adele Cutler, and following the implementation from scikit-learn.

**This feature importance is calculated as follows:**

- \( \text{importance(feature } j \text{)} = \sum \text{(over nodes which split on feature } j \text{)} \text{ of the gain, where gain is scaled by the number of instances passing through node} \)
- Normalize importances for tree to sum to 1.

**Note:** Feature importance for single decision trees can have high variance due to correlated predictor variables. Consider using a `RandomForestRegressor` to determine feature importance instead.

New in version 2.0.0.

```python
class pyspark.ml.regression.GBTRegressor (featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, subsamplingRate=1.0, checkpointInterval=10, lossType='squared', maxIter=20, stepSize=0.1, seed=None, impurity='variance', featureSubsetStrategy='all')
```

Gradient-Boosted Trees (GBTs) learning algorithm for regression. It supports both continuous and categorical features.

```python
>>> from numpy import allclose
>>> from pyspark.ml.linalg import Vectors

>>> df = spark.createDataFrame([...
     (1.0, Vectors.dense(1.0)),
     ...
     (0.0, Vectors.sparse(1, [], [])),
], ['label', 'features'])

>>> gbt = GBTRegressor(maxIter=5, maxDepth=2, seed=42)

>>> print(gbt.getImpurity())
variance

>>> print(gbt.getFeatureSubsetStrategy())
all

>>> model = gbt.fit(df)

>>> model.featureImportances
SparseVector(1, {0: 1.0})

>>> model.numFeatures
1

>>> allclose(model.treeWeights, [1.0, 0.1, 0.1, 0.1, 0.1])
True

>>> test0 = spark.createDataFrame([[(Vectors.dense(-1.0)),], ['features']])

>>> model.transform(test0).head().prediction
0.0
```

(continues on next page)
>>> test1 = spark.createDataFrame([(Vectors.sparse(1, [0], [1.0]),)], ["features"]
>>> model.transform(test1).head().prediction
1.0
>>> gbtr_path = temp_path + "gbtr"
>>> gbt.save(gbtr_path)
>>> gbt2 = GBTRegressor.load(gbtr_path)
>>> gbt2.getMaxDepth()
2
>>> model_path = temp_path + "gbtr_model"
>>> model.save(model_path)
>>> model2 = GBTRegressionModel.load(model_path)
>>> model.featureImportances == model2.featureImportances
True
>>> model.treeWeights == model2.treeWeights
True
>>> model.trees
[DecisionTreeRegressionModel (uid=...) of depth...,
  DecisionTreeRegressionModel...]
>>> validation = spark.createDataFrame([(0.0, Vectors.dense(-1.0))],
  ... ["label", "features"]
>>> model.evaluateEachIteration(validation, "squared")
[0.0, 0.0, 0.0, 0.0, 0.0]

New in version 1.4.0.

getLossType()
  Gets the value of lossType or its default value.

  New in version 1.4.0.

setFeatureSubsetStrategy(value)
  Sets the value of featureSubsetStrategy.

  New in version 2.4.0.

setLossType(value)
  Sets the value of lossType.

  New in version 1.4.0.

setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction",
  maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0,
  maxMemoryInMB=256, cacheNodeIds=False, subsamplingRate=1.0, checkPointInterval=10, lossType="squared", maxIter=20, stepSize=0.1, seed=None,
  impurity="variance", featureSubsetStrategy="all")

  Sets params for Gradient Boosted Tree Regression.

  New in version 1.4.0.

class pyspark.ml.regression.GBTRegressionModel(java_model=None)
  Model fitted by GBTRegressor.

  New in version 1.4.0.
evaluateEachIteration(dataset, loss)
Method to compute error or loss for every iteration of gradient boosting.

Parameters

- **dataset** – Test dataset to evaluate model on, where dataset is an instance of `pyspark.sql.DataFrame`
- **loss** – The loss function used to compute error. Supported options: squared, absolute

New in version 2.4.0.

featureImportances
Estimate of the importance of each feature.

Each feature’s importance is the average of its importance across all trees in the ensemble. The importance vector is normalized to sum to 1. This method is suggested by Hastie et al. (Hastie, Tibshirani, Friedman. “The Elements of Statistical Learning, 2nd Edition.” 2001.) and follows the implementation from scikit-learn.

See also:

`DecisionTreeRegressionModel.featureImportances`

New in version 2.0.0.

trees
These have null parent Estimators.

New in version 2.0.0.

**Type** Trees in this ensemble. Warning
class pyspark.ml.regression.GeneralizedLinearRegression(labelCol='label', featuresCol='features', predictionCol='prediction', family='gaussian', link=None, fitIntercept=True, maxIter=25, tol=1e-06, regParam=0.0, weightCol=None, solver='irls', linkPredictionCol=None, variancePower=0.0, linkPower=None, offsetCol=None)

Note: Experimental

Generalized Linear Regression.

Fit a Generalized Linear Model specified by giving a symbolic description of the linear predictor (link function) and a description of the error distribution (family). It supports “gaussian”, “binomial”, “poisson”, “gamma” and “tweedie” as family. Valid link functions for each family is listed below. The first link function of each family is the default one.

- “gaussian” -> “identity”, “log”, “inverse”
- “binomial” -> “logit”, “probit”, “cloglog”
- “poisson” -> “log”, “identity”, “sqrt”
- “gamma” -> “inverse”, “identity”, “log”
- “tweedie” -> power link function specified through “linkPower”. The default link power in the tweedie family is 1 - variancePower.

See also:

GLM

```python
>>> from pyspark.ml.linalg import Vectors
>>> df = spark.createDataFrame([
...   (1.0, Vectors.dense(0.0, 0.0)),
...   (1.0, Vectors.dense(1.0, 2.0)),
...])
```


```python
... (2.0, Vectors.dense(0.0, 0.0)),
... (2.0, Vectors.dense(1.0, 1.0)), ["label", "features"]

>>> glr = GeneralizedLinearRegression(family="gaussian", link="identity",
    linkPredictionCol="p")

>>> model = glr.fit(df)

>>> transformed = model.transform(df)

>>> abs(transformed.head().prediction - 1.5) < 0.001
True

>>> abs(transformed.head().p - 1.5) < 0.001
True

>>> model.coefficients
DenseVector([1.5..., -1.0...])

>>> model.numFeatures
2

>>> abs(model.intercept - 1.5) < 0.001
True

>>> glr_path = temp_path + "/glr"

>>> glr.save(glr_path)

>>> glr2 = GeneralizedLinearRegression.load(glr_path)

>>> glr.getFamily() == glr2.getFamily()
True

>>> model_path = temp_path + "/glr_model"

>>> model.save(model_path)

>>> model2 = GeneralizedLinearRegressionModel.load(model_path)

>>> model.intercept == model2.intercept
True

>>> model.coefficients[0] == model2.coefficients[0]
True
```

New in version 2.0.0.

**getFamily()**

Gets the value of family or its default value.

New in version 2.0.0.

**getLink()**

Gets the value of link or its default value.

New in version 2.0.0.

**getLinkPower()**

Gets the value of linkPower or its default value.

New in version 2.2.0.

**getLinkPredictionCol()**

Gets the value of linkPredictionCol or its default value.

New in version 2.0.0.

**getOffsetCol()**

Gets the value of offsetCol or its default value.

New in version 2.3.0.
getVariancePower()  
Gets the value of variancePower or its default value.  
New in version 2.2.0.

setFamily(value)  
Sets the value of family.  
New in version 2.0.0.

setLink(value)  
Sets the value of link.  
New in version 2.0.0.

setLinkPower(value)  
Sets the value of linkPower.  
New in version 2.2.0.

setLinkPredictionCol(value)  
Sets the value of linkPredictionCol.  
New in version 2.0.0.

setOffsetCol(value)  
Sets the value of offsetCol.  
New in version 2.3.0.

setParams(self, labelCol="label", featuresCol="features", predictionCol="prediction", family="gaussian", link=None, fitIntercept=True, maxIter=25, tol=1e-6, regParam=0.0, weightCol=None, solver="irls", linkPredictionCol=None, variancePower=0.0, linkPower=None, offsetCol=None)  
Sets params for generalized linear regression.  
New in version 2.0.0.

setVariancePower(value)  
Sets the value of variancePower.  
New in version 2.2.0.

Note: Experimental

Model fitted by GeneralizedLinearRegression.  
New in version 2.0.0.

coefficients  
Model coefficients.  
New in version 2.0.0.
evaluate *(dataset)*

Evaluates the model on a test dataset.

**Parameters**

- **dataset** – Test dataset to evaluate model on, where dataset is an instance of `pyspark.sql.DataFrame`

New in version 2.0.0.

**hasSummary**

Indicates whether a training summary exists for this model instance.

New in version 2.0.0.

**intercept**

Model intercept.

New in version 2.0.0.

**summary**

Gets summary (e.g. residuals, deviance, pValues) of model on training set. An exception is thrown if `trainingSummary is None`.

New in version 2.0.0.

```python
class pyspark.ml.regression.GeneralizedLinearRegressionSummary(java_obj=None)
```

**Note:** Experimental

Generalized linear regression results evaluated on a dataset.

New in version 2.0.0.

**aic**

Akaike’s “An Information Criterion”(AIC) for the fitted model.

New in version 2.0.0.

**degreesOfFreedom**

Degrees of freedom.

New in version 2.0.0.

**deviance**

The deviance for the fitted model.

New in version 2.0.0.

**dispersion**

The dispersion of the fitted model. It is taken as 1.0 for the “binomial” and “poisson” families, and otherwise estimated by the residual Pearson’s Chi-Squared statistic (which is defined as sum of the squares of the Pearson residuals) divided by the residual degrees of freedom.

New in version 2.0.0.
nullDeviance
The deviance for the null model.
New in version 2.0.0.

numInstances
Number of instances in DataFrame predictions.
New in version 2.2.0.

predictionCol
Field in predictions which gives the predicted value of each instance. This is set to a new column name if the original model’s predictionCol is not set.
New in version 2.0.0.

predictions
Predictions output by the model’s transform method.
New in version 2.0.0.

rank
The numeric rank of the fitted linear model.
New in version 2.0.0.

residualDegreeOfFreedom
The residual degrees of freedom.
New in version 2.0.0.

residualDegreeOfFreedomNull
The residual degrees of freedom for the null model.
New in version 2.0.0.

residuals (residualsType='deviance')
Get the residuals of the fitted model by type.

Parameters residualsType – The type of residuals which should be returned.
Supported options: deviance (default), pearson, working, and response.
New in version 2.0.0.

class pyspark.ml.regression.GeneralizedLinearRegressionTrainingSummary (java_obj=None)

Note: Experimental

Generalized linear regression training results.
New in version 2.0.0.

coefficientStandardErrors
Standard error of estimated coefficients and intercept.
If `GeneralizedLinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

New in version 2.0.0.

**numIterations**
Number of training iterations.

New in version 2.0.0.

**pValues**
Two-sided p-value of estimated coefficients and intercept.

If `GeneralizedLinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

New in version 2.0.0.

**solver**
The numeric solver used for training.

New in version 2.0.0.

**tValues**
T-statistic of estimated coefficients and intercept.

If `GeneralizedLinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

New in version 2.0.0.

```python
from pyspark.ml.regression import IsotonicRegression

ir = IsotonicRegression()
model = ir.fit(df)
test0 = spark.createDataFrame([(Vectors.dense(-1.0),)])
model.transform(test0).head().prediction
0.0
modelBoundaries
DenseVector([0.0, 1.0])
ir_path = temp_path + "\/ir"
ir.save(ir_path)
ir2 = IsotonicRegression.load(ir_path)
ir2.getIsotonic()
True
```

(continues on next page)
Learning Apache Spark with Python

>>> model_path = temp_path + "/ir_model"
>>> model.save(model_path)
>>> model2 = IsotonicRegressionModel.load(model_path)
>>> model.boundaries == model2.boundaries
True
>>> model.predictions == model2.predictions
True

New in version 1.6.0.

goingFeatureIndex()
    Gets the value of featureIndex or its default value.

goingIsotonic()
    Gets the value of isotonic or its default value.

setFeatureIndex(value)
    Sets the value of featureIndex.

setIsotonic(value)
    Sets the value of isotonic.

setParams(featuresCol='features', labelCol='label', predictionCol='prediction', weightCol=None, isotonic=True, featureIndex=0)
    setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction", weightCol=None, isotonic=True, featureIndex=0): Set the params for IsotonicRegression.

class pyspark.ml.regression.IsotonicRegressionModel(java_model=None)
    Model fitted by IsotonicRegression.

    New in version 1.6.0.

boundaries
    Boundaries in increasing order for which predictions are known.

    New in version 1.6.0.

predictions
    Predictions associated with the boundaries at the same index, monotone because of isotonic regression.

    New in version 1.6.0.

class pyspark.ml.regression.LinearRegression(featuresCol='features', labelCol='label', predictionCol='prediction', maxIter=100, regParam=0.0, elasticNetParam=0.0, tol=1e-06, fitIntercept=True, standardization=True, solver='auto', weightCol=None, aggregationDepth=2, loss='squaredError', epsilon=1.35)

    Linear regression.
The learning objective is to minimize the specified loss function, with regularization. This supports two kinds of loss:

- `squaredError` (a.k.a. squared loss)
- `huber` (a hybrid of squared error for relatively small errors and absolute error for relatively large ones, and we estimate the scale parameter from training data)

This supports multiple types of regularization:

- `none` (a.k.a. ordinary least squares)
- `L2` (ridge regression)
- `L1` (Lasso)
- `L2 + L1` (elastic net)

Note: Fitting with huber loss only supports none and L2 regularization.

```python
>>> from pyspark.ml.linalg import Vectors

>>> df = spark.createDataFrame([  
...     (1.0, 2.0, Vectors.dense(1.0)),  
...     (0.0, 2.0, Vectors.sparse(1, [], []))], ["label", "weight",  
...     "features"])

>>> lr = LinearRegression(maxIter=5, regParam=0.0, solver="normal",  
...     weightCol="weight")

>>> model = lr.fit(df)

>>> test0 = spark.createDataFrame([(Vectors.dense(-1.0),)], ["features"])

>>> abs(model.transform(test0).head().prediction - (-1.0)) < 0.001  
True

>>> abs(model.coefficients[0] - 1.0) < 0.001
True

>>> abs(model.intercept - 0.0) < 0.001
True

>>> test1 = spark.createDataFrame([(Vectors.sparse(1, [0], [1.0]),)], [  
...     "features"])

>>> abs(model.transform(test1).head().prediction - 1.0) < 0.001
True

>>> lr.setParams("vector")

Traceback (most recent call last):
...

TypeError: Method setParams forces keyword arguments.

>>> lr_path = temp_path + "/lr"

>>> lr.save(lr_path)

>>> lr2 = LinearRegression.load(lr_path)

>>> lr2.getMaxIter()

5

>>> model_path = temp_path + "/lr_model"

>>> model.save(model_path)

>>> model2 = LinearRegressionModel.load(model_path)

>>> model.coefficients[0] == model2.coefficients[0]
True

>>> model.intercept == model2.intercept
True
```

(continues on next page)
```python
>>> model.numFeatures
1
>>> model.write().format("pmml").save(model_path + "_2")
```

New in version 1.4.0.

**getEpsilon()**

Gets the value of epsilon or its default value.

New in version 2.3.0.

**setEpsilon(value)**

Sets the value of epsilon.

New in version 2.3.0.

**setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100, regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True, standardization=True, solver="auto", weightCol=None, aggregationDepth=2, loss="squaredError", epsilon=1.35)**

Sets params for linear regression.

New in version 1.4.0.

**class pyspark.ml.regression.LinearRegressionModel(java_model=None)**

Model fitted by `LinearRegression`.

New in version 1.4.0.

**coefficients**

Model coefficients.

New in version 2.0.0.

**evaluate(dataset)**

Evaluates the model on a test dataset.

**Parameters dataset** – Test dataset to evaluate model on, where dataset is an instance of `pyspark.sql.DataFrame`

New in version 2.0.0.

**hasSummary**

Indicates whether a training summary exists for this model instance.

New in version 2.0.0.

**intercept**

Model intercept.

New in version 1.4.0.

**scale**

The value by which \( \| y - X'w \| \) is scaled down when loss is “huber”, otherwise 1.0.

New in version 2.3.0.
**summary**
Gets summary (e.g. residuals, mse, r-squared) of model on training set. An exception is thrown if `trainingSummary` is None.

New in version 2.0.0.

```python
class pyspark.ml.regression.LinearRegressionSummary(java_obj=None)
```

**Note:** Experimental

Linear regression results evaluated on a dataset.

New in version 2.0.0.

**coefficientStandardErrors**
Standard error of estimated coefficients and intercept. This value is only available when using the “normal” solver.

If `LinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

**See also:**
`LinearRegression.solver`

New in version 2.0.0.

**degreesOfFreedom**
Degrees of freedom.

New in version 2.2.0.

**devianceResiduals**
The weighted residuals, the usual residuals rescaled by the square root of the instance weights.

New in version 2.0.0.

**explainedVariance**
Returns the explained variance regression score. \( \text{explainedVariance} = 1 - \frac{\text{variance}(y - \hat{y})}{\text{variance}(y)} \)

**See also:**
Wikipedia explain variation

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.

New in version 2.0.0.

**featuresCol**
Field in “predictions” which gives the features of each instance as a vector.

New in version 2.0.0.
**labelCol**
Field in “predictions” which gives the true label of each instance.
New in version 2.0.0.

**meanAbsoluteError**
Returns the mean absolute error, which is a risk function corresponding to the expected value of the absolute error loss or $L_1$-norm loss.

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.
New in version 2.0.0.

**meanSquaredError**
Returns the mean squared error, which is a risk function corresponding to the expected value of the squared error loss or quadratic loss.

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.
New in version 2.0.0.

**numInstances**
Number of instances in DataFrame predictions
New in version 2.0.0.

**pValues**
Two-sided p-value of estimated coefficients and intercept. This value is only available when using the “normal” solver.

If `LinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

**See also:**
`LinearRegression.solver`
New in version 2.0.0.

**predictionCol**
Field in “predictions” which gives the predicted value of the label at each instance.
New in version 2.0.0.

**predictions**
Dataframe outputted by the model’s `transform` method.
New in version 2.0.0.

**r2**
Returns $R^2$, the coefficient of determination.
See also:

Wikipedia coefficient of determination

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.

New in version 2.0.0.

**r2adj**

Returns Adjusted $R^2$, the adjusted coefficient of determination.

**See also:**

Wikipedia coefficient of determination, Adjusted $R^2$

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.

New in version 2.4.0.

**residuals**

Residuals (label - predicted value)

New in version 2.0.0.

**rootMeanSquaredError**

 Returns the root mean squared error, which is defined as the square root of the mean squared error.

**Note:** This ignores instance weights (setting all to 1.0) from `LinearRegression.weightCol`. This will change in later Spark versions.

New in version 2.0.0.

**tValues**

T-statistic of estimated coefficients and intercept. This value is only available when using the “normal” solver.

If `LinearRegression.fitIntercept` is set to True, then the last element returned corresponds to the intercept.

**See also:**

`LinearRegression.solver`

New in version 2.0.0.

**class** `pyspark.ml.regression.LinearRegressionTrainingSummary` *(java_obj=None)*
Learning Apache Spark with Python

Note: Experimental

Linear regression training results. Currently, the training summary ignores the training weights except for the objective trace.

New in version 2.0.0.

**objectiveHistory**
Objective function (scaled loss + regularization) at each iteration. This value is only available when using the “l-bfgs” solver.

See also:
LinearRegression.solver
New in version 2.0.0.

**totalIterations**
Number of training iterations until termination. This value is only available when using the “l-bfgs” solver.

See also:
LinearRegression.solver
New in version 2.0.0.

```python
class pyspark.ml.regression.RandomForestRegressor (featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity='variance', subsamplingRate=1.0, seed=None, numTrees=20, featureSubsetStrategy='auto')
```

Random Forest learning algorithm for regression. It supports both continuous and categorical features.

```python
>>> from numpy import allclose
>>> from pyspark.ml.linalg import Vectors

>>> df = spark.createDataFrame([...
... (1.0, Vectors.dense(1.0)),
... (0.0, Vectors.sparse(1, [], [])), 
["label", "features"])

>>> rf = RandomForestRegressor(numTrees=2, maxDepth=2, seed=42)

>>> model = rf.fit(df)

>>> model.featureImportances
```
(continues on next page)
SparseVector(1, {0: 1.0})
>>> allclose(model.treeWeights, [1.0, 1.0])
True
>>> test0 = spark.createDataFrame([(Vectors.dense(-1.0),)], ["features"])
>>> model.transform(test0).head().prediction
0.0
>>> model.numFeatures
1
>>> model.trees
[DecisionTreeRegressionModel (uid=...) of depth...,]
>>> model.getNumTrees
2
>>> test1 = spark.createDataFrame([(Vectors.sparse(1, [0], [1.0]),)], ["features"])
>>> model.transform(test1).head().prediction
0.5
>>> rfr_path = temp_path + "/rfr"
>>> rf.save(rfr_path)
>>> rf2 = RandomForestRegressor.load(rfr_path)
>>> rf2.getNumTrees()
2
>>> model_path = temp_path + "/rfr_model"
>>> model.save(model_path)
>>> model2 = RandomForestRegressionModel.load(model_path)
>>> model.featureImportances == model2.featureImportances
True

New in version 1.4.0.

**setFeatureSubsetStrategy** (*value*)

Sets the value of featureSubsetStrategy.

New in version 2.4.0.

**setParams** (*self*, **featuresCol**="features", **labelCol**="label", **predictionCol**="prediction", **maxDepth**=5, **maxBins**=32, **minInstancesPerNode**=1, **minInfoGain**=0.0, **maxMemoryInMB**=256, **cacheNodeIds**=False, **checkpointInterval**=10, **impurity**="variance", **subsamplingRate**=1.0, **seed**=None, **numTrees**=20, **featureSubsetStrategy**="auto")

Sets params for linear regression.

New in version 1.4.0.

**class** pyspark.ml.regression.RandomForestRegressionModel (*java_model*=None)

Model fitted by RandomForestRegressor.

New in version 1.4.0.

**featureImportances**

Estimate of the importance of each feature.

Each feature’s importance is the average of its importance across all trees in the ensemble. The importance vector is normalized to sum to 1. This method is suggested by Hastie et al. (Hastie,
the implementation from scikit-learn.

See also:

```
DecisionTreeRegressionModel.featureImportances
```

New in version 2.0.0.

**trees**

These have null parent Estimators.

New in version 2.0.0.

**Type** Trees in this ensemble. Warning

### 25.3 Classification API

```python
class pyspark.ml.classification.LinearSVC(featuresCol='features', labelCol='label', predictionCol='prediction', maxIter=100, regParam=0.0, tol=1e-06, rawPredictionCol='rawPrediction', fitIntercept=True, standardization=True, threshold=0.0, weightCol=None, aggregationDepth=2)
```

**Note:** Experimental

Linear SVM Classifier

This binary classifier optimizes the Hinge Loss using the OWLQN optimizer. Only supports L2
regularization currently.

```python
>>> from pyspark.sql import Row
>>> from pyspark.ml.linalg import Vectors

>>> df = sc.parallelize([...
                          Row(label=1.0, features=Vectors.dense(1.0, 1.0, 1.0)),
                          Row(label=0.0, features=Vectors.dense(1.0, 2.0, 3.0))]).toDF()
>>> svm = LinearSVC(maxIter=5, regParam=0.01)
>>> model = svm.fit(df)
>>> model.coefficients
DenseVector([0.0, -0.2792, -0.1833])
>>> model.intercept
1.0206118982229047
>>> model.numClasses
2
>>> model.numFeatures
3
>>> test0 = sc.parallelize([Row(features=Vectors.dense(-1.0, -1.0, -1.0))]).toDF()
```

(continues on next page)
```python
>>> result = model.transform(test0).head()
>>> result.prediction
1.0
>>> result.rawPrediction
DenseVector([-1.4831, 1.4831])
>>> svm_path = temp_path + "\svm"
>>> svm.save(svm_path)
>>> svm2 = LinearSVC.load(svm_path)
>>> svm2.getMaxIter()
5
>>> model_path = temp_path + "\svm_model"
>>> model.save(model_path)
>>> model2 = LinearSVCModel.load(model_path)
>>> model.coefficients[0] == model2.coefficients[0]
True
>>> model.intercept == model2.intercept
True
```

New in version 2.2.0.

**setParams**

```python
setParams(self, featuresCol=’features’, labelCol=’label’, predictionCol=’prediction’, maxIter=100, regParam=0.0, tol=1e-06, rawPredictionCol=’rawPrediction’, fitIntercept=True, standardization=True, threshold=0.0, weightCol=None, aggregationDepth=2)
```

setParams(self, featuresCol=’features’, labelCol=’label’, predictionCol=’prediction’, maxIter=100, regParam=0.0, tol=1e-06, rawPredictionCol=’rawPrediction’, fitIntercept=True, standardization=True, threshold=0.0, weightCol=None, aggregationDepth=2): Sets params for Linear SVM Classifier.

New in version 2.2.0.

**class**

```python
class pyspark.ml.classification.LinearSVCModel(java_model=None)
```

**Note:** Experimental

Model fitted by LinearSVC.

New in version 2.2.0.

**coefficients**

Model coefficients of Linear SVM Classifier.

New in version 2.2.0.

**intercept**

Model intercept of Linear SVM Classifier.

New in version 2.2.0.
Logistic regression. This class supports multinomial logistic (softmax) and binomial logistic regression.

```
>>> from pyspark.ml.linalg import Vectors

>>> bdf = sc.parallelize([(1.0, 0.0, 0.0, 5.0)),
                        (0.0, 2.0, 1.0, 2.0),
                        (1.0, 3.0, 2.0, 1.0),
                        (0.0, 4.0, 3.0, 3.0))].
  toDF()
>>> blor = LogisticRegression(regParam=0.01, weightCol="weight")
>>> blorModel = blor.fit(bdf)
>>> blorModel.coefficients
DenseVector([-1.080..., -0.646...])
>>> blorModel.intercept
3.112...
```

(continues on next page)
DenseVector([0.04..., -0.42..., 0.37...])

```python
>>> test0 = sc.parallelize([Row(features=Vectors.dense(-1.0, 1.0))]).
    toDF()
>>> result = blorModel.transform(test0).head()
>>> result.prediction
1.0
>>> result.probability
DenseVector([0.02..., 0.97...])
>>> result.rawPrediction
DenseVector([-3.54..., 3.54...])
```

```python
>>> test1 = sc.parallelize([Row(features=Vectors.sparse(2, [0], [1.0]))]).toDF()
>>> blorModel.transform(test1).head().prediction
1.0
```

```python
>>> blor.setParams("vector")
Traceback (most recent call last):
...
TypeError: Method setParams forces keyword arguments.
```

```python
>>> lr_path = temp_path + "/lr"
>>> blor.save(lr_path)
>>> lr2 = LogisticRegression.load(lr_path)
>>> lr2.getRegParam()
0.01
>>> model_path = temp_path + "/lr_model"
>>> blorModel.save(model_path)
>>> model2 = LogisticRegressionModel.load(model_path)
>>> blorModel.coefficients[0] == model2.coefficients[0]
True
>>> blorModel.intercept == model2.intercept
True
>>> model2
LogisticRegressionModel: uid = ..., numClasses = 2, numFeatures = 2
```

New in version 1.3.0.

```python
getFamily()
```

Gets the value of family or its default value.

New in version 2.1.0.

```python
getLowerBoundsOnCoefficients()
```

Gets the value of lowerBoundsOnCoefficients

New in version 2.3.0.

```python
getLowerBoundsOnIntercepts()
```

Gets the value of lowerBoundsOnIntercepts

New in version 2.3.0.

```python
getThreshold()
```

Get threshold for binary classification.

If `thresholds` is set with length 2 (i.e., binary classification), this returns the equivalent
threshold: $\frac{1}{1 + \text{thresholds}(0)}$. Otherwise, returns \text{threshold} if set or its default value if unset.

New in version 1.4.0.

\textbf{getThresholds()} 
If \text{thresholds} is set, return its value. Otherwise, if \text{threshold} is set, return the equivalent thresholds for binary classification: (1-\text{threshold}, \text{threshold}). If neither are set, throw an error.

New in version 1.5.0.

\textbf{getUpperBoundsOnCoefficients()} 
Gets the value of \text{upperBoundsOnCoefficients}

New in version 2.3.0.

\textbf{getUpperBoundsOnIntercepts()} 
Gets the value of \text{upperBoundsOnIntercepts}

New in version 2.3.0.

\textbf{setFamily(\textit{value})} 
Sets the value of \text{family}.

New in version 2.1.0.

\textbf{setLowerBoundsOnCoefficients(\textit{value})} 
Sets the value of \text{lowerBoundsOnCoefficients}

New in version 2.3.0.

\textbf{setLowerBoundsOnIntercepts(\textit{value})} 
Sets the value of \text{lowerBoundsOnIntercepts}

New in version 2.3.0.

\textbf{setParams(\textit{featuresCol}=\texttt{"features"}, \textit{labelCol}=\texttt{"label"}, \textit{predictionCol}=\texttt{"prediction"}, \textit{maxIter}=100, \textit{regParam}=0.0, \textit{elasticNetParam}=0.0, \textit{tol}=1e-06, \textit{fitIntercept}=True, \textit{threshold}=0.5, \textit{thresholds}=None, \textit{probabilityCol}=\texttt{"probability"}, \textit{rawPredictionCol}=\texttt{"rawPrediction"}, \textit{standardization}=True, \textit{weightCol}=None, \textit{aggregationDepth}=2, \textit{family}=\texttt{"auto"}, \textit{lowerBoundsOnCoefficients}=None, \textit{upperBoundsOnCoefficients}=None, \textit{lowerBoundsOnIntercepts}=None, \textit{upperBoundsOnIntercepts}=None)} 
setParams(self, featuresCol=\texttt{"features"}, labelCol=\texttt{"label"}, predictionCol=\texttt{"prediction"}, maxIter=100, regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True, threshold=0.5, thresholds=None, probabilityCol="probability", rawPredictionCol="rawPrediction", standardization=True, weightCol=None, aggregationDepth=2, family="auto", lowerBoundsOnCoefficients=None, upperBoundsOnCoefficients=None, lowerBoundsOnIntercepts=None, upperBoundsOnIntercepts=None): Sets params for logistic regression. If the threshold and thresholds Params are both set, they must be equivalent.

New in version 1.3.0.

\textbf{setThreshold(\textit{value})} 
Sets the value of \text{threshold}. Clears value of \text{thresholds} if it has been set.

New in version 1.4.0.
setThresholds \((value)\)
Sets the value of thresholds. Clears value of threshold if it has been set.

New in version 1.5.0.

setUpperBoundsOnCoefficients \((value)\)
Sets the value of upperBoundsOnCoefficients

New in version 2.3.0.

setUpperBoundsOnIntercepts \((value)\)
Sets the value of upperBoundsOnIntercepts

New in version 2.3.0.

class pyspark.ml.classification.LogisticRegressionModel \((java_model=None)\)
Model fitted by LogisticRegression.

New in version 1.3.0.

coefficientMatrix
Model coefficients.

New in version 2.1.0.

coefficients
Model coefficients of binomial logistic regression. An exception is thrown in the case of multinomial logistic regression.

New in version 2.0.0.

evaluate \((dataset)\)
Evaluates the model on a test dataset.

Parameters dataset – Test dataset to evaluate model on, where dataset is an instance of pyspark.sql.DataFrame

New in version 2.0.0.

hasSummary
Indicates whether a training summary exists for this model instance.

New in version 2.0.0.

intercept
Model intercept of binomial logistic regression. An exception is thrown in the case of multinomial logistic regression.

New in version 1.4.0.

interceptVector
Model intercept.

New in version 2.1.0.

summary
Gets summary (e.g. accuracy/precision/recall, objective history, total iterations) of model trained on the training set. An exception is thrown if trainingSummaryisNone.
Abstraction for Logistic Regression Results for a given model.

**accuracy**
- Returns accuracy. (equals to the total number of correctly classified instances out of the total number of instances.)
  - New in version 2.3.0.

**fMeasureByLabel** \((\beta=1.0)\)
- Returns f-measure for each label (category).
  - New in version 2.3.0.

**falsePositiveRateByLabel**
- Returns false positive rate for each label (category).
  - New in version 2.3.0.

**featuresCol**
- Field in “predictions” which gives the features of each instance as a vector.
  - New in version 2.0.0.

**labelCol**
- Field in “predictions” which gives the true label of each instance.
  - New in version 2.0.0.

**labels**
- Returns the sequence of labels in ascending order. This order matches the order used in metrics which are specified as arrays over labels, e.g., truePositiveRateByLabel.
  - Note: In most cases, it will be values \(\{0.0, 1.0, \ldots, \text{numClasses}-1\}\). However, if the training set is missing a label, then all of the arrays over labels (e.g., from truePositiveRateByLabel) will be of length \(\text{numClasses}-1\) instead of the expected \(\text{numClasses}\).
  - New in version 2.3.0.

**precisionByLabel**
- Returns precision for each label (category).
  - New in version 2.3.0.

**predictionCol**
- Field in “predictions” which gives the prediction of each class.
  - New in version 2.3.0.
**predictions**
Dataframe outputted by the model’s *transform* method.
New in version 2.0.0.

**probabilityCol**
Field in “predictions” which gives the probability of each class as a vector.
New in version 2.0.0.

**recallByLabel**
Returns recall for each label (category).
New in version 2.3.0.

**truePositiveRateByLabel**
Returns true positive rate for each label (category).
New in version 2.3.0.

**weightedFMeasure** (*beta=1.0*)
Returns weighted averaged f-measure.
New in version 2.3.0.

**weightedFalsePositiveRate**
Returns weighted false positive rate.
New in version 2.3.0.

**weightedPrecision**
Returns weighted averaged precision.
New in version 2.3.0.

**weightedRecall**
Returns weighted averaged recall. (equals to precision, recall and f-measure)
New in version 2.3.0.

**weightedTruePositiveRate**
Returns weighted true positive rate. (equals to precision, recall and f-measure)
New in version 2.3.0.

**class** `pyspark.ml.classification.LogisticRegressionTrainingSummary` (*java_obj=None*)

**Note:** Experimental

Abstraction for multinomial Logistic Regression Training results. Currently, the training summary ignores the training weights except for the objective trace.
New in version 2.0.0.

**objectiveHistory**
Objective function (scaled loss + regularization) at each iteration.
New in version 2.0.0.

**totalIterations**
Number of training iterations until termination.
New in version 2.0.0.

class pyspark.ml.classification.BinaryLogisticRegressionSummary (java_obj=None)

---

**Note:** Experimental

Binary Logistic regression results for a given model.
New in version 2.0.0.

**areaUnderROC**
Computes the area under the receiver operating characteristic (ROC) curve.

---

**Note:** This ignores instance weights (setting all to 1.0) from LogisticRegression.weightCol. This will change in later Spark versions.

New in version 2.0.0.

**fMeasureByThreshold**
Returns a dataframe with two fields (threshold, F-Measure) curve with beta = 1.0.

---

**Note:** This ignores instance weights (setting all to 1.0) from LogisticRegression.weightCol. This will change in later Spark versions.

New in version 2.0.0.

**pr**
Returns the precision-recall curve, which is a Dataframe containing two fields recall, precision with (0.0, 1.0) prepended to it.

---

**Note:** This ignores instance weights (setting all to 1.0) from LogisticRegression.weightCol. This will change in later Spark versions.

New in version 2.0.0.

**precisionByThreshold**
Returns a dataframe with two fields (threshold, precision) curve. Every possible probability obtained in transforming the dataset are used as thresholds used in calculating the precision.

---

**Note:** This ignores instance weights (setting all to 1.0) from LogisticRegression.weightCol.
This will change in later Spark versions.

New in version 2.0.0.

**recallByThreshold**

Returns a dataframe with two fields (threshold, recall) curve. Every possible probability obtained in transforming the dataset are used as thresholds used in calculating the recall.

**Note:** This ignores instance weights (setting all to 1.0) from `LogisticRegression.weightCol`. This will change in later Spark versions.

New in version 2.0.0.

**roc**

Returns the receiver operating characteristic (ROC) curve, which is a Dataframe having two fields (FPR, TPR) with (0.0, 0.0) prepended and (1.0, 1.0) appended to it.

**See also:**

Wikipedia reference

**Note:** This ignores instance weights (setting all to 1.0) from `LogisticRegression.weightCol`. This will change in later Spark versions.

New in version 2.0.0.

**class** `pyspark.ml.classification.BinaryLogisticRegressionTrainingSummary (java_obj=None)`

**Note:** Experimental

Binary Logistic regression training results for a given model.

New in version 2.0.0.
class pyspark.ml.classification.DecisionTreeClassifier (featuresCol='features', labelCol='label', predictionCol='prediction', probabilityCol='probability', rawPredictionCol='rawPrediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheModelIds=False, checkpointInterval=10, impurity='gini', seed=None)

Decision tree learning algorithm for classification. It supports both binary and multiclass labels, as well as both continuous and categorical features.

```python
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.feature import StringIndexer

>>> df = spark.createDataFrame([
... (1.0, Vectors.dense(1.0)),
... (0.0, Vectors.sparse(1, [], [])),
], ["label", "features"])

>>> stringIndexer = StringIndexer(inputCol="label", outputCol="indexed")

>>> si_model = stringIndexer.fit(df)

>>> td = si_model.transform(df)

>>> dt = DecisionTreeClassifier(maxDepth=2, labelCol="indexed")

>>> model = dt.fit(td)

>>> model.numNodes
3

>>> model.depth
1

>>> model.featureImportances
SparseVector(1, {0: 1.0})

>>> model.numFeatures
1

>>> model.numClasses
2

>>> print(model.toDebugString)
DecisionTreeClassificationModel (uid=...) of depth 1 with 3 nodes...

>>> test0 = spark.createDataFrame([(Vectors.dense(-1.0),)], ["features"])

>>> result = model.transform(test0).head()

>>> result.prediction
0.0
```

(continues on next page)
New in version 1.4.0.

setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction",
probabilityCol="probability", rawPredictionCol="rawPrediction",
maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0,
maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity="gini", seed=None)

Sets params for the DecisionTreeClassifier.

New in version 1.4.0.

class pyspark.ml.classification.DecisionTreeClassificationModel(java_model=None)

Model fitted by DecisionTreeClassifier.

New in version 1.4.0.

featureImportances
Estimate of the importance of each feature.

This generalizes the idea of “Gini” importance to other losses, following the explanation of Gini importance from “Random Forests” documentation by Leo Breiman and Adele Cutler, and following the implementation from scikit-learn.

This feature importance is calculated as follows:

- importance(feature j) = sum (over nodes which split on feature j) of the gain, where gain is scaled by the number of instances passing through node
- Normalize importances for tree to sum to 1.

Note: Feature importance for single decision trees can have high variance due to correlated predictor variables. Consider using a RandomForestClassifier to determine feature im-
Gradient-Boosted Trees (GBTs) learning algorithm for classification. It supports binary labels, as well as both continuous and categorical features.


Notes on Gradient Boosting vs. TreeBoost: - This implementation is for Stochastic Gradient Boosting, not for TreeBoost. - Both algorithms learn tree ensembles by minimizing loss functions. - TreeBoost (Friedman, 1999) additionally modifies the outputs at tree leaf nodes based on the loss function, whereas the original gradient boosting method does not. - We expect to implement TreeBoost in the future: SPARK-4240

Note: Multiclass labels are not currently supported.
```python
>>> model.transform(test0).head().prediction
0.0
>>> test1 = spark.createDataFrame([Vectors.sparse(1, [0], [1.0]),], [ ˓→"features")
>>> model.transform(test1).head().prediction
1.0
>>> model.totalNumNodes
15
>>> print(model.toDebugString)
GBTClassificationModel (uid=...)...with 5 trees...
>>> gbtc_path = temp_path + "gbtc"
>>> gbt.save(gbtc_path)
>>> gbt2 = GBTClassifier.load(gbtc_path)
>>> gbt2.getMaxDepth()
2
>>> model_path = temp_path + "gbtc_model"
>>> model.save(model_path)
>>> model2 = GBTClassificationModel.load(model_path)
>>> model.featureImportances == model2.featureImportances
True
>>> model.treeWeights == model2.treeWeights
True
>>> model.trees
[DecisionTreeRegressionModel (uid=...) of depth..., ...
  DecisionTreeRegressionModel...]
>>> validation = spark.createDataFrame([[(0.0, Vectors.dense(-1.0)),], ...
  ["indexed", "features")
>>> model.evaluateEachIteration(validation)
[0.25..., 0.23..., 0.21..., 0.19..., 0.18...]
>>> model.numClasses
2

New in version 1.4.0.

getLossType()

Gets the value of lossType or its default value.

New in version 1.4.0.

setFeatureSubsetStrategy(value)

Sets the value of featureSubsetStrategy.

New in version 2.4.0.

setLossType(value)

Sets the value of lossType.

New in version 1.4.0.

setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction",
maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0,
maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10,
lossType="logistic", maxIter=20, stepSize=0.1, seed=None, subsamplingRate=1.0, featureSubsetStrategy="all")
```

25.3. Classification API
Sets params for Gradient Boosted Tree Classification.

New in version 1.4.0.

class pyspark.ml.classification.GBTClassificationModel(java_model=None)

Model fitted by GBTClassifier.

New in version 1.4.0.

evaluateEachIteration(dataset)
Method to compute error or loss for every iteration of gradient boosting.

Parameters dataset -- Test dataset to evaluate model on, where dataset is an instance of pyspark.sql.DataFrame

New in version 2.4.0.

featureImportances
Estimate of the importance of each feature.

Each feature’s importance is the average of its importance across all trees in the ensemble. The importance vector is normalized to sum to 1. This method is suggested by Hastie et al. (Hastie, Tibshirani, Friedman. “The Elements of Statistical Learning, 2nd Edition.” 2001.) and follows the implementation from scikit-learn.

See also:

decisionTreeClassificationModel.featureImportances
New in version 2.0.0.

trees
These have null parent Estimators.

New in version 2.0.0.

Type Trees in this ensemble. Warning
class pyspark.ml.classification.RandomForestClassifier(featuresCol='features', labelCol='label', predictionCol='prediction', probabilityCol='probability', rawPredictionCol='rawPrediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheModelIds=False, checkpointInterval=10, impurity='gini', numTrees=20, featureSubsetStrategy='auto', seed=None, subsamplingRate=1.0)

Random Forest learning algorithm for classification. It supports both binary and multiclass labels, as well as both continuous and categorical features.

```python
classified pypart
```

---

25.3. Classification API

---

(continues on next page)
New in version 1.4.0.

**setFeatureSubsetStrategy** *(value)*

Sets the value of `featureSubsetStrategy`.

New in version 2.4.0.

**setParams** *(self, featuresCol=“features”, labelCol=“label”, predictionCol=“prediction”, probabilityCol=“probability”, rawPredictionCol=“rawPrediction”, maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, seed=None, impurity=“gini”, numTrees=20, featureSubsetStrategy=“auto”, subsamplingRate=1.0)*

Sets params for linear classification.

New in version 1.4.0.

**class** `pyspark.ml.classification.RandomForestClassificationModel` *(java_model=None)*

Model fitted by RandomForestClassifier.

New in version 1.4.0.

**featureImportances**

Estimate of the importance of each feature.

Each feature’s importance is the average of its importance across all trees in the ensemble. The importance vector is normalized to sum to 1. This method is suggested by Hastie et al. (Hastie, Tibshirani, Friedman. “The Elements of Statistical Learning, 2nd Edition.” 2001.) and follows the implementation from scikit-learn.

See also:
Naive Bayes Classifiers. It supports both Multinomial and Bernoulli NB. Multinomial NB can handle finitely supported discrete data. For example, by converting documents into TF-IDF vectors, it can be used for document classification. By making every vector a binary (0/1) data, it can also be used as Bernoulli NB. The input feature values must be nonnegative.

```python
>>> from pyspark.sql import Row
>>> from pyspark.ml.linalg import Vectors

>>> df = spark.createDataFrame([... Row(label=0.0, weight=0.1, features=Vectors.dense([0.0, 0.0])), ... Row(label=0.0, weight=0.5, features=Vectors.dense([0.0, 1.0])), ... Row(label=1.0, weight=1.0, features=Vectors.dense([1.0, 0.0]))])

>>> nb = NaiveBayes(smoothing=1.0, modelType="multinomial", weightCol="weight")

>>> model = nb.fit(df)

>>> model.pi

DenseVector([-0.81..., -0.58...])

>>> model.theta

DenseMatrix(2, 2, [-0.91..., -0.51..., -0.40..., -1.09...], 1)

>>> test0 = sc.parallelize([Row(features=Vectors.dense([1.0, 0.0]))]).toDF()

>>> result = model.transform(test0).head()

>>> result.prediction

1.0

>>> result.probability

DenseVector([0.32..., 0.67...])

>>> result.rawPrediction

DenseVector([-1.72..., -0.99...])

>>> test1 = sc.parallelize([Row(features=Vectors.sparse(2, [0], [1.0]))]).toDF()

>>> model.transform(test1).head().prediction

1.0

>>> nb_path = temp_path + "/nb"

>>> nb.save(nb_path)

>>> nb2 = NaiveBayes.load(nb_path)
```

(continues on next page)
>>> nb2.getSmoothing()
1.0
>>> model_path = temp_path + "/nb_model"
>>> model.save(model_path)
>>> model2 = NaiveBayesModel.load(model_path)
>>> model.pi == model2.pi
True
>>> model.theta == model2.theta
True
>>> nb = nb.setThresholds([0.01, 10.00])
>>> model3 = nb.fit(df)
>>> result = model3.transform(test0).head()
>>> result.prediction
0.0

New in version 1.5.0.

**getModelType()**

Gets the value of modelType or its default value.

New in version 1.5.0.

**getSmoothing()**

Gets the value of smoothing or its default value.

New in version 1.5.0.

**setModelType(value)**

Sets the value of modelType.

New in version 1.5.0.

**setParams**(self, featuresCol="features", labelCol="label", predictionCol="prediction",
probabilityCol="probability", rawPredictionCol="rawPrediction", smoothing=1.0, modelType="multinomial", thresholds=None, weightCol=None)

Sets params for Naive Bayes.

New in version 1.5.0.

**setSmoothing(value)**

Sets the value of smoothing.

New in version 1.5.0.

**class** pyspark.ml.classification.NaiveBayesModel(java_model=None)

Model fitted by NaiveBayes.

New in version 1.5.0.

**pi**

log of class priors.

New in version 2.0.0.

**theta**

log of class conditional probabilities.
New in version 2.0.0.

```python
class pyspark.ml.classification.MultilayerPerceptronClassifier(featuresCol='features', labelCol='label', predictionCol='prediction', maxIter=100, tol=1e-06, seed=None, layers=None, blockSize=128, stepSize=0.03, solver='lbfgs', initialWeights=None, probabilityCol='probability', rawPredictionCol='rawPrediction')```

Classifier trainer based on the Multilayer Perceptron. Each layer has sigmoid activation function, output layer has softmax. Number of inputs has to be equal to the size of feature vectors. Number of outputs has to be equal to the total number of labels.

```python
>>> from pyspark.ml.linalg import Vectors

>>> df = spark.createDataFrame([...
    (0.0, Vectors.dense([0.0, 0.0])),
    (1.0, Vectors.dense([0.0, 1.0])),
    (1.0, Vectors.dense([1.0, 0.0])),
    (0.0, Vectors.dense([1.0, 1.0]))], ['label', 'features'])

>>> mlp = MultilayerPerceptronClassifier(maxIter=100, layers=[2, 2, 2], blockSize=1, seed=123)

>>> model = mlp.fit(df)

>>> model.layers
[2, 2, 2]
```

(continues on next page)
>>> model.weights.size
12
>>> testDF = spark.createDataFrame([...
    (Vectors.dense([1.0, 0.0]),),
    (Vectors.dense([0.0, 0.0]),)], ["features"])
>>> model.transform(testDF).select("features", "prediction").show()
+---------+----------+
<table>
<thead>
<tr>
<th>features</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.0,0.0]</td>
<td>1.0</td>
</tr>
<tr>
<td>[0.0,0.0]</td>
<td>0.0</td>
</tr>
</tbody>
</table>
+---------+----------+
...
>>> mlp_path = temp_path + "/mlp"
>>> mlp.save(mlp_path)
>>> mlp2 = MultilayerPerceptronClassifier.load(mlp_path)
>>> mlp2.getBlockSize()
1
>>> model_path = temp_path + "/mlp_model"
>>> model.save(model_path)
>>> model2 = MultilayerPerceptronClassificationModel.load(model_path)
>>> model.layers == model2.layers
True
>>> model.weights == model2.weights
True
>>> mlp2 = mlp2.setInitialWeights(list(range(0, 12)))
>>> model3 = mlp2.fit(df)
>>> model3.weights != model2.weights
True
>>> model3.layers == model.layers
True

New in version 1.6.0.

getBlockSize()  
 Gets the value of blockSize or its default value.

New in version 1.6.0.

getInitialWeights()  
 Gets the value of initialWeights or its default value.

New in version 2.0.0.

gGetLayers()  
 Gets the value of layers or its default value.

New in version 1.6.0.

gGetStepSize()  
 Gets the value of stepSize or its default value.

New in version 2.0.0.
setBlockSize(value)
Sets the value of blockSize.
New in version 1.6.0.

setInitialWeights(value)
Sets the value of initialWeights.
New in version 2.0.0.

setLayers(value)
Sets the value of layers.
New in version 1.6.0.

setParams(featuresCol='features', labelCol='label', predictionCol='prediction', maxIter=100, tol=1e-06, seed=None, layers=None, blockSize=128, stepSize=0.03, solver='l-bfgs', initialWeights=None, probabilityCol='probability', rawPredictionCol='rawPrediction')
setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100, tol=1e-6, seed=None, layers=None, blockSize=128, stepSize=0.03, solver="l-bfgs", initialWeights=None, probabilityCol="probability", rawPredictionCol="rawPrediction"): Sets params for MultilayerPerceptronClassifier.
New in version 1.6.0.

setStepSize(value)
Sets the value of stepSize.
New in version 2.0.0.

class pyspark.ml.classification.MultilayerPerceptronClassificationModel(java_model=None)
Model fitted by MultilayerPerceptronClassifier.
New in version 1.6.0.

layers
array of layer sizes including input and output layers.
New in version 1.6.0.

weights
the weights of layers.
New in version 2.0.0.

class pyspark.ml.classification.OneVsRest(featuresCol='features', labelCol='label', predictionCol='prediction', classifier=None, weightCol=None, parallelism=1)

Note: Experimental
Reduction of Multiclass Classification to Binary Classification. Performs reduction using one against all strategy. For a multiclass classification with k classes, train k models (one per class). Each example is scored against all k models and the model with highest score is picked to label the example.

```python
>>> from pyspark.sql import Row
>>> from pyspark.ml.linalg import Vectors

>>> data_path = "data/mllib/sample_multiclass_classification_data.txt"

>>> df = spark.read.format("libsvm").load(data_path)

>>> lr = LogisticRegression(regParam=0.01)

>>> ovr = OneVsRest(classifier=lr)

>>> model = ovr.fit(df)

>>> model.models[0].coefficients
DenseVector([0.5..., -1.0..., 3.4..., 4.2...])

>>> model.models[1].coefficients
DenseVector([-2.1..., 3.1..., -2.6..., -2.3...])

>>> model.models[2].coefficients
DenseVector([0.3..., -3.4..., 1.0..., -1.1...])

>>> [x.intercept for x in model.models]
[-2.7..., -2.5..., -1.3...]

>>> test0 = sc.parallelize([Row(features=Vectors.dense(-1.0, 0.0, 1.0, 1.0))]).toDF()

>>> model.transform(test0).head().prediction
0.0

>>> test1 = sc.parallelize([Row(features=Vectors.sparse(4, [0], [1.0]))]).toDF()

>>> model.transform(test1).head().prediction
2.0

>>> test2 = sc.parallelize([Row(features=Vectors.dense(0.5, 0.4, 0.3, 0.2))]).toDF()

>>> model.transform(test2).head().prediction
0.0

>>> model_path = temp_path + "/ovr_model"

>>> model.save(model_path)

>>> model2 = OneVsRestModel.load(model_path)

>>> model2.transform(test0).head().prediction
0.0
```

New in version 2.0.0.

**copy** (extra=None)

Creates a copy of this instance with a randomly generated uid and some extra params. This creates a deep copy of the embedded paramMap, and copies the embedded and extra parameters over.

**Parameters**

- **extra** – Extra parameters to copy to the new instance

**Returns**

Copy of this instance

New in version 2.0.0.

**setParams** (featuresCol='features', labelCol='label', predictionCol='prediction', classifier=None, weightCol=None, parallelism=1)

setParams(self, featuresCol="features", labelCol="label", predictionCol="prediction", classifier=None, weightCol=None, parallelism=1): Sets params for OneVsRest.
New in version 2.0.0.

```python
class pyspark.ml.classification.OneVsRestModel(models)
```

**Note:** Experimental

Model fitted by OneVsRest. This stores the models resulting from training k binary classifiers: one for each class. Each example is scored against all k models, and the model with the highest score is picked to label the example.

New in version 2.0.0.

```python
copy(extra=None)
```

Creates a copy of this instance with a randomly generated uid and some extra params. This creates a deep copy of the embedded paramMap, and copies the embedded and extra parameters over.

**Parameters**

- `extra` – Extra parameters to copy to the new instance

**Returns**

Copy of this instance

New in version 2.0.0.

### 25.4 Clustering API

```python
class pyspark.ml.clustering.BisectingKMeans(featuresCol='features', predictionCol='prediction', maxIter=20, seed=None, k=4, minDivisibleClusterSize=1.0, distanceMeasure='euclidean')
```

A bisecting k-means algorithm based on the paper “A comparison of document clustering techniques” by Steinbach, Karypis, and Kumar, with modification to fit Spark. The algorithm starts from a single cluster that contains all points. Iteratively it finds divisible clusters on the bottom level and bisects each of them using k-means, until there are k leaf clusters in total or no leaf clusters are divisible. The bisecting steps of clusters on the same level are grouped together to increase parallelism. If bisecting all divisible clusters on the bottom level would result more than k leaf clusters, larger clusters get higher priority.

```python
>>> from pyspark.ml.linalg import Vectors

>>> data = [(Vectors.dense([0.0, 0.0])), (Vectors.dense([1.0, 1.0])), ...
          (Vectors.dense([9.0, 8.0])), (Vectors.dense([8.0, 9.0])),]

>>> df = spark.createDataFrame(data, ["features"])

>>> bkm = BisectingKMeans(k=2, minDivisibleClusterSize=1.0)

>>> model = bkm.fit(df)

>>> centers = model.clusterCenters()

>>> len(centers)
2

>>> model.computeCost(df)
```

(continues on next page)
2.000...
>>> model.hasSummary
True
>>> summary = model.summary
>>> summary.k
2
>>> summary.clusterSizes
[2, 2]
>>> transformed = model.transform(df).select("features", "prediction")
>>> rows = transformed.collect()
>>> rows[0].prediction == rows[1].prediction
True
>>> rows[2].prediction == rows[3].prediction
True
>>> bkm_path = temp_path + "/bkm"
>>> bkm.save(bkm_path)
>>> bkm2 = BisectingKMeans.load(bkm_path)
>>> bkm2.getK()
2
>>> bkm2.getDistanceMeasure()
'euclidean'
>>> model_path = temp_path + "/bkm_model"
>>> model.save(model_path)
>>> model2 = BisectingKMeansModel.load(model_path)
>>> model2.hasSummary
False
>>> model.clusterCenters()[0] == model2.clusterCenters()[0]
array([True, True], dtype=bool)
>>> model.clusterCenters()[1] == model2.clusterCenters()[1]
array([True, True], dtype=bool)

New in version 2.0.0.

getDistanceMeasure ()
    Gets the value of distanceMeasure or its default value.

    New in version 2.4.0.

getK ()
    Gets the value of k or its default value.

    New in version 2.0.0.

getMinDivisibleClusterSize ()
    Gets the value of minDivisibleClusterSize or its default value.

    New in version 2.0.0.

setDistanceMeasure (value)
    Sets the value of distanceMeasure.

    New in version 2.4.0.

setK (value)
Sets the value of $k$.
New in version 2.0.0.

**setMinDivisibleClusterSize**(*value*)
Sets the value of minDivisibleClusterSize.
New in version 2.0.0.

**setParams**(self, featuresCol="features", predictionCol="prediction", maxIter=20,
seed=None, k=4, minDivisibleClusterSize=1.0, distanceMeasure="euclidean")
Sets params for BisectingKMeans.
New in version 2.0.0.

**class** pyspark.ml.clustering.BisectingKMeansModel(java_model=None)
Model fitted by BisectingKMeans.
New in version 2.0.0.

**clusterCenters**()
Get the cluster centers, represented as a list of NumPy arrays.
New in version 2.0.0.

**computeCost**(dataset)
Computes the sum of squared distances between the input points and their corresponding cluster centers.
New in version 2.0.0.

**hasSummary**
Indicates whether a training summary exists for this model instance.
New in version 2.1.0.

**summary**
Gets summary (e.g. cluster assignments, cluster sizes) of the model trained on the training set.
An exception is thrown if no summary exists.
New in version 2.1.0.

**class** pyspark.ml.clustering.BisectingKMeansSummary(java_obj=None)

---

**Note:** Experimental

Bisecting KMeans clustering results for a given model.
New in version 2.1.0.

**class** pyspark.ml.clustering.KMeans(featuresCol='features', predictionCol='prediction', k=2, initMode='k-means||', initSteps=2, tol=0.0001, maxIter=20,
seed= None, distanceMeasure='euclidean')
K-means clustering with a k-means++ like initialization mode (the k-means|| algorithm by Bahmani...
Learning Apache Spark with Python

et al).

```python
>>> from pyspark.ml.linalg import Vectors
>>> data = [(Vectors.dense([0.0, 0.0]),), (Vectors.dense([1.0, 1.0]),),
          (Vectors.dense([8.0, 9.0]),)]
>>> df = spark.createDataFrame(data, ["features"])
>>> kmeans = KMeans(k=2, seed=1)
>>> model = kmeans.fit(df)
>>> centers = model.clusterCenters()
>>> len(centers)
2
>>> model.computeCost(df)
2.000...
>>> transformed = model.transform(df).select("features", "prediction")
>>> rows = transformed.collect()
>>> rows[0].prediction == rows[1].prediction
True
>>> rows[2].prediction == rows[3].prediction
True
>>> model.hasSummary
True
>>> summary = model.summary
>>> summary.k
2
>>> summary.clusterSizes
[2, 2]
>>> summary.trainingCost
2.000...
>>> kmeans_path = temp_path + "/kmeans"
>>> kmeans.save(kmeans_path)
>>> kmeans2 = KMeans.load(kmeans_path)
>>> kmeans2.getK()
2
>>> model_path = temp_path + "/kmeans_model"
>>> model.save(model_path)
>>> model2 = KMeansModel.load(model_path)
>>> model2.hasSummary
False
>>> model.clusterCenters()[0] == model2.clusterCenters()[0]
array([ True,  True], dtype=bool)
>>> model.clusterCenters()[1] == model2.clusterCenters()[1]
array([ True,  True], dtype=bool)
```

New in version 1.5.0.

**getDistanceMeasure()**

Gets the value of `distanceMeasure`

New in version 2.4.0.

**getInitMode()**

Gets the value of `initMode`

New in version 1.5.0.
getInitSteps()  
Gets the value of initSteps
  New in version 1.5.0.

getK()  
Gets the value of k
  New in version 1.5.0.

setDistanceMeasure(value)  
Sets the value of distanceMeasure.
  New in version 2.4.0.

setInitMode(value)  
Sets the value of initMode.
  New in version 1.5.0.

setInitSteps(value)  
Sets the value of initSteps.
  New in version 1.5.0.

setK(value)  
Sets the value of k.
  New in version 1.5.0.

setParams(self, featuresCol="features", predictionCol="prediction", k=2, initMode="k-means||", initSteps=2, tol=1e-4, maxIter=20, seed=None, distanceMeasure="euclidean")  
Sets params for KMeans.
  New in version 1.5.0.

class pyspark.ml.clustering.KMeansModel(java_model=None)  
Model fitted by KMeans.
  New in version 1.5.0.

clusterCenters()  
Get the cluster centers, represented as a list of NumPy arrays.
  New in version 1.5.0.

computeCost(dataset)  
Return the K-means cost (sum of squared distances of points to their nearest center) for this model on the given data.
  ..note:: Deprecated in 2.4.0. It will be removed in 3.0.0. Use ClusteringEvaluator instead.
     You can also get the cost on the training dataset in the summary.
  New in version 2.0.0.

hasSummary  
Indicates whether a training summary exists for this model instance.
New in version 2.1.0.

**summary**

Gets summary (e.g. cluster assignments, cluster sizes) of the model trained on the training set. An exception is thrown if no summary exists.

New in version 2.1.0.

```python
class pyspark.ml.clustering.GaussianMixture(featuresCol='features', predictionCol='prediction', k=2, probabilityCol='probability', tol=0.01, maxIter=100, seed=None)
```

GaussianMixture clustering. This class performs expectation maximization for multivariate Gaussian Mixture Models (GMMs). A GMM represents a composite distribution of independent Gaussian distributions with associated “mixing” weights specifying each’s contribution to the composite.

Given a set of sample points, this class will maximize the log-likelihood for a mixture of k Gaussians, iterating until the log-likelihood changes by less than convergenceTol, or until it has reached the max number of iterations. While this process is generally guaranteed to converge, it is not guaranteed to find a global optimum.

**Note:** For high-dimensional data (with many features), this algorithm may perform poorly. This is due to high-dimensional data (a) making it difficult to cluster at all (based on statistical/theoretical arguments) and (b) numerical issues with Gaussian distributions.

```python
>>> from pyspark.ml.linalg import Vectors

>>> data = [(Vectors.dense([-0.1, -0.05 ])),
...         (Vectors.dense([-0.01, -0.1])),
...         (Vectors.dense([0.9, 0.8])),
...         (Vectors.dense([0.75, 0.935])),
...         (Vectors.dense([-0.83, -0.68])),
...         (Vectors.dense([-0.91, -0.76])),]

>>> df = spark.createDataFrame(data, ['features'])

>>> gm = GaussianMixture(k=3, tol=0.0001, maxIter=10, seed=10)

>>> model = gm.fit(df)

>>> model.hasSummary
True

>>> summary = model.summary

>>> summary.k
3

>>> summary.clusterSizes
[2, 2, 2]

>>> summary.logLikelihood
8.14636...

>>> weights = model.weights

>>> len(weights)
3

>>> model.gaussiansDF.select("mean").head()
```

(continues on next page)
New in version 2.0.0.

**getK()**
Gets the value of $k$
New in version 2.0.0.

**setK(value)**
Sets the value of $k$.
New in version 2.0.0.

**setParams**(self, **featuresCol=“features”, **predictionCol=“prediction”, **k=2, **probabilityCol=“probability”, **tol=0.01, **maxIter=100, **seed=0)
Sets params for GaussianMixture.
New in version 2.0.0.

**class** pyspark.ml.clustering.GaussianMixtureModel**(java_model=None)**
Model fitted by GaussianMixture.
New in version 2.0.0.

**gaussiansDF**
Retrieve Gaussian distributions as a DataFrame. Each row represents a Gaussian Distribution. The DataFrame has two columns: mean (Vector) and cov (Matrix).
New in version 2.0.0.
hasSummary
Indicates whether a training summary exists for this model instance.
New in version 2.1.0.

summary
Gets summary (e.g. cluster assignments, cluster sizes) of the model trained on the training set.
An exception is thrown if no summary exists.
New in version 2.1.0.

weights
Weight for each Gaussian distribution in the mixture. This is a multinomial probability distribution over the k Gaussians, where weights[i] is the weight for Gaussian i, and weights sum to 1.
New in version 2.0.0.

class pyspark.ml.clustering.GaussianMixtureSummary(java_obj=None)

Note: Experimental

Gaussian mixture clustering results for a given model.
New in version 2.1.0.

logLikelihood
Total log-likelihood for this model on the given data.
New in version 2.2.0.

probability
DataFrame of probabilities of each cluster for each training data point.
New in version 2.1.0.

probabilityCol
Name for column of predicted probability of each cluster in predictions.
New in version 2.1.0.

class pyspark.ml.clustering.LDA(featuresCol='features', maxIter=20, seed=None, checkpointInterval=10, k=10, optimizer='online', learningOffset=1024.0, learningDecay=0.51, subsamplingRate=0.05, optimizeDocConcentration=True, docConcentration=None, topicConcentration=None, topicDistributionCol='topicDistribution', keepLastCheckpoint=True)

Latent Dirichlet Allocation (LDA), a topic model designed for text documents.

Terminology:
• “term” = “word”: an element of the vocabulary
• “token”: instance of a term appearing in a document
• “topic”: multinomial distribution over terms representing some concept
• “document”: one piece of text, corresponding to one row in the input data


Input data (featuresCol): LDA is given a collection of documents as input data, via the featuresCol parameter. Each document is specified as a Vector of length vocabSize, where each entry is the count for the corresponding term (word) in the document. Feature transformers such as pyspark.ml.feature.Tokenizer and pyspark.ml.feature.CountVectorizer can be useful for converting text to word count vectors.

```python
>>> from pyspark.ml.linalg import Vectors, SparseVector
>>> from pyspark.ml.clustering import LDA

>>> df = spark.createDataFrame([[1, Vectors.dense([0.0, 1.0])],
...                            [2, SparseVector(2, {0: 1.0})]], ["id", "features"])

>>> lda = LDA(k=2, seed=1, optimizer="em")

>>> model = lda.fit(df)

>>> model.isDistributed()
True

>>> localModel = model.toLocal()

>>> localModel.isDistributed()
False

>>> model.vocabSize()
2

>>> model.describeTopics().show()
+-------------------+---------------------------------------------+
|topic|termIndices| termWeights|
+-------------------+---------------------------------------------+
| 0| [1, 0]|[0.50401530077160...|
| 1| [0, 1]|[0.50401530077160...|
+-------------------+---------------------------------------------+

>>> model.topicsMatrix()
DenseMatrix(2, 2, [0.496, 0.504, 0.504, 0.496], 0)

>>> lda_path = temp_path + "/lda"

>>> lda.save(lda_path)

>>> sameLDA = LDA.load(lda_path)

>>> distributed_model_path = temp_path + "/lda_distributed_model"

>>> model.save(distributed_model_path)

>>> sameModel = DistributedLDAModel.load(distributed_model_path)

>>> local_model_path = temp_path + "/lda_local_model"

>>> localModel.save(local_model_path)

>>> sameLocalModel = LocalLDAModel.load(local_model_path)
```

New in version 2.0.0.

**getDocConcentration()**

- Gets the value of docConcentration or its default value.

New in version 2.0.0.
getK()  
get the value of k or its default value.

New in version 2.0.0.

getKeepLastCheckpoint()  
get the value of keepLastCheckpoint or its default value.

New in version 2.0.0.

getLearningDecay()  
get the value of learningDecay or its default value.

New in version 2.0.0.

getLearningOffset()  
get the value of learningOffset or its default value.

New in version 2.0.0.

getOptimizeDocConcentration()  
get the value of optimizeDocConcentration or its default value.

New in version 2.0.0.

getOptimizer()  
get the value of optimizer or its default value.

New in version 2.0.0.

getSubsamplingRate()  
get the value of subsamplingRate or its default value.

New in version 2.0.0.

getTopicConcentration()  
get the value of topicConcentration or its default value.

New in version 2.0.0.

getTopicDistributionCol()  
get the value of topicDistributionCol or its default value.

New in version 2.0.0.

setDocConcentration(value)  
Sets the value of docConcentration.

```python
>>> algo = LDA().setDocConcentration([0.1, 0.2])
>>> algo.getDocConcentration()
[0.1..., 0.2...]
```

New in version 2.0.0.

setK(value)  
Sets the value of k.
Learning Apache Spark with Python

>>> algo = LDA().setK(10)
>>> algo.getK()
10

New in version 2.0.0.

**setKeepLastCheckpoint**( value )
Sets the value of keepLastCheckpoint.

>>> algo = LDA().setKeepLastCheckpoint(False)
>>> algo.getKeepLastCheckpoint()
False

New in version 2.0.0.

**setLearningDecay**( value )
Sets the value of learningDecay.

>>> algo = LDA().setLearningDecay(0.1)
>>> algo.getLearningDecay()
0.1...

New in version 2.0.0.

**setLearningOffset**( value )
Sets the value of learningOffset.

>>> algo = LDA().setLearningOffset(100)
>>> algo.getLearningOffset()
100.0

New in version 2.0.0.

**setOptimizeDocConcentration**( value )
Sets the value of optimizeDocConcentration.

>>> algo = LDA().setOptimizeDocConcentration(True)
>>> algo.getOptimizeDocConcentration()
True

New in version 2.0.0.

**setOptimizer**( value )
Sets the value of optimizer. Currently only support ‘em’ and ‘online’.

>>> algo = LDA().setOptimizer("em")
>>> algo.getOptimizer()
'em'

New in version 2.0.0.
setParams(self, featuresCol="features", maxIter=20, seed=None, checkpointInterval=10, k=10, optimizer="online", learningOffset=1024.0, learningDecay=0.51, subsamplingRate=0.05, optimizeDocConcentration=True, docConcentration=None, topicConcentration=None, topicDistributionCol="topicDistribution", keepLastCheckpoint=True)

Sets params for LDA.

New in version 2.0.0.

setSubsamplingRate(value)

Sets the value of subsamplingRate.

```python
>>> algo = LDA().setSubsamplingRate(0.1)
>>> algo.getSubsamplingRate()
0.1...
```

New in version 2.0.0.

setTopicConcentration(value)

Sets the value of topicConcentration.

```python
>>> algo = LDA().setTopicConcentration(0.5)
>>> algo.getTopicConcentration()
0.5...
```

New in version 2.0.0.

setTopicDistributionCol(value)

Sets the value of topicDistributionCol.

```python
>>> algo = LDA().setTopicDistributionCol("topicDistributionCol")
>>> algo.getTopicDistributionCol()
'topicDistributionCol'
```

New in version 2.0.0.

class pyspark.ml.clustering.LDAModel(java_model=None)

Latent Dirichlet Allocation (LDA) model. This abstraction permits for different underlying representations, including local and distributed data structures.

New in version 2.0.0.

describeTopics(maxTermsPerTopic=10)

Return the topics described by their top-weighted terms.

New in version 2.0.0.

estimatedDocConcentration()

Value for LDA.docConcentration estimated from data. If Online LDA was used and LDA.optimizeDocConcentration was set to false, then this returns the fixed (given) value for the LDA.docConcentration parameter.

New in version 2.0.0.
isDistributed()
Indicates whether this instance is of type DistributedLDAModel
New in version 2.0.0.

logLikelihood(dataset)
Calculates a lower bound on the log likelihood of the entire corpus. See Equation (16) in the
Online LDA paper (Hoffman et al., 2010).

WARNING: If this model is an instance of DistributedLDAModel (produced when optimizer is set to “em”), this involves collecting a large topicsMatrix() to the driver. This implementation may be changed in the future.
New in version 2.0.0.

logPerplexity(dataset)
Calculate an upper bound on perplexity. (Lower is better.) See Equation (16) in the Online LDA paper (Hoffman et al., 2010).

WARNING: If this model is an instance of DistributedLDAModel (produced when optimizer is set to “em”), this involves collecting a large topicsMatrix() to the driver. This implementation may be changed in the future.
New in version 2.0.0.

topicsMatrix()
Inferred topics, where each topic is represented by a distribution over terms. This is a matrix of size vocabSize x k, where each column is a topic. No guarantees are given about the ordering of the topics.

WARNING: If this model is actually a DistributedLDAModel instance produced by the Expectation-Maximization (“em”) optimizer, then this method could involve collecting a large amount of data to the driver (on the order of vocabSize x k).
New in version 2.0.0.

dacl
Vocabulary size (number of terms or words in the vocabulary)
New in version 2.0.0.

class pyspark.ml.clustering.LocalLDAModel(java_model=None)
Local (non-distributed) model fitted by LDA. This model stores the inferred topics only; it does not store info about the training dataset.
New in version 2.0.0.

class pyspark.ml.clustering.DistributedLDAModel(java_model=None)
Distributed model fitted by LDA. This type of model is currently only produced by Expectation-Maximization (EM).
This model stores the inferred topics, the full training dataset, and the topic distribution for each training document.
New in version 2.0.0.
**getCheckpointFiles()**

If using checkpointing and `LDA.keepLastCheckpoint` is set to true, then there may be saved checkpoint files. This method is provided so that users can manage those files.

**Note:** Removing the checkpoints can cause failures if a partition is lost and is needed by certain `DistributedLDAModel` methods. Reference counting will clean up the checkpoints when this model and derivative data go out of scope.

```
:return List of checkpoint files from training
New in version 2.0.0.
```

**logPrior()**

Log probability of the current parameter estimate: \( \log P(\text{topics, topic distributions for docs} \mid \alpha, \eta) \)

New in version 2.0.0.

**toLocal()**

Convert this distributed model to a local representation. This discards info about the training dataset.

**WARNING:** This involves collecting a large `topicsMatrix()` to the driver.

New in version 2.0.0.

**trainingLogLikelihood()**

Log likelihood of the observed tokens in the training set, given the current parameter estimates: \( \log P(\text{docs} \mid \text{topics, topic distributions for docs, Dirichlet hyperparameters}) \)

**Notes:**

- This excludes the prior; for that, use `logPrior()`.
- Even with `logPrior()`, this is NOT the same as the data log likelihood given the hyperparameters.
- This is computed from the topic distributions computed during training. If you call `logLikelihood()` on the same training dataset, the topic distributions will be computed again, possibly giving different results.

New in version 2.0.0.

```python
class pyspark.ml.clustering.PowerIterationClustering(k=2, maxIter=20, initMode='random', srcCol='src', dstCol='dst', weightCol=None)
```

**Note:** Experimental
Power Iteration Clustering (PIC), a scalable graph clustering algorithm developed by Lin and Cohen. From the abstract: PIC finds a very low-dimensional embedding of a dataset using truncated power iteration on a normalized pair-wise similarity matrix of the data.

This class is not yet an Estimator/Transformer, use `assignClusters()` method to run the PowerIterationClustering algorithm.

See also:

Wikipedia on Spectral clustering

```python
>>> data = [(1, 0, 0.5),
          (2, 0, 0.5), (2, 1, 0.7),
          (3, 0, 0.5), (3, 1, 0.7), (3, 2, 0.9),
          (4, 0, 0.5), (4, 1, 0.7), (4, 2, 0.9), (4, 3, 1.1),
          (5, 0, 0.5), (5, 1, 0.7), (5, 2, 0.9), (5, 3, 1.1), (5, 4, 1.3)]
>>> df = spark.createDataFrame(data).toDF("src", "dst", "weight")
>>> pic = PowerIterationClustering(k=2, maxIter=40, weightCol="weight")
>>> assignments = pic.assignClusters(df)
>>> assignments.sort(assignments.id).show(truncate=False)
+---+-------+
|id |cluster|
+---+-------+
| 0 |1 |
| 1 |1 |
| 2 |1 |
| 3 |1 |
| 4 |1 |
| 5 |0 |
+---+-------+
...  
>>> pic_path = temp_path + "/pic"
>>> pic.save(pic_path)
>>> pic2 = PowerIterationClustering.load(pic_path)
>>> pic2.getK()
2
>>> pic2.getMaxIter()
40
```

New in version 2.4.0.

**assignClusters(dataset)**

Run the PIC algorithm and returns a cluster assignment for each input vertex.

Parameters

- **dataset** – A dataset with columns src, dst, weight representing the affinity matrix, which is the matrix A in the PIC paper. Suppose the src column value is i, the dst column value is j, the weight column value is similarity s_{ij}, which must be nonnegative. This is a symmetric matrix and hence s_{ij} = s_{ji}.

For any (i, j) with nonzero similarity, there should be either (i, j, s_{ij}) or (j, i, s_{ji}) in the input. Rows with i = j are ignored, because we assume s_{ii} = 0.0.

Returns

A dataset that contains columns of vertex id and the corresponding cluster for the id. The schema of it will be: - id: Long - cluster: Int
New in version 2.4.0.

getDstCol()
    Gets the value of dstCol or its default value.
    New in version 2.4.0.

getInitMode()
    Gets the value of initMode or its default value.
    New in version 2.4.0.

getK()
    Gets the value of k or its default value.
    New in version 2.4.0.

getSrcCol()
    Gets the value of srcCol or its default value.
    New in version 2.4.0.

setDstCol(value)
    Sets the value of dstCol.
    New in version 2.4.0.

setInitMode(value)
    Sets the value of initMode.
    New in version 2.4.0.

setK(value)
    Sets the value of k.
    New in version 2.4.0.

setParams(self, k=2, maxIter=20, initMode="random", srcCol="src", dstCol="dst", weightCol=None)
    Sets params for PowerIterationClustering.
    New in version 2.4.0.

setSrcCol(value)
    Sets the value of srcCol.
    New in version 2.4.0.
25.5 Recommendation API

```python
class pyspark.ml.recommendation.ALS(rank=10, maxIter=10, regParam=0.1, numUserBlocks=10, numItemBlocks=10, implicitPrefs=False, alpha=1.0, userCol='user', itemCol='item', seed=None, ratingCol='rating', nonnegative=False, checkpointInterval=10, intermediateStorageLevel='MEMORY_AND_DISK', finalStorageLevel='MEMORY_AND_DISK', coldStartStrategy='nan')
```

Alternating Least Squares (ALS) matrix factorization.

ALS attempts to estimate the ratings matrix $R$ as the product of two lower-rank matrices, $X$ and $Y$, i.e. $X \times Y^t = R$. Typically these approximations are called ‘factor’ matrices. The general approach is iterative. During each iteration, one of the factor matrices is held constant, while the other is solved for using least squares. The newly-solved factor matrix is then held constant while solving for the other factor matrix.

This is a blocked implementation of the ALS factorization algorithm that groups the two sets of factors (referred to as “users” and “products”) into blocks and reduces communication by only sending one copy of each user vector to each product block on each iteration, and only for the product blocks that need that user’s feature vector. This is achieved by pre-computing some information about the ratings matrix to determine the “out-links” of each user (which blocks of products it will contribute to) and “in-link” information for each product (which of the feature vectors it receives from each user block it will depend on). This allows us to send only an array of feature vectors between each user block and product block, and have the product block find the users’ ratings and update the products based on these messages.

For implicit preference data, the algorithm used is based on “Collaborative Filtering for Implicit Feedback Datasets”, adapted for the blocked approach used here.

Essentially instead of finding the low-rank approximations to the rating matrix $R$, this finds the approximations for a preference matrix $P$ where the elements of $P$ are 1 if $r > 0$ and 0 if $r \leq 0$. The ratings then act as ‘confidence’ values related to strength of indicated user preferences rather than explicit ratings given to items.
predictions[0]  
Row(user=0, item=2, prediction=-0.13807615637779236)
predictions[1]  
Row(user=1, item=0, prediction=2.6258413791656494)
predictions[2]  
Row(user=2, item=0, prediction=-1.5018409490585327)
user_recs = model.recommendForAllUsers(3)
user_recs.where(user_recs.user == 0) .select("recommendations.
   item", "recommendations.rating").collect()
Row(item=[0, 1, 2], rating=[3.910..., 1.992..., -0.138...])
item_recs = model.recommendForAllItems(3)
item_recs.where(item_recs.item == 2) .select("recommendations.
   user", "recommendations.rating").collect()
Row(user=[2, 1, 0], rating=[4.901..., 3.981..., -0.138...])
user_subset = df.where(df.user == 2)
user_subset_recs = model.recommendForUserSubset(user_subset, 3)
user_subset_recs.select("recommendations.item", "recommendations.
   rating").first()
Row(item=[2, 1, 0], rating=[4.901..., 1.056..., -1.501...])
item_subset = df.where(df.item == 0)
item_subset_recs = model.recommendForItemSubset(item_subset, 3)
item_subset_recs.select("recommendations.user", "recommendations.
   rating").first()
Row(user=[0, 1, 2], rating=[3.910..., 2.625..., -1.501...])
als_path = temp_path + "/als"
als.save(als_path)
als2 = ALS.load(als_path)
als.getMaxIter()  
5
model_path = temp_path + "/als_model"
model.save(model_path)
model2 = ALSModel.load(model_path)
model.rank == model2.rank
True
sorted(model.userFactors.collect()) == sorted(model2.userFactors.
   collect())
True
sorted(model.itemFactors.collect()) == sorted(model2.itemFactors.
   collect())
True

New in version 1.4.0.

**getAlpha()**

Gets the value of alpha or its default value.

New in version 1.4.0.

**getColdStartStrategy()**

Gets the value of coldStartStrategy or its default value.

New in version 2.2.0.
getFinalStorageLevel()  
Gets the value of finalStorageLevel or its default value.

New in version 2.0.0.

getImplicitPrefs()  
Gets the value of implicitPrefs or its default value.

New in version 1.4.0.

getIntermediateStorageLevel()  
Gets the value of intermediateStorageLevel or its default value.

New in version 2.0.0.

g getItemCol()  
Gets the value of itemCol or its default value.

New in version 1.4.0.

g getNonnegative()  
Gets the value of nonnegative or its default value.

New in version 1.4.0.

g getNumItemBlocks()  
Gets the value of numItemBlocks or its default value.

New in version 1.4.0.

g getNumUserBlocks()  
Gets the value of numUserBlocks or its default value.

New in version 1.4.0.

g getRank()  
Gets the value of rank or its default value.

New in version 1.4.0.

g getRatingCol()  
Gets the value of ratingCol or its default value.

New in version 1.4.0.

g getUserCol()  
Gets the value of userCol or its default value.

New in version 1.4.0.

setAlpha(value)  
Sets the value of alpha.

New in version 1.4.0.

setColdStartStrategy(value)  
Sets the value of coldStartStrategy.

New in version 2.2.0.
setFinalStorageLevel(value)
Sets the value of finalStorageLevel.
New in version 2.0.0.

setImplicitPrefs(value)
Sets the value of implicitPrefs.
New in version 1.4.0.

setIntermediateStorageLevel(value)
Sets the value of intermediateStorageLevel.
New in version 2.0.0.

setItemCol(value)
Sets the value of itemCol.
New in version 1.4.0.

setNonnegative(value)
Sets the value of nonnegative.
New in version 1.4.0.

setNumBlocks(value)
Sets both numUserBlocks and numItemBlocks to the specific value.
New in version 1.4.0.

setNumItemBlocks(value)
Sets the value of numItemBlocks.
New in version 1.4.0.

setNumUserBlocks(value)
Sets the value of numUserBlocks.
New in version 1.4.0.

setParams(self, rank=10, maxIter=10, regParam=0.1, numUserBlocks=10, numItemBlocks=10, implicitPrefs=False, alpha=1.0, userCol="user", itemCol="item", seed=None, ratingCol="rating", nonnegative=False, checkpointInterval=10, intermediateStorageLevel="MEMORY_AND_DISK", finalStorageLevel="MEMORY_AND_DISK", coldStartStrategy="nan")
Sets params for ALS.
New in version 1.4.0.

setRank(value)
Sets the value of rank.
New in version 1.4.0.

setRatingCol(value)
Sets the value of ratingCol.
New in version 1.4.0.
setUserCol (value)
   Sets the value of userCol.
   New in version 1.4.0.

class pyspark.ml.recommendation.ALSModel (java_model=None)
   Model fitted by ALS.
   New in version 1.4.0.

itemFactors
   id and features
   New in version 1.4.0.

   Type a DataFrame that stores item factors in two columns

rank
   rank of the matrix factorization model
   New in version 1.4.0.

recommendForAllItems (numUsers)
   Returns top numUsers users recommended for each item, for all items.

   Parameters numUsers – max number of recommendations for each item

   Returns a DataFrame of (itemCol, recommendations), where recommendations are
   stored as an array of (userCol, rating) Rows.
   New in version 2.2.0.

recommendForAllUsers (numItems)
   Returns top numItems items recommended for each user, for all users.

   Parameters numItems – max number of recommendations for each user

   Returns a DataFrame of (userCol, recommendations), where recommendations are
   stored as an array of (itemCol, rating) Rows.
   New in version 2.2.0.

recommendForItemSubset (dataset, numUsers)
   Returns top numUsers users recommended for each item id in the input data set. Note that if
   there are duplicate ids in the input dataset, only one set of recommendations per unique id will
   be returned.

   Parameters

   • dataset – a Dataset containing a column of item ids. The column name must
     match itemCol.

   • numUsers – max number of recommendations for each item

   Returns a DataFrame of (itemCol, recommendations), where recommendations are
   stored as an array of (userCol, rating) Rows.
   New in version 2.3.0.
**recommendForUserSubset** *(dataset, numItems)*

Returns top `numItems` items recommended for each user id in the input data set. Note that if there are duplicate ids in the input dataset, only one set of recommendations per unique id will be returned.

**Parameters**

- **dataset** – a Dataset containing a column of user ids. The column name must match `userCol`.
- **numItems** – max number of recommendations for each user

**Returns** a DataFrame of (userCol, recommendations), where recommendations are stored as an array of (itemCol, rating) Rows.

New in version 2.3.0.

**userFactors**

`id` and `features`

New in version 1.4.0.

**Type** a DataFrame that stores user factors in two columns

---

### 25.6 Pipeline API

**class** `pyspark.ml.pipeline.Pipeline(stages=None)`

A simple pipeline, which acts as an estimator. A Pipeline consists of a sequence of stages, each of which is either an **Estimator** or a **Transformer**. When Pipeline.fit() is called, the stages are executed in order. If a stage is an **Estimator**, its **Estimator.fit()** method will be called on the input dataset to fit a model. Then the model, which is a transformer, will be used to transform the dataset as the input to the next stage. If a stage is a **Transformer**, its **Transformer.transform()** method will be called to produce the dataset for the next stage. The fitted model from a **Pipeline** is a **PipelineModel**, which consists of fitted models and transformers, corresponding to the pipeline stages. If stages is an empty list, the pipeline acts as an identity transformer.

New in version 1.3.0.

**copy(extra=None)**

Creates a copy of this instance.

**Parameters** `extra` – extra parameters

**Returns** new instance

New in version 1.4.0.

**getStages()**

Get pipeline stages.

New in version 1.3.0.

**classmethod read()**

Returns an MLReader instance for this class.
New in version 1.3.0.

```python
setStages(value)
```
Set pipeline stages.

**Parameters** value – a list of transformers or estimators

**Returns** the pipeline instance

New in version 1.3.0.

```python
write()
```
Returns an MLWriter instance for this ML instance.

New in version 2.0.0.

```python
class pyspark.ml.pipeline.PipelineModel(stages)
```
Represents a compiled pipeline with transformers and fitted models.

New in version 1.3.0.

```python
copy(extra=None)
```
Creates a copy of this instance.

**Parameters** extra – extra parameters

**Returns** new instance

New in version 1.4.0.

```python
classmethod read()
```
Returns an MLReader instance for this class.

New in version 2.0.0.

```python
write()
```
Returns an MLWriter instance for this ML instance.

New in version 2.0.0.

```python
class pyspark.ml.pipeline.PipelineModelReader(cls)
```
(Private) Specialization of MLReader for PipelineModel types

```python
load(path)
```
Load the ML instance from the input path.

```python
class pyspark.ml.pipeline.PipelineModelWriter(instance)
```
(Private) Specialization of MLWriter for PipelineModel types

```python
saveImpl(path)
```
save() handles overwriting and then calls this method. Subclasses should override this method to implement the actual saving of the instance.
class pyspark.ml.pipeline.PipelineReader(cls)
   (Private) Specialization of MLReader for Pipeline types

   load(path)
   Load the ML instance from the input path.

class pyspark.ml.pipeline.PipelineSharedReadWrite

   Note: DeveloperApi

   Functions for MLReader and MLWriter shared between Pipeline and PipelineModel
   New in version 2.3.0.

   static getStagePath(stageUid, stageIdx, numStages, stagesDir)
   Get path for saving the given stage.

   static load(metadata, sc, path)
   Load metadata and stages for a Pipeline or PipelineModel
   Returns (UID, list of stages)

   static saveImpl(instance, stages, sc, path)
   Save metadata and stages for a Pipeline or PipelineModel - save metadata to path/metadata - save stages to stages/IDX_UID

   static validateStages(stages)
   Check that all stages are Writable

class pyspark.ml.pipeline.PipelineWriter(instance)
   (Private) Specialization of MLWriter for Pipeline types

   saveImpl(path)
   save() handles overwriting and then calls this method. Subclasses should override this method to implement the actual saving of the instance.

25.7 Tuning API

class pyspark.ml.tuning.ParamGridBuilder
   Builder for a param grid used in grid search-based model selection.

    >>> from pyspark.ml.classification import LogisticRegression
    >>> lr = LogisticRegression()
    >>> output = ParamGridBuilder()
    ... .baseOn((lr.labelCol: 'l'))
    ... .baseOn([lr.predictionCol, 'p'])
    ... .addGrid(lr.regParam, [1.0, 2.0])
    ... .addGrid(lr.maxIter, [1, 5])
    ... .build()
    >>> expected = [}
New in version 1.4.0.

addGrid (param, values)
Sets the given parameters in this grid to fixed values.
New in version 1.4.0.

baseOn (*)(args)
Sets the given parameters in this grid to fixed values. Accepts either a parameter dictionary or a list of (parameter, value) pairs.
New in version 1.4.0.

build()
Builds and returns all combinations of parameters specified by the param grid.
New in version 1.4.0.

class pyspark.ml.tuning.CrossValidator (estimator=None, estimatorParamMaps=None, evaluator=None, numFolds=3, seed=None, parallelism=1, collectSubModels=False)

K-fold cross validation performs model selection by splitting the dataset into a set of non-overlapping randomly partitioned folds which are used as separate training and test datasets e.g., with k=3 folds, K-fold cross validation will generate 3 (training, test) dataset pairs, each of which uses 2/3 of the data for training and 1/3 for testing. Each fold is used as the test set exactly once.

```python
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.evaluation import BinaryClassificationEvaluator
>>> from pyspark.ml.linalg import Vectors
>>> dataset = spark.createDataFrame(
... [(Vectors.dense([0.0]), 0.0),
... (Vectors.dense([0.4]), 1.0),
... (Vectors.dense([0.5]), 0.0),
... (Vectors.dense([0.6]), 1.0),
... (Vectors.dense([1.0]), 1.0)] * 10,
... ["features", "label"])
>>> lr = LogisticRegression()
>>> grid = ParamGridBuilder().addGrid(lr.maxIter, [0, 1]).build()
```
```python
>>> evaluator = BinaryClassificationEvaluator()
>>> cv = CrossValidator(estimator=lr, estimatorParamMaps=grid,
... evaluator=evaluator,
... parallelism=2)
>>> cvModel = cv.fit(dataset)
>>> cvModel.avgMetrics[0]
0.5
>>> evaluator.evaluate(cvModel.transform(dataset))
0.8333...
```

New in version 1.4.0.

**copy**(extra=None)

Creates a copy of this instance with a randomly generated uid and some extra params. This copies creates a deep copy of the embedded paramMap, and copies the embedded and extra parameters over.

**Parameters**

extra – Extra parameters to copy to the new instance

**Returns**

Copy of this instance

New in version 1.4.0.

**getNumFolds**()

Gets the value of numFolds or its default value.

New in version 1.4.0.

**class method** read()

Returns an MLReader instance for this class.

New in version 2.3.0.

**setNumFolds**(value)

Sets the value of numFolds.

New in version 1.4.0.

**setParams**(estimator=None, estimatorParamMaps=None, evaluator=None, numFolds=3,
seed=None, parallelism=1, collectSubModels=False)

setParams(self, estimator=None, estimatorParamMaps=None, evaluator=None, numFolds=3,
seed=None, parallelism=1, collectSubModels=False): Sets params for cross validator.

New in version 1.4.0.

**write**()

Returns an MLWriter instance for this ML instance.

New in version 2.3.0.

**class** pyspark.ml.tuning.CrossValidatorModel(bestModel, avgMetrics=[], sub-
Models=None)

CrossValidatorModel contains the model with the highest average cross-validation metric across folds and uses this model to transform input data. CrossValidatorModel also tracks the metrics for each param map evaluated.
New in version 1.4.0.

**avgMetrics = None**
Average cross-validation metrics for each paramMap in CrossValidator.estimatorParamMaps, in the corresponding order.

**bestModel = None**
best model from cross validation

**copy(extra=None)**
Creates a copy of this instance with a randomly generated uid and some extra params. This copies the underlying bestModel, creates a deep copy of the embedded paramMap, and copies the embedded and extra parameters over. It does not copy the extra Params into the subModels.

**Parameters**
- **extra** – Extra parameters to copy to the new instance

**Returns**
Copy of this instance

New in version 1.4.0.

**class method read()**
Returns an MLReader instance for this class.

New in version 2.3.0.

**subModels = None**
sub model list from cross validation

**write()**
Returns an MLWriter instance for this ML instance.

New in version 2.3.0.

```python
class pyspark.ml.tuning.TrainValidationSplit (estimator=None, estimatorParamMaps=None, evaluator=None, trainRatio=0.75, parallelism=1, collectSubModels=False, seed=None)
```

**Note:** Experimental

Validation for hyper-parameter tuning. Randomly splits the input dataset into train and validation sets, and uses evaluation metric on the validation set to select the best model. Similar to `CrossValidator`, but only splits the set once.

```python
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.evaluation import BinaryClassificationEvaluator
>>> from pyspark.ml.linalg import Vectors

>>> dataset = spark.createDataFrame(
...   [(Vectors.dense([0.0]), 0.0),
...    (Vectors.dense([0.4]), 1.0),
...    (Vectors.dense([0.5]), 0.0),
...    (Vectors.dense([0.6]), 1.0),
```

(continues on next page)
... (Vectors.dense([1.0], 1.0)) * 10,
... ["features", "label"]

```
>>> lr = LogisticRegression()
>>> grid = ParamGridBuilder().addGrid(lr.maxIter, [0, 1]).build()
>>> evaluator = BinaryClassificationEvaluator()
>>> tvs = TrainValidationSplit(estimator=lr, estimatorParamMaps=grid,
... evaluator=evaluator,
... parallelism=2)
>>> tvsModel = tvs.fit(dataset)
>>> evaluator.evaluate(tvsModel.transform(dataset))
0.8333...
```

New in version 2.0.0.

**copy (extra=None)**

Creates a copy of this instance with a randomly generated uid and some extra params. This copies creates a deep copy of the embedded paramMap, and copies the embedded and extra parameters over.

**Parameters**

**extra** – Extra parameters to copy to the new instance

**Returns**

Copy of this instance

New in version 2.0.0.

**getTrainRatio()**

Gets the value of trainRatio or its default value.

New in version 2.0.0.

**classmethod read()**

Returns an MLReader instance for this class.

New in version 2.3.0.

**setParams (estimator=None, estimatorParamMaps=None, evaluator=None, trainRatio=0.75, parallelism=1, collectSubModels=False, seed=None)**

Sets params for the train validation split.

New in version 2.0.0.

**setTrainRatio (value)**

Sets the value of trainRatio.

New in version 2.0.0.

**write ()**

Returns an MLWriter instance for this ML instance.

New in version 2.3.0.
Learning Apache Spark with Python

```
class pyspark.ml.tuning.TrainValidationSplitModel (bestModel, validation-
Metrics=[], subMod-
els=None)

Note: Experimental

Model from train validation split.
New in version 2.0.0.

bestModel = None
best model from train validation split

copy (extra=None)
Creates a copy of this instance with a randomly generated uid and some extra params. This
copies the underlying bestModel, creates a deep copy of the embedded paramMap, and copies
the embedded and extra parameters over. And, this creates a shallow copy of the validation-
Metrics. It does not copy the extra Params into the subModels.

Parameters extra – Extra parameters to copy to the new instance

Returns Copy of this instance
New in version 2.0.0.

classmethod read()
Returns an MLReader instance for this class.
New in version 2.3.0.

subModels = None
sub models from train validation split

validationMetrics = None
evaluated validation metrics

write ()
Returns an MLWriter instance for this ML instance.
New in version 2.3.0.

25.8 Evaluation API

class pyspark.ml.evaluation.Evaluator
Base class for evaluators that compute metrics from predictions.
New in version 1.4.0.

evaluate (dataset, params=None)
Evaluates the output with optional parameters.

Parameters
- **dataset** – a dataset that contains labels/observations and predictions
- **params** – an optional param map that overrides embedded params

**Returns** metric

New in version 1.4.0.

**isLargerBetter()**

Indicates whether the metric returned by `evaluate()` should be maximized (True, default) or minimized (False). A given evaluator may support multiple metrics which may be maximized or minimized.

New in version 1.5.0.

```python
class pyspark.ml.evaluation.BinaryClassificationEvaluator(rawPredictionCol='rawPrediction', labelCol='label', metricName='areaUnderROC')
```

**Note:** Experimental

Evaluator for binary classification, which expects two input columns: rawPrediction and label. The rawPrediction column can be of type double (binary 0/1 prediction, or probability of label 1) or of type vector (length-2 vector of raw predictions, scores, or label probabilities).

```python
>>> from pyspark.ml.linalg import Vectors
>>> scoreAndLabels = map(lambda x: (Vectors.dense([1.0 - x[0], x[0]]), x[1]), [(0.1, 0.0), (0.1, 1.0), (0.4, 0.0), (0.6, 0.0), (0.6, 1.0), (0.6, 1.0)])
>>> dataset = spark.createDataFrame(scoreAndLabels, ['raw', 'label'])
... >>> evaluator = BinaryClassificationEvaluator(rawPredictionCol="raw")
... >>> evaluator.evaluate(dataset)
0.70...
... >>> evaluator.evaluate(dataset, {evaluator.metricName: "areaUnderPR"})
0.83...
... >>> bce_path = temp_path + "/bce"
... >>> evaluator.save(bce_path)
... >>> evaluator2 = BinaryClassificationEvaluator.load(bce_path)
... >>> str(evaluator2.getRawPredictionCol())
'raw'
```

New in version 1.4.0.

**getMetricName()**

Gets the value of metricName or its default value.

New in version 1.4.0.
Learning Apache Spark with Python

setMetricName(value)
Sets the value of metricName.
New in version 1.4.0.

setParams(self, rawPredictionCol="rawPrediction", labelCol="label", metricName="areaUnderROC")
Sets params for binary classification evaluator.
New in version 1.4.0.

class pyspark.ml.evaluation.RegressionEvaluator(predictionCol='prediction', labelCol='label', metricName='rmse')

Note: Experimental

Evaluator for Regression, which expects two input columns: prediction and label.

```python
>>> scoreAndLabels = [(-28.98343821, -27.0), (20.21491975, 21.5),
                    ... (-25.98418959, -22.0), (30.69731842, 33.0), (74.69283752, 71.0)]
>>> dataset = spark.createDataFrame(scoreAndLabels, ['raw', 'label'])
...
>>> evaluator = RegressionEvaluator(predictionCol="raw")
>>> evaluator.evaluate(dataset)
2.842...
>>> evaluator.evaluate(dataset, {evaluator.metricName: "r2"})
0.993...
>>> evaluator.evaluate(dataset, {evaluator.metricName: "mae"})
2.649...
>>> re_path = temp_path + "/re"
>>> evaluator.save(re_path)
>>> evaluator2 = RegressionEvaluator.load(re_path)
>>> str(evaluator2.getPredictionCol())
'raw'
```

New in version 1.4.0.

getMetricName()
Gets the value of metricName or its default value.
New in version 1.4.0.

setMetricName(value)
Sets the value of metricName.
New in version 1.4.0.

setParams(self, predictionCol="prediction", labelCol="label", metricName="rmse")
Sets params for regression evaluator.
New in version 1.4.0.
Learning Apache Spark with Python

class pyspark.ml.evaluation.MulticlassClassificationEvaluator (predictionCol='prediction', labelCol='label', metricName='f1')

Note: Experimental

Evaluator for Multiclass Classification, which expects two input columns: prediction and label.

```python
>>> scoreAndLabels = [(0.0, 0.0), (0.0, 1.0), (0.0, 0.0),
... (1.0, 0.0), (1.0, 1.0), (1.0, 1.0), (1.0, 1.0), (2.0, 2.0), (2.0, 0.0)]
>>> dataset = spark.createDataFrame(scoreAndLabels, ['prediction', 'label'])
... >>> evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
... >>> evaluator.evaluate(dataset)
0.66...
... >>> evaluator.evaluate(dataset, {evaluator.metricName: "accuracy"})
0.66...
... >>> mce_path = temp_path + "/mce"
... >>> evaluator.save(mce_path)
... >>> evaluator2 = MulticlassClassificationEvaluator.load(mce_path)
... >>> str(evaluator2.getPredictionCol())
'prediction'

New in version 1.5.0.

getMetricName ()
   Gets the value of metricName or its default value.
   New in version 1.5.0.

setMetricName (value)
   Sets the value of metricName.
   New in version 1.5.0.

setParams (self, predictionCol="prediction", labelCol="label", metricName="f1")
   Sets params for multiclass classification evaluator.
   New in version 1.5.0.

class pyspark.ml.evaluation.ClusteringEvaluator (predictionCol='prediction', featuresCol='features', metricName='silhouette', distanceMeasure='squaredEuclidean')
```
Note: Experimental

Evaluator for Clustering results, which expects two input columns: prediction and features. The metric computes the Silhouette measure using the squared Euclidean distance.

The Silhouette is a measure for the validation of the consistency within clusters. It ranges between 1 and -1, where a value close to 1 means that the points in a cluster are close to the other points in the same cluster and far from the points of the other clusters.

```python
>>> from pyspark.ml.linalg import Vectors

>>> featureAndPredictions = map(lambda x: (Vectors.dense(x[0]), x[1]), ...
  ...   
  ...   
  ...   
  ...   
  ...   
  ...

>>> dataset = spark.createDataFrame(featureAndPredictions, ["features", ...
  ...

>>> evaluator = ClusteringEvaluator(predictionCol="prediction")

>>> evaluator.evaluate(dataset)
0.9079...

>>> ce_path = temp_path + "/ce"

>>> evaluator.save(ce_path)

>>> evaluator2 = ClusteringEvaluator.load(ce_path)

>>> str(evaluator2.getPredictionCol())
'prediction'

New in version 2.3.0.

getAddressMeasure()  
  Gets the value of distanceMeasure  
  New in version 2.4.0.

getMetricName()  
  Gets the value of metricName or its default value.  
  New in version 2.3.0.

setDistanceMeasure(value)  
  Sets the value of distanceMeasure.  
  New in version 2.4.0.

setMetricName(value)  
  Sets the value of metricName.  
  New in version 2.3.0.

setParams(self, predictionCol="prediction", featuresCol="features", metric-
  Name="silhouette", distanceMeasure="squaredEuclidean")  
  Sets params for clustering evaluator.  
  New in version 2.3.0.
```
CHAPTER
TWENTYSIX

MAIN REFERENCE


p

pyspark.ml.classification, 406
pyspark.ml.clustering, 429
pyspark.ml.evaluation, 457
pyspark.ml.pipeline, 450
pyspark.ml.recommendation, 445
pyspark.ml.regression, 385
pyspark.ml.stat, 379
pyspark.ml.tuning, 452
INDEX

A
accuracy (pyspark.ml.classification.LogisticRegressionSummary attribute), 412
addGrid() (pyspark.ml.tuning.ParamGridBuilder method), 453
AFTSurvivalRegression (class in pyspark.ml.regression), 385
AFTSurvivalRegressionModel (class in pyspark.ml.regression), 387
aic (pyspark.ml.regression.GeneralizedLinearRegressionSummary attribute), 395
ALS (class in pyspark.ml.recommendation), 445
ALSModel (class in pyspark.ml.recommendation), 449
areaUnderROCAreaUnderROC
(pyspark.ml.classification.BinaryLogisticRegressionSummary attribute), 414
assignClusters() (pyspark.ml.clustering.PowerIterationClustering method), 443
avgMetrics (pyspark.ml.tuning.CrossValidatorModel attribute), 455

B
baseOn() (pyspark.ml.tuning.ParamGridBuilder method), 453
bestModel (pyspark.ml.tuning.CrossValidatorModel attribute), 455
bestModel (pyspark.ml.tuning.TrainValidationSplitModel attribute), 457
BinaryClassificationEvaluator (class in pyspark.ml.evaluation), 458
BinaryLogisticRegressionSummary (class in pyspark.ml.classification), 414

binaryLogisticRegressionTrainingSummary (class in pyspark.ml.classification), 415
BisectingKMeans (class in pyspark.ml.clustering), 429
BisectingKMeansModel (class in pyspark.ml.clustering), 431
BisectingKMeansSummary (class in pyspark.ml.clustering), 431
boundaries (pyspark.ml.regression.IsotonicRegressionModel attribute), 398
build() (pyspark.ml.tuning.ParamGridBuilder method), 453

C
ChiSquareTest (class in pyspark.ml.stat), 379
clusterCenters() (pyspark.ml.clustering.BisectingKMeansModel method), 431
clusterCenters() (pyspark.ml.clustering.KMeansModel method), 433
ClusteringEvaluator (class in pyspark.ml.evaluation), 460
coefficientMatrix (pyspark.ml.classification.LogisticRegressionModel attribute), 411
coefficients (pyspark.ml.classification.LinearSVCModel attribute), 407
coefficients (pyspark.ml.classification.LogisticRegressionModel attribute), 411
coefficients (pyspark.ml.regression.AFTSurvivalRegressionModel attribute), 387
evaluateEachIteration() (pyspark.ml.regression.GBTRegressionModel method), 390
evaluatedVariance (pyspark.ml.regression.LinearRegressionSummary attribute), 401
evaluator (pyspark.ml.evaluation), 457
gaussiansDF (pyspark.ml.clustering.GaussianMixtureModel attribute), 435
GBTClassifier (class in pyspark.ml.classification), 419
GBTClassificationModel (class in pyspark.ml.classification), 421
GBTRegressor (class in pyspark.ml.regression), 389
getAlpha() (pyspark.ml.recommendation.ALS method), 446
getBlockSize() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 429
getCensorCol() (pyspark.ml.regression.AFTSurvivalRegression method), 386
getCheckpointFiles() (pyspark.ml.clustering.DistributedLDAModel method), 441
getColdStartStrategy() (pyspark.ml.recommendation.ALS method), 446
getDistanceMeasure() (pyspark.ml.clustering.BisectingKMeans method), 430
getDistanceMeasure() (pyspark.ml.clustering.KMeans method), 432
getDistanceMeasure() (pyspark.ml.evaluation.ClusteringEvaluator method), 463
GaussianMixture (class in pyspark.ml.clustering), 434
GaussianMixtureModel (class in pyspark.ml.clustering), 435
GaussianMixtureSummary (class in pyspark.ml.clustering), 436
getDocConcentration() (pyspark.ml.clustering.LDA method), 437
getDstCol() (pyspark.ml.clustering.PowerIterationClustering method), 461

Index 471
method), 444
getEpsilon() (pyspark.ml.regression.LinearRegression method), 400
getFamily() (pyspark.ml.classification.LogisticRegression method), 409
getFeatureIndex() (pyspark.ml.regression.IsotonicRegression method), 398
getFinalStorageLevel() (pyspark.ml.recommendation.ALS method), 446
getImplicitPrefs() (pyspark.ml.recommendation.ALS method), 447
getInitialWeights() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 426
getInitMode() (pyspark.ml.clustering.KMeans method), 432
getInitMode() (pyspark.ml.clustering.PowerIterationClustering method), 444
getInitSteps() (pyspark.ml.clustering.KMeans method), 432
getIntermediateStorageLevel() (pyspark.ml.recommendation.ALS method), 447
getIsotonic() (pyspark.ml.regression.IsotonicRegression method), 398
getItemCol() (pyspark.ml.recommendation.ALS method), 447
getK() (pyspark.ml.clustering.BisectingKMeans method), 430
getK() (pyspark.ml.clustering.GaussianMixture method), 435
getK() (pyspark.ml.clustering.KMeans method), 433
getK() (pyspark.ml.clustering.PowerIterationClustering method), 444

getKeepLastCheckpoint () (pyspark.ml.clustering.LDA method), 438
getLayers () (pyspark.ml.classification.MultilayerPerceptronClassifier method), 426
getLearningDecay() (pyspark.ml.clustering.LDA method), 438
getLearningOffset() (pyspark.ml.clustering.LDA method), 438
getLink() (pyspark.ml.regression.GeneralizedLinearRegression method), 393
getLinkPower() (pyspark.ml.regression.GeneralizedLinearRegression method), 393
getLinkPredictionCol() (pyspark.ml.regression.GeneralizedLinearRegression method), 393
getLossType() (pyspark.ml.classification.GBTClassifier method), 419
getLossType() (pyspark.ml.regression.GBTRegressor method), 390
getLowerBoundsOnCoefficients() (pyspark.ml.classification.LogisticRegression method), 409
getLowerBoundsOnIntercepts() (pyspark.ml.classification.LogisticRegression method), 409
getMetricName() (pyspark.ml.evaluation.BinaryClassificationEvaluator method), 458
getMetricName() (pyspark.ml.evaluation.ClusteringEvaluator method), 461
getMetricName() (pyspark.ml.evaluation.MulticlassClassificationEvaluator method), 460
getMetricName() (pyspark.ml.evaluation.RegressionEvaluator method), 459
getMinDivisibleClusterSize() (pyspark.ml.clustering.BisectingKMeans method), 430
getMinDivisibleClusterSize() (pyspark.ml.clustering.PowerIterationClustering method), 444
getModelType() (pyspark.ml.classification.NaiveBayes method), 424

472 Index
getNonnegative() (pyspark.ml.recommendation.ALS method), 409
getNumFolds() (pyspark.ml.tuning.CrossValidator method), 447
getNumItemBlocks() (pyspark.ml.recommendation.ALS method), 447
getNumUserBlocks() (pyspark.ml.recommendation.ALS method), 447
getOffsetCol() (pyspark.ml.regression.GeneralizedLinearRegression method), 393
getOptimizeDocConcentration() (pyspark.ml.clustering.LDA method), 438
getOptimizer() (pyspark.ml.clustering.LDA method), 438
getQuantileProbabilities() (pyspark.ml.regression.AFTSurvivalRegression method), 386
getQuantilesCol() (pyspark.ml.regression.AFTSurvivalRegression method), 386
getRank() (pyspark.ml.recommendation.ALS method), 447
getRatingCol() (pyspark.ml.recommendation.ALS method), 447
getSmoothing() (pyspark.ml.classification.NaiveBayes method), 424
getSrcCol() (pyspark.ml.clustering.PowerIterationClustering method), 444
getStagePath() (pyspark.ml.pipeline.PipelineSharedReadWrite static method), 452
getStages() (pyspark.ml.pipeline.Pipeline method), 450
getStepSize() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 426
getSubsamplingRate() (pyspark.ml.clustering.LDA method), 438
getThreshold() (pyspark.ml.classification.LogisticRegression method), 409
getThresholds() (pyspark.ml.classification.LogisticRegression method), 410
getTopicConcentration() (pyspark.ml.clustering.LDA method), 438
getTopicDistributionCol() (pyspark.ml.clustering.LDA method), 438
getTrainRatio() (pyspark.ml.tuning.TrainValidationSplit method), 456
getUpperBoundsOnCoefficients() (pyspark.ml.classification.LogisticRegression method), 410
getUpperBoundsOnIntercepts() (pyspark.ml.classification.LogisticRegression method), 410
getUserCol() (pyspark.ml.recommendation.ALS method), 447
getVariancePower() (pyspark.ml.regression.GeneralizedLinearRegression method), 393

H

hasSummary (pyspark.ml.classification.LogisticRegressionModel attribute), 411
hasSummary (pyspark.ml.clustering.BisectingKMeansModel attribute), 431
hasSummary (pyspark.ml.clustering.GaussianMixtureModel attribute), 435
hasSummary (pyspark.ml.clustering.KMeansModel attribute), 433
hasSummary (pyspark.ml.regression.GeneralizedLinearRegressionModel attribute), 395
hasSummary (pyspark.ml.regression.LinearRegressionModel attribute), 400
intercept (pyspark.ml.classification.LinearSVCModel attribute), 407

Index

473
intercept

(\texttt{pyspark.ml.classification.LogisticRegressionModel.attribute}), 411

(\texttt{pyspark.ml.regression.AFTSurvivalRegressionModel.attribute}), 387

(\texttt{pyspark.ml.regression.LinearRegressionModel.attribute}), 400

(\texttt{pyspark.ml.regression.LinearRegressionSummary.attribute}), 401

(\texttt{pyspark.ml.regression.LinearRegressionTrainingSummary.attribute}), 403

(\texttt{pyspark.ml.regression.LinearSVC.attribute}), 406

(\texttt{pyspark.ml.pipeline.PipelineModelReader.method}), 451

(\texttt{pyspark.ml.pipeline.PipelineReader.method}), 452

(\texttt{pyspark.ml.pipeline.PipelineSharedReadWrite.static method}), 452

(\texttt{pyspark.ml.clustering.LocalLDAModel}), 441

(\texttt{pyspark.ml.classification.LogisticRegression}), 407

(\texttt{pyspark.ml.classification.LogisticRegressionModel}), 411

(\texttt{pyspark.ml.classification.LogisticRegressionSummary}), 412

(\texttt{pyspark.ml.classification.LogisticRegressionTrainingSummary}), 413

(\texttt{pyspark.ml.clustering.GaussianMixtureSummary.attribute}), 436

(\texttt{pyspark.ml.clustering.DistributedLDAModel.method}), 442

(\texttt{pyspark.ml.stat.KMeans.attribute}), 431

(\texttt{pyspark.ml.stat.KMeansModel}), 433

(\texttt{pyspark.ml.stat.KolmogorovSmirnovTest.attribute}), 381

(\texttt{pyspark.ml.regression.KLinearSummary.attribute}), 474

(\texttt{pyspark.ml.stat.KMeans.attribute}), 431

\text{K}

KMeans (\texttt{class in pyspark.ml.clustering}), 431

KMeansModel (\texttt{class in pyspark.ml.clustering}), 433

KolmogorovSmirnovTest (\texttt{class in pyspark.ml.stat}), 381

\text{L}

labelCol (\texttt{pyspark.ml.classification.LogisticRegressionSummary.attribute}), 412

labelCol (\texttt{pyspark.ml.regression.LinearRegressionSummary.attribute}), 401

labels (\texttt{pyspark.ml.classification.LogisticRegressionSummary.attribute}), 412

layers (\texttt{pyspark.ml.classification.MultilayerPerceptronClassificationModel.attribute}), 427

LDA (\texttt{class in pyspark.ml.clustering}), 436

LDAModel (\texttt{class in pyspark.ml.clustering}), 440

logLikelihood (\texttt{pyspark.ml.clustering.GaussianMixtureSummary.attribute}), 436

logLikelihood (\texttt{pyspark.ml.clustering.DistributedLDAModel.method}), 442

logPerplexity (\texttt{pyspark.ml.clustering.DistributedLDAModel.method}), 441

max (\texttt{pyspark.ml.stat.Summarizer static method}), 409

mean (\texttt{pyspark.ml.stat.Summarizer static method}), 383

meanAbsoluteError (\texttt{pyspark.ml.regression.LinearRegressionSummary.attribute}), 474
attribute, 396
predictions (pyspark.ml.regression.IsotonicRegressionModel attribute), 396
predictions (pyspark.ml.regression.LinearRegressionSummary attribute), 398
predictQuantiles() (pyspark.ml.regression.AFTSurvivalRegressionModel method), 387
probability (pyspark.ml.clustering.GaussianMixtureSummary attribute), 436
probabilityCol (pyspark.ml.clustering.GaussianMixtureSummary attribute), 436
pValues (pyspark.ml.regression.GeneralizedLinearRegressionTrainingSummary attribute), 397
pValues (pyspark.ml.regression.LinearRegressionSummary attribute), 402
pyspark.ml.classification (module), 406
pyspark.ml.clustering (module), 429
pyspark.ml.evaluation (module), 457
pyspark.ml.pipeline (module), 450
pyspark.ml.recommendation (module), 445
pyspark.ml.regression (module), 385
pyspark.ml.stat (module), 379
pyspark.ml.tuning (module), 452
R
r2 (pyspark.ml.regression.LinearRegressionSummary attribute), 402
r2adj (pyspark.ml.regression.LinearRegressionSummary attribute), 403
RandomForestClassificationModel (class in pyspark.ml.classification), 422
RandomForestClassifier (class in pyspark.ml.classification), 420
RandomForestRegressionModel (class in pyspark.ml.regression), 405
RandomForestRegressor (class in pyspark.ml.regression), 404
rank (pyspark.ml.recommendation.ALSModel attribute), 449
r2 (pyspark.ml.regression.LinearRegressionSummary attribute), 402
read() (pyspark.ml.pipeline.Pipeline class method), 450
read() (pyspark.ml.pipeline.PipelineModel class method), 451
read() (pyspark.ml.tuning.CrossValidator class method), 454
read() (pyspark.ml.tuning.CrossValidatorModel class method), 455
read() (pyspark.ml.tuning.TrainValidationSplit class method), 457
read() (pyspark.ml.tuning.TrainValidationSplitModel class method), 456
recallByLabel (pyspark.ml.classification.LogisticRegressionSummary attribute), 413
recallByThreshold (pyspark.ml.classification.BinaryLogisticRegressionSummary attribute), 415
recommendForAllItems() (pyspark.ml.recommendation.ALSModel method), 449
recommendForAllUsers() (pyspark.ml.recommendation.ALSModel method), 449
recommendForItemSubset() (pyspark.ml.recommendation.ALSModel method), 449
recommendForUserSubset() (pyspark.ml.recommendation.ALSModel method), 449
RegressionEvaluator (class in pyspark.ml.evaluation), 459
residualDegreeOfFreedom (pyspark.ml.regression.GeneralizedLinearRegressionSummary attribute), 396
residualDegreeOfFreedomNull (pyspark.ml.regression.GeneralizedLinearRegressionSummary attribute), 396
residuals (pyspark.ml.regression.LinearRegressionSummary attribute), 403
residuals() (pyspark.ml.regression.GeneralizedLinearRegressionSummary method), 396
roc (pyspark.ml.classification.BinaryLogisticRegressionSummary attribute), 415
rootMeanSquaredError (pyspark.ml.regression.LinearRegressionSummary attribute), 403

Run on Databricks Community Cloud, 11

S

saveImpl() (pyspark.ml.pipeline.PipelineModelWriter method), 451

saveImpl() (pyspark.ml.pipeline.PipelineSharedReadWrite static method), 452

scale (pyspark.ml.regression.AFTSurvivalRegressionModel attribute), 387

scale (pyspark.ml.regression.LinearRegressionModel attribute), 400

Set up Spark on Cloud, 26

setAlpha() (pyspark.ml.recommendation.ALS method), 447

setBlockSize() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 426

setCensorCol() (pyspark.ml.regression.AFTSurvivalRegression method), 386

setColdStartStrategy() (pyspark.ml.recommendation.ALS method), 447

setDistanceMeasure() (pyspark.ml.clustering.BisectingKMeans method), 430

setDistanceMeasure() (pyspark.ml.clustering.KMeans method), 433

setDistanceMeasure() (pyspark.ml.evaluation.ClusteringEvaluator method), 461

setDocConcentration() (pyspark.ml.clustering.LDA method), 438

setDstCol() (pyspark.ml.clustering.PowerIterationClustering method), 444

setEpsilon() (pyspark.ml.regression.LinearRegression method), 400

setFamily() (pyspark.ml.classification.LogisticRegression method), 410

setFamily() (pyspark.ml.regression.GeneralizedLinearRegression method), 394

setFeatureIndex() (pyspark.ml.regression.IsotonicRegression method), 398

setFeatureSubsetStrategy() (pyspark.ml.classification.GBTClassifier method), 419

setFeatureSubsetStrategy() (pyspark.ml.regression.GBTRegressor method), 390

setFeatureSubsetStrategy() (pyspark.ml.regression.RandomForestRegressor method), 405

setFinalStorageLevel() (pyspark.ml.recommendation.ALS method), 447

setImplicitPrefs() (pyspark.ml.recommendation.ALS method), 448

setInitialWeights() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 427

setInitMode() (pyspark.ml.clustering.KMeans method), 433

setInitMode() (pyspark.ml.clustering.PowerIterationClustering method), 444

setInitSteps() (pyspark.ml.clustering.KMeans method), 433

setIntermediateStorageLevel() (pyspark.ml.recommendation.ALS method), 448

setIsotonic() (pyspark.ml.regression.IsotonicRegression method), 398

setItemCol() (pyspark.ml.recommendation.ALS method), 448

setK() (pyspark.ml.clustering.BisectingKMeans method), 430
setK() (pyspark.ml.clustering.GaussianMixture method), 435
setK() (pyspark.ml.clustering.KMeans method), 433
setK() (pyspark.ml.clustering.LDA method), 438
setK() (pyspark.ml.clustering.PowerIterationClustering method), 444
setKeepLastCheckpoint() (pyspark.ml.clustering.LDA method), 439
setLayers() (pyspark.ml.clustering.LDA method), 427
setLearningDecay() (pyspark.ml.clustering.LDA method), 439
setLearningOffset() (pyspark.ml.clustering.LDA method), 439
setLink() (pyspark.ml.regression.GeneralizedLinearRegression method), 394
setLinkPower() (pyspark.ml.regression.GeneralizedLinearRegression method), 394
setLinkPredictionCol() (pyspark.ml.regression.GeneralizedLinearRegression method), 394
setLossType() (pyspark.ml.classification.GBTClassifier method), 419
setLossType() (pyspark.ml.classification.GBTRegressor method), 390
setLowerBoundsOnCoefficients() (pyspark.ml.classification.LogisticRegression method), 410
setLowerBoundsOnIntercepts() (pyspark.ml.classification.LogisticRegression method), 410
setMetricName() (pyspark.ml.evaluation.BinaryClassificationEvaluator method), 458
setMetricName() (pyspark.ml.evaluation.ClusteringEvaluator method), 461
setMetricName() (pyspark.ml.evaluation.MulticlassClassificationEvaluator method), 460
setMetricName() (pyspark.ml.evaluation.RegressionEvaluator method), 424
setMinDivisibleClusterSize() (pyspark.ml.clustering.BisectingKMeans method), 431
setModelType() (pyspark.ml.classification.NaiveBayes method), 424
setNonnegative() (pyspark.ml.recommendation.ALS method), 448
setNumBlocks() (pyspark.ml.recommendation.ALS method), 448
setNumItemBlocks() (pyspark.ml.recommendation.ALS method), 448
setNumUserBlocks() (pyspark.ml.recommendation.ALS method), 448
setOptimizeDocConcentration() (pyspark.ml.clustering.LDA method), 439
setOptimizer() (pyspark.ml.clustering.LDA method), 439
setParams() (pyspark.ml.classification.DecisionTreeClassifier method), 417
setParams() (pyspark.ml.classification.GBTClassifier method), 419
setParams() (pyspark.ml.classification.LinearSVC method), 407
setParams() (pyspark.ml.classification.LogisticRegression method), 410
setParams() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 427
setParams() (pyspark.ml.classification.NaiveBayes method), 424
setParams() (pyspark.ml.classification.NaiveBayes method), 424
setParams() (pyspark.ml.classification.OneVsRest method), 428
setParams() (pyspark.ml.classification.RandomForestClassifier method), 422
setParams() (pyspark.ml.clustering.BisectingKMeans method), 431
setParams() (pyspark.ml.clustering.GaussianMixture method), 435
setParams() (pyspark.ml.clustering.KMeans method), 433
setParams() (pyspark.ml.clustering.LDA method), 439
setParams() (pyspark.ml.clustering.PowerIterationClustering method), 444
setParams() (pyspark.ml.evaluation.BinaryClassificationEvaluator method), 459
setParams() (pyspark.ml.evaluation.ClusteringEvaluator method), 461
setParams() (pyspark.ml.evaluation.MulticlassClassificationEvaluator method), 460
setParams() (pyspark.ml.evaluation.RegressionEvaluator method), 459
setParams() (pyspark.ml.pipeline.Pipeline method), 451
setParams() (pyspark.ml.recommendation.ALS method), 448
setParams() (pyspark.ml.regression.AFTSurvivalRegression method), 386
setParams() (pyspark.ml.regression.IsotonicRegression method), 398
setParams() (pyspark.ml.regression.LinearRegression method), 400
setParams() (pyspark.ml.regression.RandomForestRegressor method), 405
setQuantileProbabilities() (pyspark.ml.regression.AFTSurvivalRegression method), 386
setQuantilesCol() (pyspark.ml.regression.AFTSurvivalRegression method), 386
setRank() (pyspark.ml.recommendation.ALS method), 448
setRatingCol() (pyspark.ml.recommendation.ALS method), 448
setSmoothing() (pyspark.ml.classification.NaiveBayes method), 424
setSrcCol() (pyspark.ml.clustering.PowerIterationClustering method), 444
setThreshold() (pyspark.ml.regression.IsotonicRegression method), 398
setThresholds() (pyspark.ml.classification.LogisticRegression method), 410
setTopicConcentration() (pyspark.ml.clustering.LDA method), 440
setTopicDistributionCol() (pyspark.ml.clustering.LDA method), 440
setStages() (pyspark.ml.pipeline.Pipeline method), 451
setStepSize() (pyspark.ml.classification.MultilayerPerceptronClassifier method), 427
setSubsamplingRate() (pyspark.ml.clustering.LDA method), 440
setThreshold() (pyspark.ml.classification.LogisticRegression method), 410
setThresholds() (pyspark.ml.classification.LogisticRegression method), 410
setTopicConcentration() (pyspark.ml.clustering.LDA method), 440
setTopicDistributionCol() (pyspark.ml.clustering.LDA method), 440
Learning Apache Spark with Python

setTrainRatio() (pyspark.ml.tuning.TrainValidationSplit
method), 456
setUpperBoundsOnCoefficients() (pyspark.ml.classification.LogisticRegression
method), 411
setUpperBoundsOnIntercepts() (pyspark.ml.classification.LogisticRegression
method), 411
setUserCol() (pyspark.ml.recommendation.ALS
method), 448
setVariancePower() (pyspark.ml.regression.GeneralizedLinearRegression
method), 394
solver (pyspark.ml.regression.GeneralizedLinearRegressionTrainingSummary
attribute), 397
subModels (pyspark.ml.tuning.CrossValidatorModel
attribute), 455
subModels (pyspark.ml.tuning.TrainValidationSplitModel
attribute), 457
Summarizer (class in pyspark.ml.stat), 382
summary (pyspark.ml.classification.LogisticRegressionModel
attr
attribute), 411
summary (pyspark.ml.clustering.BisectingKMeansModel
attribute), 431
summary (pyspark.ml.clustering.GaussianMixtureModel
attribute), 436
summary (pyspark.ml.clustering.KMeansModel
attribute), 434
summary (pyspark.ml.regression.GeneralizedLinearRegressionModel
attribute), 395
summary (pyspark.ml.regression.LinearRegressionModel
attribute), 400
summary() (pyspark.ml.stat.SummaryBuilder
method), 384
Summarizer (class in pyspark.ml.stat), 384

T

test() (pyspark.ml.stat.ChiSquareTest static
method), 379
test() (pyspark.ml.stat.KolmogorovSmirnovTest
static method), 381
theta (pyspark.ml.classification.NaiveBayesModel
attribute), 424
toLocal() (pyspark.ml.clustering.DistributedLDAModel
method), 442

U

UJ

userFactors (pyspark.ml.recommendation.ALSModel
attribute), 450

V

validateStages() (pyspark.ml.pipeline.PipelineSharedReadWrite
static method), 452
validationMetrics (pyspark.ml.tuning.TrainValidationSplitModel
attribute), 457
variance() (pyspark.ml.stat.SummaryBuilder static
method), 384
vocabSize() (pyspark.ml.clustering.LDAModel method), 441

W
weightedFalsePositiveRate (pyspark.ml.classification.LogisticRegressionSummary attribute), 413
weightedFMeasure() (pyspark.ml.classification.LogisticRegressionSummary method), 413
weightedPrecision (pyspark.ml.classification.LogisticRegressionSummary attribute), 413
weightedRecall (pyspark.ml.classification.LogisticRegressionSummary attribute), 413
weightedTruePositiveRate (pyspark.ml.classification.LogisticRegressionSummary attribute), 413
weights (pyspark.ml.classification.MultilayerPerceptronClassificationModel attribute), 427
weights (pyspark.ml.clustering.GaussianMixtureModel attribute), 436
write() (pyspark.ml.pipeline.Pipeline method), 451
write() (pyspark.ml.pipeline.PipelineModel method), 451
write() (pyspark.ml.tuning.CrossValidator method), 454
write() (pyspark.ml.tuning.CrossValidatorModel method), 455
write() (pyspark.ml.tuning.TrainValidationSplit method), 456
write() (pyspark.ml.tuning.TrainValidationSplitModel method), 457