Reference

DataCamp course

Course Description

- This course covers the fundamentals of Big Data via PySpark. Spark is “lightning fast cluster computing" framework for Big Data. It provides a general data processing platform engine and lets you run programs up to 100x faster in memory, or 10x faster on disk, than Hadoop.
- You’ll use PySpark, a Python packaged for spark programming and it's powerful, higher-level libraries such as SparkSQL, MLlib (for machine learning), etc., to interact with works of William Shakespeare, analyze Fifa football 2018 data and perform clustering of genomic datasets.

Introduction to Big Data analysis with Spark

Understanding SparkContext

A SparkContext represents the entry point to Spark functionality. It's like a key to your car. PySpark automatically creates a SparkContext for you and is exposed to the PySpark shell via variable sc.

In this simple exercise, you'll find out the attributes of the SparkContext in your PySpark shell which you'll be using for the rest of the course.

In [ ]:
```
# Print the version of SparkContext
print("The version of Spark Context in the PySpark shell is:", sc.version)

# Print the Python version of SparkContext
print("The Python version of Spark Context in the PySpark shell is:", sc.pythonVer)

# Print the master of SparkContext
print("The master of Spark Context in the PySpark shell is:", sc.master)
```

SparkContext in PySpark shell

- Load a simple list containing numbers ranging from 1 to 100 using the PySpark shell.
- The most important thing to understand here is that we are not creating any SparkContext object because PySpark automatically creates the SparkContext object named sc, by default in the PySpark shell.
- For the rest of this course, you'll have a SparkContext called sc available in PySpark shell.

In [ ]:
```
# Create a python list named numb
numb = range(1, 101)

# Load the list into PySpark
spark_data = sc.parallelize(numb)
```

Loading data in PySpark shell

In PySpark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. In the previous exercise, you have seen an example of loading list as parallelized collections and in this exercise, you'll load the data from a local file using PySpark shell.

Remember you already have a SparkContext sc and file_path variable (which is the path to the README.md file) already available in your workspace.

In [ ]:
```
# Load a local file into PySpark shell
lines = sc.textFile(file_path)
```

SparkContext's textFile() method is quite powerful for creating distributed collections of unstructured data which you'll see in the next chapter.

Lambda functions - map

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda and a list. The general syntax of map() function is map(condition_to_apply, list_of_inputs)

In this exercise, you'll be using lambda function inside the map() built-in function to square the number of each element in the list.

In [ ]:
```
# Print my_list in the console
print("Input list is:", my_list)

# Square each term in my_list using lambda function
squared_list_lambda = list(map(lambda x: x**2, my_list))

# Print the result of the map function
print("The squared numbers are:", squared_list_lambda)
```

Input list is: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The squared numbers are: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Lambda functions - filter

Another function that is used extensively in Python is the filter() function. This function takes two arguments - A condition and the list to filter. If you want to filter your list using some condition you use filter function. The general syntax of filter() function is filter(condition_to_apply, list_of_input).

In [ ]:
```
# Print my_list2 in the console
print("Input list is:", my_list2)

# Filter numbers divisible by 10
filtered_list = list(filter(lambda x: x%10 == 0, my_list2))

# Print the numbers divisible by 10
print("Numbers divisible by 10 are:", filtered_list)
```

```
Input list is: [10, 21, 31, 40, 51, 60, 72, 80, 93, 101]
Numbers divisible by 10 are: [10, 40, 60, 80]
```

Programming in PySpark RDD's

Parallelized collections

Since RDDs are fundamental and the core of Spark, it is important that you understand how to create them. In this exercise, you'll create your first RDD in PySpark. The simplest way to create the RDD is to take an already existing list and pass it to the SparkContext's parallelize() method. The base class of such created RDD is pyspark.RDD.

In [ ]:
```
# Create an RDD from a list of words
RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of RDD
print("The type of RDD is", type(RDD))
```

RDDs from External Datasets

PySpark can easily create RDDs from files that are stored in external storage devices such as HDFS (Hadoop Distributed File System), Amazon S3 buckets, etc. However, the most common method of creating RDD's is from files stored in your local file system. You can do this using SparkContext's textFile() method. This method takes a file path and reads it as a collection of lines. In this exercise, you'll create an RDD from a local file (README.md) and the file path (file_path) which is already available in your workspace.

In [ ]:
```
# Print the file_path
print("The file_path is", file_path)

# Create a fileRDD from file_path
fileRDD = sc.textFile(file_path)

# Check the type of fileRDD
print("The file type of fileRDD is", type(fileRDD))
```

Partitions in your data

SparkContext's textFile() method (and also the parallelize() method) takes an optional second argument called minPartitions for specifying the minimum number of partitions. In this exercise, you'll create an RDD named fileRDD_part with 5 partitions and then compare that with the default number of partitions in fileRDD that you created in the earlier exercise.

In [ ]:
```
# Check the number of partitions in fileRDD
print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD2 from file_path with 5 partitions
fileRDD_part = sc.textFile(file_path, 5)

# Check the number of partitions in fileRDD2
print("Number of partitions in fileRDD_part is", fileRDD_part.getNumPartitions())
```

Number of partitions in fileRDD is 2
Number of partitions in fileRDD_part is 5
Note that modifying the number of partitions may result in faster performance due to parallelization.

Map and Collect

The map() transformation takes in a function and applies it to each element in the RDD. map() transformations can be used to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers. In this simple exercise, you'll use map() transformation to cube each number of the numbRDD RDD that you created earlier. Next, you'll return all the elements to a variable and finally print out the output.

In [ ]:
```
# Create map() transformation to cube numbers
cubedRDD = numbRDD.map(lambda x: x ** 3)

# Collect the results
numbers_all = cubedRDD.collect()

# Print the numbers from numbers_all
for numb in numbers_all:
    print(numb)
```

1 8 27 64 125 216 343 512 729 1000

Filter and Count

filter() transformation is useful for filtering large datasets based on a keyword. The README.md file that you used earlier to create fileRDD consists of lines of text. For this exercise, you'll have to first filter out lines containing Spark, then count the total number of lines containing the keyword Spark and finally print the first 4 lines of the filtered RDD.

Remember, you already have a SparkContext sc, file_path and fileRDD available in your workspace.

In [ ]:
```
# Filter the fileRDD to select lines with Spark keyword
fileRDD_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines in fileRDD?
print("The total number of lines with the word Spark is", fileRDD_filter.count())

# Print the first four lines of fileRDD
for line in fileRDD_filter.take(4):
    print(line)
```

```
The total number of lines with the word Spark is 7
Examples for Learning Spark
Examples for the Learning Spark book. These examples require a number of libraries and as such have long
build files. We have also added a stand alone example with minimal dependencies and a small build file
These examples have been updated to run against Spark 1.3 so they may
be slightly different than the versions in your copy of "Learning Spark".
```

Note that the filter() operation does not mutate the existing fileRDD. Instead, it returns a pointer to an entirely new RDD.

ReduceBykey and Collect

One of the most popular pair RDD transformations is reduceByKey() which is used to aggregate data separately for each key. reduceByKey() operates on key, value pairs and **combines values with the same key**. In this exercise, you'll first create a pair RDD with a list of tuples, then add the values with the same key using reduceByKey() transformation and finally print out the result.

Remember, you already have a SparkContext sc available in your workspace.

In [ ]:
```
# Create a pair RDD named `rdd` with tuples
Rdd = sc.parallelize([(1,2), (3,4), (3,6), (4,5)])

# Transform the contents into a variable
Rdd_Reduced = Rdd.reduceByKey(lambda x, y: x + y)

# Iterate over the result and print the output
for num in Rdd_Reduced.collect():
    print(num)
```

(4, 5) (1, 2) (3, 10)

SortByKey and Collect

Many times it is useful to sort the key/value pair (pair RDD) based on the key (for example word count). The sortByKey() transformation returns an RDD sorted by key in ascending or descending order. In this exercise, you'll sort the pair RDD Rdd_Reduced that you created using reduceByKey() transformation in the previous exercise into descending order and then finally print the final output.

Remember, you already have a SparkContext sc, Rdd and Rdd_Reduced available in your workspace.

In [ ]:
```
# Sort the reduced RDD with the key by descending order
Rdd_Reduced_Sort = Rdd_Reduced.sortByKey(ascending=False)

# Iterate over the result and print the output
for num in Rdd_Reduced_Sort.collect():
      print(num)
```

(4, 5) (3, 10) (1, 2)

CountingBykeys

Unlike reduceByKey() and sortByKey(), countByKey() **is an action and not a transformation on the pair RDD. Just as a note, Transformations are RDD operations that create new RDDs, whereas Actions are operations that perform some computations**. In this simple exercise, you'll use the Rdd that you created earlier and count the number of unique keys in that pair RDD.

Remember, you already have a SparkContext sc and Rdd available in your workspace.

In [ ]:
```
# Transform the rdd with countByKey()
total = Rdd.countByKey()

# What is the type of total?
print("The type of total is", type(total))

# Iterate over the total and print the output
for k, v in total.items():
  print("key", k, "has", v, "counts")
```

The type of total is key 1 has 1 counts
key 3 has 2 counts
key 4 has 1 counts

Create a base RDD and transform it

The volume of unstructured data (log lines, images, binary files) in existence is growing dramatically, and PySpark is an excellent framework for analyzing this type of data through RDDs. In this 3 part exercise, you will write code that calculates the most common words from Complete Works of William Shakespeare.

Here are the brief steps for writing the word counting program:

- Create a base RDD from Complete_Shakespeare.txt file.
- Use RDD transformation to create a long list of words from each element of the base RDD.
- Remove stop words from your data.
- Create pair RDD where each element is a pair tuple of ('w', 1)
- Group the elements of the pair RDD by key (word) and add up their values.
- Swap the keys (word) and values (counts) so that keys is count and value is the word.
- Finally, sort the RDD by descending order. In this first exercise, you'll first create a base RDD from Complete_Shakespeare.txt file and transform it to create a long list of words.

Remember, you already have a SparkContext sc already available in your workspace. A file_path variable (which is the path to the Complete_Shakespeare.txt file) is also loaded for you.

In [ ]:
```
# Create a baseRDD from the file path
baseRDD = sc.textFile(file_path)

# Split the lines of baseRDD into words
splitRDD = baseRDD.flatMap(lambda x: x.split())
#unlike map, flatMap can give multiple output for a same input. It is not a function in math.

# Count the total number of words
print("Total number of words in splitRDD:", splitRDD.count())
```

output: Total number of words in splitRDD: 904061

You have succesfully created and transformed RDD from unstructured data.

Remove stop words and reduce the dataset

In the third step, you'll remove stop words from your data. Stop words are common words that are often uninteresting. For example "I", "the", "a" etc., are stop words. You can remove many obvious stop words with a list of your own. But for this exercise, you will just remove the stop words from a curated list stop_words provided to you in your environment.

After removing stop words, you'll next create a pair RDD where each element is a pair tuple (k, v) where k is the key and v is the value. In this example, pair RDD is composed of (w, 1) where w is for each word in the RDD.

Remember you already have a SparkContext sc and splitRDD available in your workspace.

In [ ]:
```
# Convert the words in lower case and remove stop words
splitRDD_no_stop = splitRDD.filter(lambda x: x.lower() not in stop_words)

# Create a tuple of the word and 1
splitRDD_no_stop_words = splitRDD_no_stop.map(lambda w: (w, 1))

# Count of the number of occurences of each word
resultRDD = splitRDD_no_stop_words.reduceByKey(lambda x, y: x + y) #sear earlier example
```

Printing word frequencies

Finally, you'll return the word frequencies using the take() action. You could have used the collect() action but as a best practice, it is not recommended as collect() returns all the elements from your RDD. You'll use take(N) instead, to return N elements from your RDD.

What if we want to return the top 10 words? For this first, you'll need to swap the keys (word) and values (counts) so that keys is count and value is the word.

You already have a SparkContext sc and resultRDD available in your workspace.

In [ ]:
```
# Display the first 10 words and their frequencies
for word in resultRDD.take(10):
    print(word)

# Swap the keys and values
resultRDD_swap = resultRDD.map(lambda x: (x[1], x[0])) # a nice way to swap.

# Sort the keys in descending order
resultRDD_swap_sort = resultRDD_swap.sortByKey(ascending=False)

# Show the top 10 most frequent words
for word in resultRDD_swap_sort.take(10):
    print(word)
```

('Quince', 1) ('Corin,', 2) ('circle', 10) ('enrooted', 1) ('divers', 20) ('Doubtless', 2) ('undistinguishable,', 1) ('widowhood,', 1) ('incorporate.', 1) ('rare,', 10)

```
(4247, 'thou')
(3630, 'thy')
(3018, 'shall')
(2046, 'good')
(1974, 'would')
(1926, 'Enter')
(1780, 'thee')
(1737, "I'll")
(1614, 'hath')
(1452, 'like')
```

PySpark SQL & DataFrames

RDD to DataFrame

- Similar to RDDs, DataFrames are **immutable and distributed data structures** in Spark. In PySpark, **a DataFrame is actually a wrapper around RDD**. Even though RDDs are a fundamental data structure in Spark, working with data in DataFrame is easier than RDD most of the time and so understanding of how to convert RDD to DataFrame is necessary.
- First make an RDD using the sample_list which contains the list of tuples ('Mona',20), ('Jennifer',34),('John',20), ('Jim',26) with each tuple contains the name of the person and their age. Next, create a DataFrame using the RDD and the schema (which is the list of 'Name' and 'Age') and finally confirm the output as PySpark DataFrame.

In [ ]:
```
sample_list = [('Mona', 20), ('Jennifer', 34), ('John', 20), ('Jim', 26)]

# Create a RDD from the list
rdd = sc.parallelize(sample_list)

# Create a PySpark DataFrame
names_df = spark.createDataFrame(rdd, schema=['Name', 'Age'])

# Check the type of people_df
print("The type of names_df is", type(names_df))
```

The type of names_df is <class 'pyspark.sql.dataframe.DataFrame'

Creating DataFrames from RDDs may not be a common practise but helps in certain circumstances.

Loading CSV into DataFrame

Loading data from CSV file is the most common method of creating DataFrames.

In [ ]:
```
# Create an DataFrame from people.csv file
people_df = spark.read.csv(file_path, header=True, inferSchema=True)

print("The type of people_df is", type(people_df))
```

```
### Inspecting data in PySpark DataFrame
Inspect the data in the people_df DataFrame that you have created in the previous exercise using basic
DataFrame operators.
```

In [ ]:
```
# Print the first 10 observations
people_df.show(10) #similar to head(10)

# Count the number of rows
print("There are {} rows in the people_df DataFrame.".format(people_df.count()))

# Count the number of columns and their names
print("There are {} columns in the people_df DataFrame and their names are
      {}".format(len(people_df.columns), people_df.columns))
```

PySpark DataFrame subsetting and cleaning

After data inspection, it is often necessary to clean the data which mainly involves subsetting, renaming the columns, removing duplicated rows etc., PySpark DataFrame API provides several operators to do this. In this exercise, your job is to subset 'name', 'sex' and 'date of birth' columns from people_df DataFrame, remove any duplicate rows from that dataset and count the number of rows before and after duplicates removal step.

In [ ]:
```
# Select name, sex and date of birth columns
people_df_sub = people_df.select('name', 'sex', 'date of birth')

# Print the first 10 observations from people_df_sub
people_df_sub.show(10)

# Remove duplicate entries from people_df_sub
people_df_sub_nodup = people_df_sub.dropDuplicates()

# Count the number of rows
print("There were {} rows before removing duplicates, and {}
      rows after removing duplicates".format(people_df_sub.count(), people_df_sub_nodup.count()))
```

```
+----------------+------+-------------+
|            name|   sex|date of birth|
+----------------+------+-------------+
|  Penelope Lewis|female|   1990-08-31|
|   David Anthony|  male|   1971-10-14|
|       Ida Shipp|female|   1962-05-24|
|    Joanna Moore|female|   2017-03-10|
```

```
|  Lisandra Ortiz|female|   2020-08-05|
|   David Simmons|  male|   1999-12-30|
|   Edward Hudson|  male|   1983-05-09|
|    Albert Jones|  male|   1990-09-13|
|Leonard Cavender|  male|   1958-08-08|
|  Everett Vadala|  male|   2005-05-24|
+----------------+------+-------------+
only showing top 10 rows

There were 100000 rows before removing duplicates, and 99998 rows after removing duplicates
```

Filtering your DataFrame

In [ ]:
```
people_df_female = people_df.filter(people_df.sex == "female")

people_df_male = people_df.filter(people_df.sex == "male")

print("There are {} rows in the people_df_female DataFrame and {}
      rows in the people_df_male DataFrame".format(people_df_female.count(), people_df_male.count()))
```

Running SQL Queries Programmatically

DataFrames can easily be manipulated using SQL queries in PySpark. The sql() function on a SparkSession enables applications to run SQL queries programmatically and returns the result as another DataFrame.

In [ ]:
```
# Create a tempoaray table "people". See other notes for why we need this.
people_df.createOrReplaceTempView("people")

# Construct a query to select the names of the people
query = '''SELECT name FROM people'''

# Assign the result of Spark's query to people_df_names
people_df_names = spark.sql(query)

# Print the top 10 names of the people
people_df_names.show(10)
```

```
+----------------+
|            name|
+----------------+
|  Penelope Lewis|
|   David Anthony|
|       Ida Shipp|
|    Joanna Moore|
|  Lisandra Ortiz|
|   David Simmons|
|   Edward Hudson|
|    Albert Jones|
|Leonard Cavender|
|  Everett Vadala|
+----------------+
only showing top 10 rows
```

Spark SQL operations generally return DataFrames. This means you can freely mix DataFrames and SQL.

SQL queries for filtering Table

In [ ]:
```
# Filter the people table to select female sex
people_female_df = spark.sql('SELECT * FROM people WHERE sex=="female"')

# Filter the people table DataFrame to select male sex
people_male_df = spark.sql('SELECT * FROM people WHERE sex=="male"')

# Count the number of rows in both people_df_female and people_male_df DataFrames
print("There are {} rows in the people_female_df and {} rows in the people_male_df
      DataFrames".format(people_female_df.count(), people_male_df.count()))
```

PySpark DataFrame visualization

Print the column names of names_df DataFrame created earlier, then convert the names_df to Pandas DataFrame and finally plot the contents as horizontal bar plot with names of the people on the x-axis and their age on the y-axis.

In [ ]:
```
print("The column names of names_df are", names_df.columns)

# Convert to Pandas DataFrame
df_pandas = names_df.toPandas()

# Create a bar plot
df_pandas.plot(kind='barh', x='Name', y='Age', colormap='winter_r')
plt.show()
```

Part 1: Create a DataFrame from CSV file

In [ ]:
```
# Load the Dataframe
fifa_df = spark.read.csv(file_path, header=True, inferSchema=True)

# Check the schema of columns
fifa_df.printSchema()

# Show the first 10 observations
fifa_df.show(10)

# Print the total number of rows
print("There are {} rows in the fifa_df DataFrame".format(fifa_df.count()))
```

Part 2: SQL Queries on DataFrame

In [ ]:
```
# Create a temporary view of fifa_df
fifa_df.createOrReplaceTempView('fifa_df_table')

# Construct the "query"
query = '''SELECT Age FROM fifa_df_table WHERE Nationality == "Germany"'''

# Apply the SQL "query"
fifa_df_germany_age = spark.sql(query)

# Generate basic stastics
fifa_df_germany_age.describe().show()
```

```
+-------+-----------------+
|summary|              Age|
+-------+-----------------+
|  count|             1140|
|   mean|24.20263157894737|
| stddev|4.197096712293752|
|    min|               16|
|    max|               36|
+-------+-----------------+
```

Notice how consise SQL queries are compared to DataFrame operations.

Part 3: Data visualization

In [ ]:
```
# Convert fifa_df to fifa_df_germany_age_pandas DataFrame
fifa_df_germany_age_pandas = fifa_df_germany_age.toPandas()

# Plot the 'Age' density of Germany Players
fifa_df_germany_age_pandas.plot(kind='density')
plt.show()
```

Machine Learning with PySpark MLlib

PySpark MF libraries

What kind of data structures does pyspark.mllib built-in library support in Spark?
Answer:
pyspark.mllib can only support RDDs unless you change DataFrames to RDDs.

PySpark MLlib algorithms

Before using any Machine learning algorithms in PySpark shell, you'll have to import the submodules of pyspark.mllib library and then chose the appropriate class that is needed for a specific machine learning task.

In [ ]:
```
# Import the library for ALS
from pyspark.mllib.recommendation import ALS

# Import the library for Logistic Regression
from pyspark.mllib.classification import LogisticRegressionWithLBFGS

# Import the library for Kmeans
from pyspark.mllib.clustering import KMeans
```

Loading Movie Lens dataset into RDDs

- **Collaborative filtering** is a technique for recommender systems wherein users' ratings and interactions with various products are used to recommend new ones. In this 3-part exercise, your goal is to develop a simple movie recommendation system using PySpark MLlib using a subset of MovieLens 100k dataset.
- Load the MovieLens data (ratings.csv) into RDD and from each line in the RDD which is formatted as userId, movieId, rating, timestamp, you'll need to map the MovieLens data to a Ratings object (userID, productID, rating) after removing timestamp column and finally you'll split the RDD into training and test RDDs.
- file_path variable (which is the path to the ratings.csv file), and ALS class are already available in your workspace.

In [ ]:
```
# Load the data into RDD
data = sc.textFile(file_path)

# Split the RDD
ratings = data.map(lambda l: l.split(','))

# Transform the ratings RDD
ratings_final = ratings.map(lambda line: Rating(int(line[0]), int(line[1]), float(line[2])))

# Split the data into training and test
training_data, test_data = ratings_final.randomSplit([0.8, 0.2])
```

Model training and predictions

- Train the ALS algorithm using the training data. PySpark MLlib's ALS algorithm has the following mandatory parameters - rank (the number of latent factors in the model) and iterations (number of iterations to run). After training the ALS model, you can use the model to predict the ratings from the test data. For this, you will provide the user and item columns from the test dataset and finally print the first 2 rows of predictAll() output.

In [ ]:
```
# Create the ALS model on the training data
model = ALS.train(training_data, rank=10, iterations=10)

# Drop the ratings column
testdata = ratings_final.map(lambda p: (p[0], p[1]))

# Predict the model
predictions = model.predictAll(testdata)

# Print the first rows of the RDD
predictions.take(2)
```

Model evaluation using MSE

- After generating the predicted ratings from the test data using ALS model, in this final part of the exercise, you'll prepare the data for calculating Mean Square Error (MSE) of the model. The MSE is the average value of (original rating – predicted rating)^2 for all users and indicates the absolute fit of the model to the data.

- To do this, first, you'll organize both the ratings and prediction RDDs to make a tuple of ((user, product), rating)), then join the ratings RDD with prediction RDD and finally apply a squared difference function along with mean() to get the MSE.

In [ ]:
```
# Prepare ratings data
rates = ratings_final.map(lambda r: ((r[0], r[1]), r[2]))

# Prepare predictions data
preds = predictions.map(lambda r: ((r[0], r[1]), r[2]))

# Join the ratings data with predictions data
rates_and_preds = rates.join(preds)

# Calculate and print MSE
MSE = rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
print("Mean Squared Error of the model for the test data is {:.2f}".format(MSE))
```

Mean Squared Error of the model for the test data is 0.46

Loading spam and non-spam data

Logistic Regression is a popular method to predict a categorical response. Probably one of the most common applications of the logistic regression is the message or email spam classification. In this 3-part exercise, you'll create an email spam classifier with logistic regression using Spark MLlib. Here are the brief steps for creating a spam classifier.

Create an RDD of strings representing email.
Run MLlib's feature extraction algorithms to convert text into an RDD of vectors.
Call a classification algorithm on the RDD of vectors to return a model object to classify new points.
Evaluate the model on a test dataset using one of MLlib's evaluation functions.

- Load the 'spam' and 'ham' (non-spam) files into RDDs, split the emails into individual words and look at the first element in each of the RDD.
- file_path_spam variable (which is the path to the spam file) and file_path_ham (which is the path to the 'non-spam' file) is already available in your workspace.

In [ ]:
```
# Load the datasets into RDDs
spam_rdd = sc.textFile(file_path_spam)
non_spam_rdd = sc.textFile(file_path_non_spam)

# Split the email messages into words
spam_words = spam_rdd.map(lambda email: email.split(' '))
non_spam_words = non_spam_rdd.map(lambda email: email.split(' '))

# Print the first element in the split RDD
print("The first element in spam_words is", spam_words.first())
print("The first element in non_spam_words is", non_spam_words.first())
```

```
The first element in spam_words is ['You', 'have', '1', 'new', 'message.', 'Please', 'call', '0871240020
0.']
The first element in non_spam_words is ['Rofl.', 'Its', 'true', 'to', 'its', 'name']
```

Feature hashing and LebelPoint

- After splitting the emails into words, our raw data set of 'spam' and 'non-spam' is currently composed of 1-line messages consisting of spam and non-spam messages. In order to classify these messages, we need to convert text into features.
- First create a HashingTF() instance to map text to vectors of 200 features, then for each message in 'spam' and 'non-spam' files you'll split them into words, and each word is mapped to one feature. These are the features that will be used to decide whether a message is 'spam' or 'non-spam'.
- Next, you'll create labels for features. For a valid message, the label will be 0 (i.e. the message is not spam) and for a 'spam' message, the label will be 1 (i.e. the message is spam). Finally, you'll combine both the labeled datasets.
- spam_words and non_spam_words variables are already available in your workspace.

In [ ]:
```
# Create a HashingTf instance with 200 features
tf = HashingTF(numFeatures=200)

# Map each word to one feature
spam_features = tf.transform(spam_words)
non_spam_features = tf.transform(non_spam_words)

# Label the features: 1 for spam, 0 for non-spam
spam_samples = spam_features.map(lambda features:LabeledPoint(1, features))
non_spam_samples = non_spam_features.map(lambda features:LabeledPoint(0, features))

# Combine the two datasets
samples = spam_samples.union(non_spam_samples)
```

Feature hashing and LabeledPoints are quite powerful for text based classification in PySpark MLlib!

Logistic Regression model training

After creating labels and features for the data, we're ready to build a model that can learn from it (training). But before you train the model, you'll split the combined dataset into training and testing dataset because it can assign a probability of being spam to each data point. We can then decide to classify messages as spam or not, depending on how high the probability.

In this final part of the exercise, you'll split the data into training and test, run Logistic Regression on the training data, apply the same HashingTF() feature transformation to get vectors on a positive example (spam) and a negative one (non-spam) and finally check the accuracy of the model trained.

In [ ]:
```
# Split the data into training and testing
train_samples,test_samples = samples.randomSplit([0.8, 0.2])

# Train the model
model = LogisticRegressionWithLBFGS.train(train_samples)

# Predict labels from features using the trained model
predictions = model.predict(test_samples.map(lambda x: x.features))

# Combine original labels with the predicted labels
labels_and_preds = test_samples.map(lambda x: x.label).zip(predictions)

# Check the accuracy of the model on the test data
accuracy = labels_and_preds.filter(lambda x: x[0] == x[1]).count() / float(test_samples.count())
print("Model accuracy : {:.2f}".format(accuracy))
```

Model accuracy : 0.82

You sucessfully created a spam classifier in just a few steps. Your classifier predicted about 82% of the labels correctly. Can you think of a way to improve this accuracy?

Loading and parsing the 5000 points data

In this 3 part exercise, you'll find out how many clusters are there in a dataset containing 5000 rows and 2 columns. For this you'll first load the data into an RDD, parse the RDD based on the delimiter, run the KMeans model, evaluate the model and finally visualize the clusters.

In the first part, you'll load the data into RDD, parse the RDD based on the delimiter and convert the string type of the data to an integer.

file_path variable (which is the path to the 5000_points.txt file) is already available in your workspace.

In [ ]:
```
# Load the dataset into a RDD
clusterRDD = sc.textFile(file_path)

# Split the RDD based on "\t"
rdd_split = clusterRDD.map(lambda x: x.split("\t"))

# Transform the split RDD by creating a list of integers
rdd_split_int = rdd_split.map(lambda x: [int(x[0]), int(x[1])])

# Count the number of rows in RDD
print("There are {} rows in rdd_split_int.".format(rdd_split_int.count()))
```

```
There are 5000 rows in rdd_split_int.
```

K-means training

Now that the RDD is ready for training, in the second part of the exercise, you'll train the RDD with PySpark's MLlib's KMeans algorithm. The algorithm is somewhat naive--it clusters the data into k clusters, even if k is not the right number of clusters to use. Therefore, when using k-means clustering, the most important parameter is a target number of clusters to generate, k. In practice, you rarely know the “true” number of clusters in advance, so the best practice is to try several values of k until the average intracluster distance stops decreasing dramatically

In this 2nd part, you'll test with k's ranging from 13 to 16 and use the elbow method to chose the correct k. The idea of the elbow method is to run k-means clustering on the dataset for a range of values of k, calculate Within Set Sum of Squared Error (WSSSE, this function is already provided to you) and select the best k based on the sudden drop in WSSSE. Finally, you'll retrain the model with the best k (15 in this case) and print the centroids (cluster centers).

In [ ]:
```
# Train the model with clusters from 13 to 16 and computer error
for clst in range(13, 17):
    model = KMeans.train(rdd_split_int, clst, seed=1)
    WSSSE = rdd_split_int.map(lambda point: error(point)).reduce(lambda x, y: x + y)
    print("The cluster {} has Within Set Sum of Squared Error {}".format(clst, WSSSE))

# Train the model again with the best k
model = KMeans.train(rdd_split_int, k=15, seed=1)

# Get cluster centers
cluster_centers = model.clusterCenters
```

```
For real world data, you should train the model with a wide range of K values.
```

Visualizing clusters

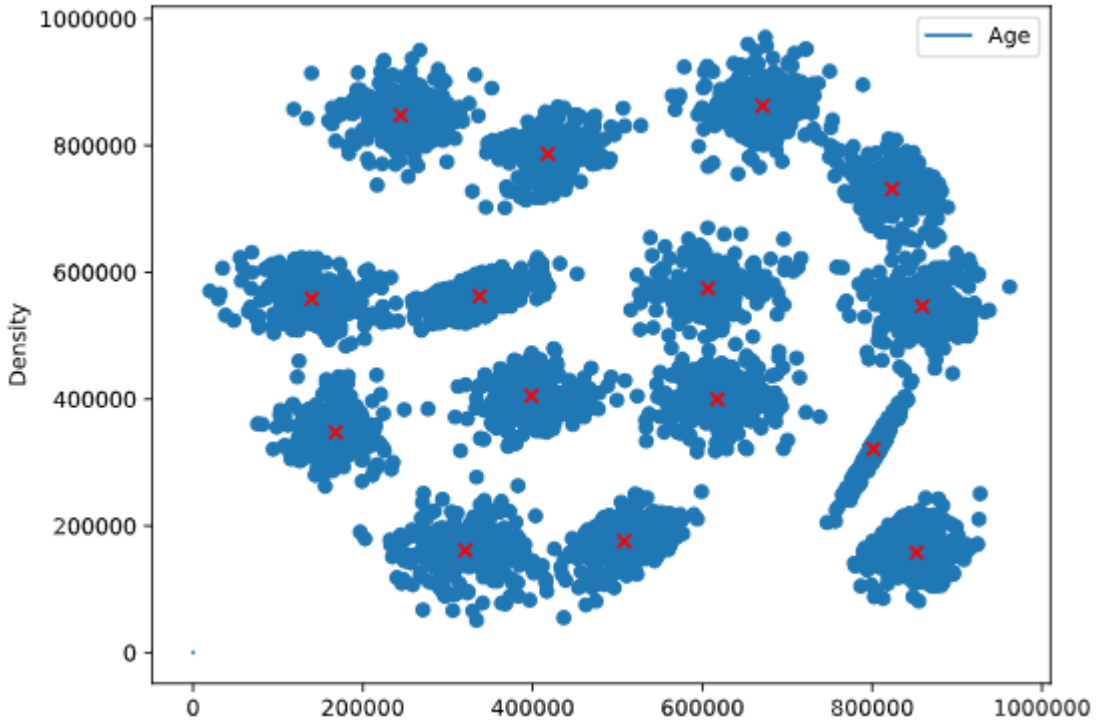
After KMeans model training with an optimum K value (K = 15), in this final part of the exercise, you will visualize the clusters and their cluster centers (centroids) and see if they overlap with each other. For this, you'll first convert rdd_split_int RDD into spark DataFrame and then into Pandas DataFrame for plotting. Similarly, you'll convert cluster_centers into Pandas DataFrame. Once the DataFrames are created, you'll use matplotlib library to create scatter plots.

In [ ]:
```
# Convert rdd_split_int RDD into Spark DataFrame
rdd_split_int_df = spark.createDataFrame(rdd_split_int, schema=["col1", "col2"])

# Convert Spark DataFrame into Pandas DataFrame
rdd_split_int_df_pandas = rdd_split_int_df.toPandas()

# Convert cluster_centers into Panda DataFrame
cluster_centers_pandas = pd.DataFrame(cluster_centers, columns=["col1", "col2"])

# Create a scatter plot
plt.scatter(rdd_split_int_df_pandas["col1"], rdd_split_int_df_pandas["col2"])
plt.scatter(cluster_centers_pandas["col1"], cluster_centers_pandas["col2"], color="red", marker="x")
plt.show()
```


Age
Density
1000000
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1000000