Get to know the SparkContext.

* Call print() on sc to verify there's a SparkContext in your environment.
* print() sc.version to see what version of Spark is running on your cluster.

# Verify SparkContext

print(sc)

# Print Spark version

print(sc.version)

* Import SparkSession from pyspark.sql.
* Make a new SparkSession called my\_spark using SparkSession.builder.getOrCreate().
* Print my\_spark to the console to verify it's a SparkSession

# Import SparkSession from pyspark.sql

from pyspark.sql import SparkSession

# Create my\_spark

my\_spark = SparkSession.builder.getOrCreate()

# Print my\_spark

print(my\_spark)

See what tables are in your cluster by calling spark.catalog.listTables() and printing the result!

# Print the tables in the catalog

print(spark.catalog.listTables())

* Use the .sql() method to get the first 10 rows of the flights table and save the result to flights10. The variable query contains the appropriate SQL query.
* Use the DataFrame method .show() to print flights10.

# Don't change this query

query = "FROM flights SELECT \* LIMIT 10"

# Get the first 10 rows of flights

flights10 = spark.sql(query)

# Show the results

flights10.show()

* Run the query using the .sql() method. Save the result in flight\_counts.
* Use the .toPandas() method on flight\_counts to create a pandas DataFrame called pd\_counts.
* Print the .head() of pd\_counts to the console.

# Don't change this query

query = "SELECT origin, dest, COUNT(\*) as N FROM flights GROUP BY origin, dest"

# Run the query

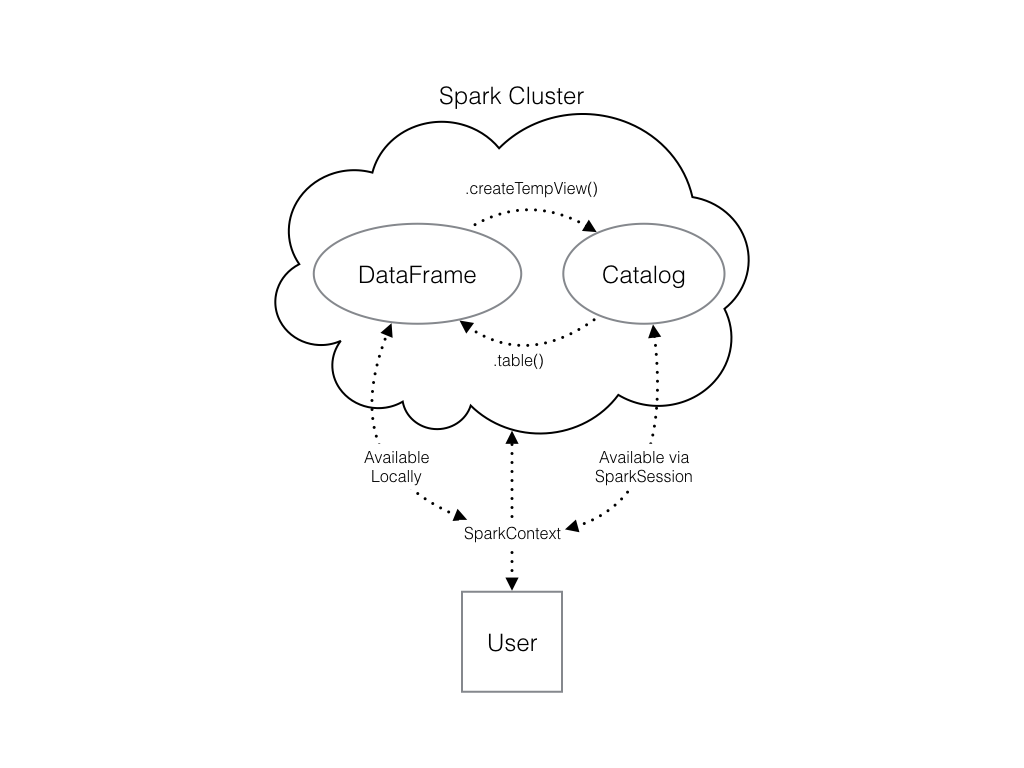
flight\_counts = spark.sql(query)

# Convert the results to a pandas DataFrame

pd\_counts = flight\_counts.toPandas()

# Print the head of pd\_counts

print(pd\_counts.head())



* The code to create a pandas DataFrame of random numbers has already been provided and saved under pd\_temp.
* Create a Spark DataFrame called spark\_temp by calling the .createDataFrame() method with pd\_temp as the argument.
* Examine the list of tables in your Spark cluster and verify that the new DataFrame is *not* present. Remember you can use spark.catalog.listTables() to do so.
* Register spark\_temp as a temporary table named "temp" using the .createOrReplaceTempView() method. Remember that the table name is set including it as the only argument!

# Create pd\_temp

pd\_temp = pd.DataFrame(np.random.random(10))

# Create spark\_temp from pd\_temp

spark\_temp = spark.createDataFrame(pd\_temp)

# Examine the tables in the catalog

print(spark.catalog.listTables())

# Add spark\_temp to the catalog

spark\_temp.createOrReplaceTempView(

"temp")

# Examine the tables in the catalog again

print(spark\_temp)

* Use the .read.csv() method to create a Spark DataFrame called airports
  + The first argument is file\_path
  + Pass the argument header=True so that Spark knows to take the column names from the first line of the file.
* Print out this DataFrame by calling .show()

# Don't change this file path

file\_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data

airports = spark.read.csv(file\_path, header=True)

# Show the data

airports.show()

* Use the spark.table() method with the argument "flights" to create a DataFrame containing the values of the flights table in the .catalog. Save it as flights.
* Show the head of flights using flights.show(). The column air\_time contains the duration of the flight in minutes.
* Update flights to include a new column called duration\_hrs, that contains the duration of each flight in hours.

# Create the DataFrame flights

flights = spark.table("flights")

# Show the head

flights.show()

# Add duration\_hrs

flights = flights.withColumn("duration\_hrs", flights.air\_time/60)

Which of the following queries returns a table of tail numbers and destinations for flights that lasted more than 10 hours?

SELECT dest, tail\_num FROM flights WHERE air\_time > 600;

* Use the .filter() method to find all the flights that flew over 1000 miles two ways:
  + First, pass a SQL **string** to .filter() that checks whether the distance is greater than 1000. Save this as long\_flights1.
  + Then pass a column of boolean values to .filter() that checks the same thing. Save this as long\_flights2.
* Use .show() to print heads of both DataFrames and make sure they're actually equal!

# Filter flights by passing a string

long\_flights1 = flights.filter("distance>1000")

# Filter flights by passing a column of boolean values

long\_flights2 = flights.filter(flights.distance>1000)

# Print the data to check they're equal

long\_flights1.show()

long\_flights2.show()

* Select the columns tailnum, origin, and dest from flights by passing the column names as strings. Save this as selected1.
* Select the columns origin, dest, and carrier using the df.colName syntax and then filter the result using both of the filters already defined for you (filterA and filterB) to only keep flights from SEA to PDX. Save this as selected2

# Select the first set of columns

selected1 = flights.select("tailnum", "origin", "dest")

# Select the second set of columns

temp = flights.select(flights.origin, flights.dest, flights.carrier)

# Define first filter

filterA = flights.origin == "SEA"

# Define second filter

filterB = flights.dest == "PDX"

# Filter the data, first by filterA then by filterB

selected2 = temp.filter(filterA).filter(filterB)

Create a table of the average speed of each flight both ways.

* Calculate average speed by dividing the distance by the air\_time (converted to hours). Use the .alias() method name this column "avg\_speed". Save the output as the variable avg\_speed.
* Select the columns "origin", "dest", "tailnum", and avg\_speed (without quotes!). Save this as speed1.
* Create the same table using .selectExpr() and a string containing a SQL expression. Save this as speed2.

# Define avg\_speed

avg\_speed = (flights.distance/(flights.air\_time/60)).alias("avg\_speed")

# Select the correct columns

speed1 = flights.select("origin", "dest", "tailnum", avg\_speed)

# Create the same table using a SQL expression

speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air\_time/60) as avg\_speed")

* Find the length of the shortest (in terms of distance) flight that left PDX by first .filter()ing and using the .min() method. Perform the filtering by referencing the column directly, not passing a SQL string.
* Find the length of the longest (in terms of time) flight that left SEA by filter()ing and using the .max() method. Perform the filtering by referencing the column directly, not passing a SQL string.

# Find the shortest flight from PDX in terms of distance.

flights.filter(flights.origin == "PDX").groupBy().min("distance").show()

# Find the longest flight from SEA in terms of air time

flights.filter(flights.origin == "SEA").groupBy().max("air\_time").show()

* Use the .avg() method to get the average air time of Delta Airlines flights (where the carrier column has the value "DL") that left SEA. The place of departure is stored in the column origin. show() the result.
* Use the .sum() method to get the total number of hours all planes in this dataset spent in the air by creating a column called duration\_hrs from the column air\_time. show() the result.

# Average duration of Delta flights.

flights.filter(flights.carrier=="DL").filter(flights.origin == "SEA").groupBy().avg("air\_time").show()

# Total hours in the air

flights.withColumn("duration\_hrs", flights.air\_time/60).groupBy().sum("duration\_hrs").show()

* Create a DataFrame called by\_plane that is grouped by the column tailnum.
* Use the .count() method with no arguments to count the number of flights each plane made.
* Create a DataFrame called by\_origin that is grouped by the column origin.
* Find the .avg() of the air\_time column to find average duration of flights from PDX and SEA.

# Import pyspark.sql.functions as F

import pyspark.sql.functions as F

# Group by month and dest

by\_month\_dest = flights.groupBy("month", "dest")

# Average departure delay by month and destination

by\_month\_dest.avg("dep\_delay").show()

# Standard deviation of departure delay

by\_month\_dest.agg(F.stddev("dep\_delay")).show()

* Examine the airports DataFrame by calling .show(). Note which key column will let you join airports to the flights table.
* Rename the faa column in airports to dest by re-assigning the result of airports.withColumnRenamed("faa", "dest") to airports.
* Join the flights with the airports DataFrame on the dest column by calling the .join() method on flights. Save the result as flights\_with\_airports.
  + The first argument should be the other DataFrame, airports.
  + The argument on should be the key column.
  + The argument how should be "leftouter".
* Call .show() on flights\_with\_airports to examine the data again. Note the new information that has been added.

# Examine the data

print(airports.show())

# Rename the faa column

airports = airports.withColumnRenamed("faa", "dest")

# Join the DataFrames

flights\_with\_airports = flights.join(airports, on="dest", how="leftouter")

# Examine the new DataFrame

print(flights\_with\_airports.show())

* First, rename the year column of planes to plane\_year to avoid duplicate column names.
* Create a new DataFrame called model\_data by joining the flights table with planes using the tailnum column as the key.

# Rename year column

planes = planes.withColumnRenamed("year", "plane\_year")

# Join the DataFrames

model\_data = flights.join(planes, on="tailnum", how="leftouter")

Use the method .withColumn() to .cast() the following columns to type "integer". Access the columns using the df.col notation:

* model\_data.arr\_delay
* model\_data.air\_time
* model\_data.month
* model\_data.plane\_year

# Cast the columns to integers

model\_data = model\_data.withColumn("arr\_delay", model\_data.arr\_delay.cast("integer"))

model\_data = model\_data.withColumn("air\_time", model\_data.air\_time.cast("integer"))

model\_data = model\_data.withColumn("month", model\_data.month.cast("integer"))

model\_data = model\_data.withColumn("plane\_year", model\_data.plane\_year.cast("integer"))

Create the column plane\_age using the .withColumn() method and subtracting the year of manufacture (column plane\_year) from the year (column year) of the flight.

# Create the column plane\_age

model\_data = model\_data.withColumn("plane\_age", model\_data.year - model\_data.plane\_year)

* Use the .withColumn() method to create the column is\_late. This column is equal to model\_data.arr\_delay > 0.
* Convert this column to an integer column so that you can use it in your model and name it label (this is the default name for the response variable in Spark's machine learning routines).
* Filter out missing values (this has been done for you).

# Create is\_late

model\_data = model\_data.withColumn("is\_late", model\_data.arr\_delay>0)

# Convert to an integer

model\_data = model\_data.withColumn("label", model\_data.is\_late.cast("integer"))

# Remove missing values

model\_data = model\_data.filter("arr\_delay is not NULL and dep\_delay is not NULL and air\_time is not NULL and plane\_year is not NULL")

* Create a StringIndexer called carr\_indexer by calling StringIndexer() with inputCol="carrier" and outputCol="carrier\_index".
* Create a OneHotEncoder called carr\_encoder by calling OneHotEncoder() with inputCol="carrier\_index" and outputCol="carrier\_fact"

# Create a StringIndexer

carr\_indexer = StringIndexer(inputCol="carrier", outputCol="carrier\_index")

# Create a OneHotEncoder

carr\_encoder = OneHotEncoder(inputCol="carrier\_index", outputCol="carrier\_fact")

* Create a StringIndexer called dest\_indexer by calling StringIndexer() with inputCol="dest" and outputCol="dest\_index".
* Create a OneHotEncoder called dest\_encoder by calling OneHotEncoder() with inputCol="dest\_index" and outputCol="dest\_fact"

# Create a StringIndexer

dest\_indexer = StringIndexer(inputCol="dest", outputCol="dest\_index")

# Create a OneHotEncoder

dest\_encoder = OneHotEncoder(inputCol="dest\_index", outputCol="dest\_fact")

Create a VectorAssembler by calling VectorAssembler() with the inputCols names as a list and the outputCol name "features".

* The list of columns should be ["month", "air\_time", "carrier\_fact", "dest\_fact", "plane\_age"]

# Make a VectorAssembler

vec\_assembler = VectorAssembler(inputCols=["month", "air\_time", "carrier\_fact", "dest\_fact", "plane\_age"], outputCol="features")

* Import Pipeline from pyspark.ml.
* Call the Pipeline() constructor with the keyword argument stages to create a Pipeline called flights\_pipe.
  + stages should be a list holding all the stages you want your data to go through in the pipeline. Here this is just: [dest\_indexer, dest\_encoder, carr\_indexer, carr\_encoder, vec\_assembler]

# Import Pipeline

from pyspark.ml import Pipeline

# Make the pipeline

flights\_pipe = Pipeline(stages=[dest\_indexer, dest\_encoder, carr\_indexer, carr\_encoder, vec\_assembler])

Create the DataFrame piped\_data by calling the Pipeline methods .fit() and .transform() in a chain. Both of these methods take model\_data as their only argument.

# Fit and transform the data

piped\_data = flights\_pipe.fit(model\_data).transform(model\_data)

Use the DataFrame method .randomSplit() to split piped\_data into two pieces, training with 60% of the data, and test with 40% of the data by passing the list [.6, .4] to the .randomSplit() method.

# Split the data into training and test sets

training, test = piped\_data.randomSplit([.6, .4])

* Import the LogisticRegression class from pyspark.ml.classification.
* Create a LogisticRegression called lr by calling LogisticRegression() with no arguments.

# Import LogisticRegression

from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression Estimator

lr = LogisticRegression()

* Import the submodule pyspark.ml.evaluation as evals.
* Create evaluator by calling evals.BinaryClassificationEvaluator() with the argument metricName="areaUnderROC"

# Import the evaluation submodule

import pyspark.ml.evaluation as evals

# Create a BinaryClassificationEvaluator

evaluator = evals.BinaryClassificationEvaluator(metricName="areaUnderROC")

* Import the submodule pyspark.ml.tuning under the alias tune.
* Call the class constructor ParamGridBuilder() with no arguments. Save this as grid.
* Call the .addGrid() method on grid with lr.regParam as the first argument and np.arange(0, .1, .01) as the second argument. This second call is a function from the numpy module (imported as np) that creates a list of numbers from 0 to .1, incrementing by .01. Overwrite grid with the result.
* Update grid again by calling the .addGrid() method a second time create a grid for lr.elasticNetParam that includes only the values [0, 1].
* Call the .build() method on grid and overwrite it with the output.

# Import the tuning submodule

import pyspark.ml.tuning as tune

# Create the parameter grid

grid = tune.ParamGridBuilder()

# Add the hyperparameter

grid = grid.addGrid(lr.regParam, np.arange(0, .1, .01))

grid = grid.addGrid(lr.elasticNetParam, [0, 1])

# Build the grid

grid = grid.build()

* Create a CrossValidator by calling tune.CrossValidator() with the arguments:
  + estimator=lr
  + estimatorParamMaps=grid
  + evaluator=evaluator
* Name this object cv

# Create the CrossValidator

cv = tune.CrossValidator(estimator=lr,

estimatorParamMaps=grid,

evaluator=evaluator

)

* Create best\_lr by calling lr.fit() on the training data.
* Print best\_lr to verify that it's an object of the LogisticRegressionModel class.

# Call lr.fit()

best\_lr = lr.fit(training)

# Print best\_lr

print(best\_lr)

* Use your model to generate predictions by applying best\_lr.transform() to the test data. Save this as test\_results.
* Call evaluator.evaluate() on test\_results to compute the AUC. Print the output.

# Use the model to predict the test set

test\_results = best\_lr.transform(test)

# Evaluate the predictions

print(evaluator.evaluate(test\_results))

Pyspark

* Print the version of SparkContext in the PySpark shell.
* Print the Python version of SparkContext in the PySpark shell.
* What is the master of SparkContext in the PySpark shell?

# 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)

* Create a python list named numb containing the numbers 1 to 100.
* Load the list into Spark using Spark Context's parallelize method and assign it to a variable spark\_data

# Create a python list of numbers from 1 to 100

numb = range(1, 100)

# Load the list into PySpark

spark\_data = sc.parallelize(numb)

* Load a local text file README.md in PySpark shell.

# Load a local file into PySpark shell

lines = sc.textFile(file\_path)

* Print my\_list which is available in your environment.
* Square each item in my\_list using map() and lambda().
* Print the result of map function.

# Print my\_list in the console

print("Input list is", my\_list)

# Square all numbers in my\_list

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)

* Print my\_list2 which is available in your environment.
* Filter the numbers divisible by 10 from my\_list2 using filter() and lambda().
* Print the numbers divisible by 10 from my\_list2

# 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)

* Create an RDD named RDD from a list of words.
* Confirm the object created is RDD.

# Create an RDD from a list of words

RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", type(RDD))

* Print the file\_path in the PySpark shell.
* Create an RDD named fileRDD from a file\_path with the file name README.md.
* Print the type of the fileRDD created.

# 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))

* Find the number of partitions that support fileRDD RDD.
* Create an RDD named fileRDD\_part from the file path but create 5 partitions.
* Confirm the number of partitions in the new fileRDD\_part RDD.

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(file\_path, minPartitions = 5)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.getNumPartitions())

* Create map() transformation that cubes all of the numbers in numbRDD.
* Collect the results in a numbers\_all variable.
* Print the output from numbers\_all variable.

# 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)

* Create filter() transformation to select the lines containing the keyword Spark.
* How many lines in fileRDD\_filter contains the keyword Spark?
* Print the first four lines of the resulting RDD.

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

print(line)

* Create a pair RDD named Rdd with tuples (1,2),(3,4),(3,6),(4,5).
* Transform the Rdd with reduceByKey() into a pair RDD Rdd\_Reduced by adding the values with the same key.
* Collect the contents of pair RDD Rdd\_Reduced and iterate to print the output.

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([(1,2), (3,4), (3,6), (4,5)])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: x + y)

# Iterate over the result and print the output

for num in Rdd\_Reduced.collect():

print("Key {} has {} Counts".format(num[0], num[1]))

* Sort the Rdd\_Reduced RDD using the key in descending order.
* Collect the contents and iterate to print the output.

# 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("Key {} has {} Counts".format(num[0], num[1]))

* Use the countByKey() action on the Rdd to count the unique keys and assign the result to a variable total.
* What is the type of total?
* Iterate over the total and print the keys and their counts.

# 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")

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 and print the 10 most frequent words and their frequencies.
* Create an RDD called baseRDD that reads lines from file\_path.
* Transform the baseRDD into a long list of words and create a new splitRDD.
* Count the total words in splitRDD.

# 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())

# Count the total number of words

print("Total number of words in splitRDD:", splitRDD.count())

* Convert the words in splitRDD in lower case and then remove stop words from stop\_words.
* Create a pair RDD tuple containing the word and the number 1 from each word element in splitRDD.
* Get the count of the number of occurrences of each word (word frequency) in the pair RDD using reduceByKey()

# Convert the words in lower case and remove stop words from 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)

* Print the first 10 words and their frequencies from the resultRDD.
* Swap the keys and values in the resultRDD.
* Sort the keys according to descending order.
* Print the top 10 most frequent words and their frequencies.

# 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]))

# Sort the keys in descending order

resultRDD\_swap\_sort = resultRDD\_swap.sortByKey(ascending=False)

# Show the top 10 most frequent words and their frequencies

for word in resultRDD\_swap\_sort.take(10):

print("{} has {} counts". format(word[1], word[0]))

* Create a sample\_list from tuples - ('Mona',20), ('Jennifer',34), ('John',20), ('Jim',26).
* Create an RDD from the sample\_list.
* Create a PySpark DataFrame using the above RDD and schema.
* Confirm the output as PySpark DataFrame.

# Create a list of tuples

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 names\_df

print("The type of names\_df is", type(names\_df))

* Create a DataFrame from file\_path variable which is the path to the people.csv file.
* Confirm the output as PySpark DataFrame.

# Create an DataFrame from file\_path

people\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Check the type of people\_df

print("The type of people\_df is", type(people\_df))

* Print the first 10 observations in the people\_df DataFrame.
* Count the number of rows in the people\_df DataFrame.
* How many columns does people\_df DataFrame have and what are their names?

# Print the first 10 observations

people\_df.show(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))

* Select 'name', 'sex' and 'date of birth' columns from people\_df and create people\_df\_sub DataFrame.
* Print the first 10 observations in the people\_df DataFrame.
* Remove duplicate entries from people\_df\_sub DataFrame and create people\_df\_sub\_nodup DataFrame.
* How many rows are there before and after duplicates are removed?

# 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()))

* Filter the people\_df DataFrame to select all rows where sex is female into people\_df\_female DataFrame.
* Filter the people\_df DataFrame to select all rows where sex is male into people\_df\_male DataFrame.
* Count the number of rows in people\_df\_female and people\_df\_male DataFrames.

# Filter people\_df to select females

people\_df\_female = people\_df.filter(people\_df.sex == "female")

# Filter people\_df to select males

people\_df\_male = people\_df.filter(people\_df.sex == "male")

# Count the number of rows

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()))

* Create a temporary table people that's a pointer to the people\_df DataFrame.
* Construct a query to select the names of the people from the temporary table people.
* Assign the result of Spark's query to a new DataFrame - people\_df\_names.
* Print the top 10 names of the people from people\_df\_names DataFrame.

# Create a temporary table "people"

people\_df.createOrReplaceTempView("people")

# Construct a query to select the names of the people from the temporary table "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)

* Filter the people table to select all rows where sex is female into people\_female\_df DataFrame.
* Filter the people table to select all rows where sex is male into people\_male\_df DataFrame.
* Count the number of rows in both people\_female and people\_male DataFrames.

# 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 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()))

* Print the names of the columns in names\_df DataFrame.
* Convert names\_df DataFrame to df\_pandas Pandas DataFrame.
* Use matplotlib's plot() method to create a horizontal bar plot with 'Name' on x-axis and 'Age' on y-axis.

# Check the column names of names\_df

print("The column names of names\_df are", names\_df.columns)

# Convert to Pandas DataFrame

df\_pandas = names\_df.toPandas()

# Create a horizontal bar plot

df\_pandas.plot(kind='barh', x='Name', y='Age', colormap='winter\_r')

plt.show()

* Create a PySpark DataFrame from file\_path which is the path to the Fifa2018\_dataset.csv file.
* Print the schema of the DataFrame.
* Print the first 10 observations.
* How many rows are in there in the DataFrame?

# 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()))

* Create temporary table fifa\_df from fifa\_df\_table DataFrame.
* Construct a "query" to extract the "Age" column from Germany players.
* Apply the SQL "query" to the temporary view table and create a new DataFrame.
* Computes basic statistics of the created DataFrame.

# 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 statistics

fifa\_df\_germany\_age.describe().show()

* Convert fifa\_df\_germany\_age to fifa\_df\_germany\_age\_pandas Pandas DataFrame.
* Generate a density plot of the 'Age' column from the fifa\_df\_germany\_age\_pandas Pandas DataFrame.

# 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()

* Import pyspark.mllib recommendation submodule and Alternating Least Squares class.
* Import pyspark.mllib classification submodule and Logistic Regression with LBFGS class.
* Import pyspark.mllib clustering submodule and kmeans class.

# 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

* Load the ratings.csv dataset into an RDD.
* Split the RDD using , as a delimiter.
* For each line of the RDD, using Rating() class create a tuple of userID, productID, rating.
* Randomly split the data into training data and test data (0.8 and 0.2).

# 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])

* Train ALS algorithm with training data and configured parameters (rank = 10 and iterations = 10).
* Drop the rating column in the test data.
* Test the model by predicting the rating from the test data.
* Print the first two rows of the predicted ratings.

# Create the ALS model on the training data

model = ALS.train(training\_data, rank=10, iterations=10)

# Drop the ratings column

testdata\_no\_rating = test\_data.map(lambda p: (p[0], p[1]))

# Predict the model

predictions = model.predictAll(testdata\_no\_rating)

# Print the first rows of the RDD

predictions.take(2)

* Organize ratings RDD to make ((user, product), rating).
* Organize predictions RDD to make ((user, product), rating).
* Join the prediction RDD with the ratings RDD.
* Evaluate the model using MSE between original rating and predicted rating and print it.

# 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 = {:.2f}".format(MSE))

* Create two RDDS, one for 'spam' and one for 'non-spam (ham)'.
* Split each email in 'spam' and 'non-spam' RDDs into words.
* Print the first element in the split RDD of both 'spam' and 'non-spam'.

# 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.flatMap(lambda email: email.split(' '))

non\_spam\_words = non\_spam\_rdd.flatMap(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())

* Create a HashingTF() instance to map email text to vectors of 200 features.
* Each message in 'spam' and 'non-spam' datasets are split into words, and each word is mapped to one feature.
* Label the features: 1 for spam, 0 for non-spam.
* Combine both the spam and non-spam samples into a single dataset.

# 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.join(non\_spam\_samples)

* Split the combined data into training and test sets (80/20).
* Train the Logistic Regression (LBFGS variant) model with the training dataset.
* Create a prediction label from the trained model on the test dataset.
* Combine the labels in the test dataset with the labels in the prediction dataset.
* Calculate the accuracy of the trained model using original and predicted labels on the labels\_and\_preds.

# 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)

# Create a prediction label from the test data

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))

* Split the combined data into training and test sets (80/20).
* Train the Logistic Regression (LBFGS variant) model with the training dataset.
* Create a prediction label from the trained model on the test dataset.
* Combine the labels in the test dataset with the labels in the prediction dataset.
* Calculate the accuracy of the trained model using original and predicted labels on the labels\_and\_preds.

# 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)

# Create a prediction label from the test data

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))

* Load the 5000\_points dataset into a RDD named clusterRDD.
* Transform the clusterRDD by splitting the lines based on the tab ("\t").
* Transform the split RDD to create a list of integers for the two columns.
* Confirm that there are 5000 rows in the dataset.

# Load the dataset into a RDD

clusterRDD = sc.textFile(file\_path)

# Split the RDD based on tab

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 the rdd\_split\_int dataset".format(rdd\_split\_int.count()))

* Train the KMeans model with clusters from 13 to 16 and print the WSSSE for each cluster.
* Train the KMeans model again with the best cluster (k=15).
* Get the Cluster Centers (centroids) of KMeans model trained with k=15.

# Train the model with clusters from 13 to 16 and compute WSSSE

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

* Convert rdd\_split\_int RDD into a Spark DataFrame.
* Convert Spark DataFrame into a Pandas DataFrame.
* Create a Pandas DataFrame from cluster\_centers list.
* Create a scatter plot of the raw data and an overlaid scatter plot with centroids for k = 15.

# 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" that you generated earlier into Pandas DataFrame

cluster\_centers\_pandas = pd.DataFrame(cluster\_centers, columns=["col1", "col2"])

# Create an overlaid 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()

* Load csv data from the file trainsched.txt into a dataframe stored in a variable named df.
* Create a temporary table from df. Call the table "table1".

# Load trainsched.txt

df = spark.read.csv("trainsched.txt", header=True)

# Create temporary table called table1

df.createOrReplaceTempView('table1')

Use a DESCRIBE query to determine the names and types of the columns in the table schedule

# Inspect the columns in the table df

spark.sql("DESCRIBE schedule").show()

* Run a query that adds an additional column to the records in this dataset called running\_total. The column running\_total SUM()s the difference between station time given by the diff\_min column.
* Run the query and display the result.

# Add col running\_total that sums diff\_min col in each group

query = """

SELECT train\_id, station, time, diff\_min,

SUM(diff\_min) OVER (PARTITION BY train\_id ORDER BY time) AS running\_total

FROM schedule

"""

# Run the query and display the result

spark.sql(query).show()

* Provide the row number of the erroneous row as an integer.
* Provide the clause (as a string) that when added to the OVER clause fixes the problem.

query = """

SELECT

ROW\_NUMBER() OVER (ORDER BY time) AS row,

train\_id,

station,

time,

LEAD(time,1) OVER (ORDER BY time) AS time\_next

FROM schedule

"""

spark.sql(query).show()

# Give the number of the bad row as an integer

bad\_row = 7

# Provide the missing clause, SQL keywords in upper case

clause = 'PARTITION by train\_id'

* Fill in the blanks to get the first pair of commands to display the identical result.
* The fourth result, named result, is a naive attempt at replicating the previous line. However, it is counter-intuitively different. How? Fill in the blank to print the name of the second column of result.

# Give the identical result in each command

spark.sql('SELECT train\_id, MIN(time) AS start FROM schedule GROUP BY train\_id').show()

df.groupBy('train\_id').agg({'time':'min'}).withColumnRenamed('min(time)', 'start').show()

# Print the second column of the result

spark.sql('SELECT train\_id, MIN(time), MAX(time) FROM schedule GROUP BY train\_id').show()

result = df.groupBy('train\_id').agg({'time':'min', 'time':'max'})

result.show()

print(result.columns[0])

* Write a SQL query that gives an identical result to the dot notation query.
* from pyspark.sql.functions import min, max, col
* expr = [min(col("time")).alias('start'), max(col("time")).alias('end')]
* dot\_df = df.groupBy("train\_id").agg(\*expr)
* dot\_df.show()

# Write a SQL query giving a result identical to dot\_df

query = "SELECT train\_id, MIN(time) AS start, MAX(time) AS end FROM schedule GROUP BY train\_id"

sql\_df = spark.sql(query)

sql\_df.show()

Create a dataframe called dot\_df that contains the identical result as df, using dot notation instead of SQL.

* The LEAD clause has an equivalent function in pyspark.sql.functions.
* The PARTITION BY, and ORDER BY clauses each have an equivalent dot notation function that is called on the Window object.
* The following imports are available:
  + from pyspark.sql import Window
  + from pyspark.sql.functions import lead
* df = spark.sql("""
* SELECT \*,
* LEAD(time,1) OVER(PARTITION BY train\_id ORDER BY time) AS time\_next
* FROM schedule
* """)

# Obtain the identical result using dot notation

dot\_df = df.withColumn('time\_next', lead('time', 1)

.over(Window.partitionBy('train\_id')

.orderBy('time')))

* Create a SQL query to obtain an identical result to dot\_df. Please format the query according to the scaffolding.
* window = Window.partitionBy('train\_id').orderBy('time')
* dot\_df = df.withColumn('diff\_min',
* (unix\_timestamp(lead('time', 1).over(window),'H:m')
* - unix\_timestamp('time', 'H:m'))/60)

# Create a SQL query to obtain an identical result to dot\_df

query = """

SELECT \*,

(UNIX\_TIMESTAMP(LEAD(time, 1) OVER (PARTITION BY train\_id ORDER BY time),'H:m')

- UNIX\_TIMESTAMP(time, 'H:m'))/60 AS diff\_min

FROM schedule

"""

sql\_df = spark.sql(query)

sql\_df.show()

* Load sherlock\_sentences.parquet.
* Filter on "id > 70", and show the first 5 rows.

# Load the dataframe

df = spark.read.load('sherlock\_sentences.parquet')

# Filter and show the first 5 rows

df.where('id > 70').show(5, truncate=False)

* Split the clause column into a column called words, containing an array of individual words.
* Explode the words column into a column called word.
* Count the resulting number of rows.

# Split the clause column into a column called words

split\_df = clauses\_df.select(split('clause', ' ').alias('words'))

split\_df.show(5, truncate=False)

# Explode the words column into a column called word

exploded\_df = split\_df.select(explode('words').alias('word'))

exploded\_df.show(10)

# Count the resulting number of rows in exploded\_df

print("\nNumber of rows: ", exploded\_df.count())

Get the word for each row, along with the previous two words and the subsequent two words.# Word for each row, previous two and subsequent two words

query = """

SELECT

part,

LAG(word, 2) OVER(PARTITION BY part ORDER BY id) AS w1,

LAG(word, 1) OVER(PARTITION BY part ORDER BY id) AS w2,

word AS w3,

LEAD(word, 1) OVER(PARTITION BY part ORDER BY id) AS w4,

LEAD(word, 2) OVER(PARTITION BY part ORDER BY id) AS w5

FROM text

"""

spark.sql(query).where("part = 12").show(10)

* Repartition the text\_df into 12 partitions, with each chapter in its own partition.
* Display the number of partitions in the new dataframe.

# Repartition text\_df into 12 partitions on 'chapter' column

repart\_df = text\_df.repartition(12, 'chapter')

# Prove that repart\_df has 12 partitions

repart\_df.rdd.getNumPartitions()

Create a query query that finds the **10** most common 5-tuples in the dataset.

# Find the top 10 sequences of five words

query = """

SELECT w1, w2, w3, w4, w5, COUNT(\*) AS count FROM (

SELECT word AS w1,

LEAD(word, 1) OVER(PARTITION BY part ORDER BY id ) AS w2,

LEAD(word, 2) OVER(PARTITION BY part ORDER BY id ) AS w3,

LEAD(word, 3) OVER(PARTITION BY part ORDER BY id) AS w4,

LEAD(word, 4) OVER(PARTITION BY part ORDER BY id) AS w5

FROM text

)

GROUP BY w1, w2, w3, w4, w5

ORDER BY count DESC

LIMIT 10 """

df = spark.sql(query)

df.show()

Retrieve the last ten unique 5-tuples sorted alphabetically in descending order.

# Unique 5-tuples sorted in descending order

query = """

SELECT DISTINCT w1, w2, w3, w4, w5 FROM (

SELECT word AS w1,

LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,

LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3,

LEAD(word,3) OVER(PARTITION BY part ORDER BY id ) AS w4,

LEAD(word,4) OVER(PARTITION BY part ORDER BY id ) AS w5

FROM text

)

ORDER BY w1 DESC, w2 DESC, w3 DESC, w4 DESC, w5 DESC

LIMIT 10

"""

df = spark.sql(query)

df.show()

* Get the most frequent 3-tuple per chapter.

# Most frequent 3-tuple per chapter

query = """

SELECT chapter, w1, w2, w3, count FROM

(

SELECT

chapter,

ROW\_NUMBER() OVER (PARTITION BY chapter ORDER BY count DESC) AS row,

w1, w2, w3, count

FROM ( %s )

)

WHERE row = 1

ORDER BY chapter ASC

""" % subquery

spark.sql(query).show()

* Cache df1 only.
* Run a first action on df1 and repeat it, then run an action df2 and repeat it. This has been done for you.
* Confirm whether or not df1 is cached.

# Unpersists df1 and df2 and initializes a timer

prep(df1, df2)

# Cache df1

df1.cache()

# Run actions on both dataframes

run(df1, "df1\_1st")

run(df1, "df1\_2nd")

run(df2, "df2\_1st")

run(df2, "df2\_2nd", elapsed=True)

# Prove df1 is cached

print(df1.is\_cached)

* Cache df2, but not df1.
* Run a first action on df1 and repeat it, then run an action df2 and repeat it. This has been done for you.

# Unpersist df1 and df2 and initializes a timer

prep(df1, df2)

# Persist df2 using memory and disk storage level

df2.persist(storageLevel=pyspark.StorageLevel.MEMORY\_AND\_DISK)

# Run actions both dataframes

run(df1, "df1\_1st")

run(df1, "df1\_2nd")

run(df2, "df2\_1st")

run(df2, "df2\_2nd", elapsed=True)

* List the tables with the listTables() method.
* Cache table1 and confirm that it is cached.
* Uncache table1 and confirm that it is uncached.

# List the tables

print("Tables:\n", spark.catalog.listTables())

# Cache table1 and Confirm that it is cached

spark.catalog.cacheTable('table1')

print("table1 is cached: ", spark.catalog.isCached('table1'))

# Uncache table1 and confirm that it is uncached

spark.catalog.uncacheTable('table1')

print("table1 is cached: ", spark.catalog.isCached('table1'))

* Log columns of text\_df as debug message.
* Log whether table1 is cached as info message.
* Log first row of text\_df as warning message.
* Log selected columns of text\_df as error message.

# Log columns of text\_df as debug message

logging.debug("text\_df columns: %s", text\_df.columns)

# Log whether table1 is cached as info message

logging.info("table1 is cached: %s", spark.catalog.isCached(tableName="table1"))

# Log first row of text\_df as warning message

logging.warning("The first row of text\_df:\n %s", text\_df.first())

# Log selected columns of text\_df as error message

logging.error("Selected columns: %s", text\_df.select("id", "word"))

Several log statements are provided. All of them are initially commented out. Uncomment the five statements that do not trigger an action on text\_df

# Uncomment the 5 statements that do NOT trigger text\_df

logging.debug("text\_df columns: %s", text\_df.columns)

logging.info("table1 is cached: %s", spark.catalog.isCached(tableName="table1"))

#logging.warning("The first row of text\_df: %s", text\_df.first())

logging.error("Selected columns: %s", text\_df.select("id", "word"))

logging.info("Tables: %s", spark.sql("show tables").collect())

logging.debug("First row: %s", spark.sql("SELECT \* FROM table1 limit 1"))

#logging.debug("Count: %s", spark.sql("SELECT COUNT(\*) AS count FROM table1").collect())

* Run explain on text\_df.
* Run explain on a SQL query that does a "SELECT COUNT(\*) as count" on table1.
* Run explain on a SQL query that counts the number of unique words in table1.

# Run explain on text\_df

text\_df.explain()

# Run explain on "SELECT COUNT(\*) AS count FROM table1"

spark.sql("SELECT COUNT(\*) AS count FROM table1").explain()

# Run explain on "SELECT COUNT(DISTINCT word) AS words FROM table1"

spark.sql("SELECT COUNT(DISTINCT word) AS words FROM table1").explain()

* Create a udf that returns true if and only if the value is a nonempty vector, using numNonzeros()
* Create a udf that returns the first element of the array and returns its string representation.

# Returns true if the value is a nonempty vector

nonempty\_udf = udf(lambda x:

True if (x and hasattr(x, "toArray") and x.numNonzeros())

else False, BooleanType())

# Returns first element of the array as string

s\_udf = udf(lambda x: str(x[0]) if (x and type(x) is list and len(x) > 0)

else '', StringType())

* Show the rows of df\_before where doc contains the item 5.
* Create a udf that removes items in TRIVIAL\_TOKENS from an array column. The order does not need to be preserved.
* Remove tokens from the in and out columns in df2 that appear in TRIVIAL\_TOKENS.

# Show the rows where doc contains the item '5'

df\_before.where(array\_contains('doc', '5')).show()

# UDF removes items in TRIVIAL\_TOKENS from array

rm\_trivial\_udf = udf(lambda x:

list(set(x) - TRIVIAL\_TOKENS) if x

else x,

ArrayType(StringType()))

# Remove trivial tokens from 'in' and 'out' columns of df2

df\_after = df\_before.withColumn('in', rm\_trivial\_udf('in'))\

.withColumn('out', rm\_trivial\_udf('out'))

# Show the rows of df\_after where doc contains the item '5'

df\_after.where(array\_contains('doc','5')).show()

* Create a UDF called first\_udf. It selects the first element of a vector column. Set the result to a default value of 0.0 for any item that is not a vector containing at least one item and cast the output as a float.
* Use the select operation on df to apply first\_udf to the output column.

# Selects the first element of a vector column

first\_udf = udf(lambda x:

float(x.indices[0])

if (x and hasattr(x, "toArray") and x.numNonzeros())

else 0.0,

FloatType())

# Apply first\_udf to the output column

df.select(first\_udf("output").alias("result")).show(5)

* Create a new dataframe called df\_new by adding a new column to df. Call the new column label .
* Show the first five rows of df\_new.

# Add label by applying the get\_first\_udf to output column

df\_new = df.withColumn('label', get\_first\_udf('output'))

# Show the first five rows

df\_new.show(5)

+----------------------+-------+------------------------------------+

|in |out |invec |

+----------------------+-------+------------------------------------+

|[then, how, many, are]|[there]|(126,[3,18,28,30],[1.0,1.0,1.0,1.0])|

|[how] |[many] |(126,[28],[1.0]) |

|[i, donot] |[know] |(126,[15,78],[1.0,1.0]) |

+----------------------+-------+------------------------------------+

only showing top 3 rows

+------------------------------------+----------------+

|invec |outvec |

+------------------------------------+----------------+

|(126,[3,18,28,30],[1.0,1.0,1.0,1.0])|(126,[11],[1.0])|

|(126,[28],[1.0]) |(126,[18],[1.0])|

|(126,[15,78],[1.0,1.0]) |(126,[21],[1.0])|

+------------------------------------+----------------+

only showing top 3 rows

* Create a dataframe called result by using model to transform() df. result has the columns sentence, in, out, and invec. invec is the vector transformation of the in column.
* Add a column to result called outvec. result now has the columns sentence, in, out, invec, and outvec.

# Transform df using model

result = model.transform(df.withColumnRenamed('in', 'words'))\

.withColumnRenamed('words', 'in')\

.withColumnRenamed('vec', 'invec')

result.drop('sentence').show(3, False)

# Add a column based on the out column called outvec

result = model.transform(result.withColumnRenamed('out', 'words'))\

.withColumnRenamed('words', 'out')\

.withColumnRenamed('vec', 'invec')

result.select('invec', 'outvec').show(3, False)

* Import the lit function.
* Select the rows where endword is 'him' and add a integer column label having the value 1.
* Select the rows where endword is not 'him' and add a integer column label having the value 0.
* Union these two sets, using a number of negative examples equal to the number of positive examples.

# Import the lit function

from pyspark.sql import lit

# Select the rows where endword is 'him' and label 1

df\_pos = df.where("endword = 'him'")\

.withColumn('label', lit(1))

# Select the rows where endword is not 'him' and label 0

df\_neg = df.where("endword <> 'him'")\

.withColumn('label', lit(0))

# Union pos and neg in equal number

df\_examples = df\_pos.union(df\_neg.limit(df\_pos.count()))

print("Number of examples: ", df\_examples.count())

df\_examples.where("endword <> 'him'").sample(False, .1, 42).show(5)

* Split the examples into train and test using a 80/20 split.
* Print the number of training examples.
* Print the number of test examples.

# Split the examples into train and test, use 80/20 split

df\_trainset, df\_testset = df\_examples.randomSplit((0.8, 0.2), 42)

# Print the number of training examples

print("Number training: ", df\_trainset.count())

# Print the number of test examples

print("Number test: ", df\_testset.count())

* Import the Logistic Regression Classifier.
* Instantiate the classifier. Set maximum iterations to 100, the regularization parameter to 0.4, and the elastic net parameter to 0.0.
* Train the classifier on the trainset.
* Print the number of training iterations.

# Import the logistic regression classifier

from pyspark.ml.classification import LogisticRegression

# Instantiate logistic setting elasticnet to 0.0

logistic = LogisticRegression(maxIter=100, regParam=0.4, elasticNetParam=0.0)

# Train the logistic classifer on the trainset

df\_fitted = logistic.fit(df\_trainset)

# Print the number of training iterations

print("Training iterations: ", df\_fitted.summary.totalIterations)

* Score the trained model on the test data.
* Print the Area Under Curve metric.

# Score the model on test data

testSummary = df\_fitted.evaluate(df\_testset)

# Print the AUC metric

print("\ntest AUC: %.3f" % testSummary.areaUnderROC)

* Apply the model to the data in df\_testset.
* Print "incorrect" if prediction does not match label.

# Apply the model to the test data

predictions = df\_fitted.transform(df\_testset).select(fields)

# Print incorrect if prediction does not match label

for x in predictions.take(8):

print()

if x.label != int(x.prediction):

print("INCORRECT ==> ")

for y in fields:

print(y,":", x[y])