Getting to know PySpark

```
!pip install pyspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-eu.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-hadoop2.7.tgz
!tar xf spark-3.0.1-bin-hadoop2.7.tgz
!pip install -q findspark

import pyspark as sp


import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.0.1-bin-hadoop2.7"


! echo $SPARK_HOME


import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
#sc = spark.SparkContext.getOrCreate()


# Verify SparkContext
print(spark)

# Print Spark version
print(spark.version)
```

Creating a SparkSession

```
# Import SparkSession from pyspark.sql
from pyspark.sql import SparkSession

# Create my_spark
spark = SparkSession.builder.getOrCreate()

# Print my_spark
print(spark)
```

Viewing tables

```
from google.colab import files
files.upload()
```

```
!ls
```

```
flights = spark.read.csv('flights_small.csv', inferSchema=True, header =True)
print(flights)
```

```
flights.printSchema()
```

```
spark.catalog.listTables()
```

Are you query-ious?

```
flights.createOrReplaceTempView("flights");
```

```
# 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()
```

Pandafy a Spark DataFrame

```
# 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())
```

Put some Spark in your data

```
import numpy as np
import pandas as pd
```

```
# 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.name = spark_temp.createOrReplaceTempView('temp')

# Examine the tables in the catalog again
print(spark.catalog.listTables())


from google.colab import files
files.upload()
```

Dropping the middle man

```
# Don't change this file path
file_path = "airports.csv"

# Read in the airports data
airports = spark.read.csv(file_path, header=True)

# Show the data
airports.show()


type(airports)


spark.catalog.listDatabases()


spark.catalog.listTables()
```

Manipulating data

Creating columns

```
flights = spark.read.csv('flights_small.csv', header=True)


flights.show()


flights.name = flights.createOrReplaceTempView('flights')
```

```
spark.catalog.listTables()


# Create the DataFrame flights
flights_1 = spark.table('flights')

# Show the head
print(flights_1.show())


# Add duration_hrs
flights = flights.withColumn('duration_hrs', flights.air_time / 60)


flights.show()
```

Filtering Data

```
# Filter flights with a SQL string
long_flights1 = flights.filter('distance > 1000')

# Filter flights with a boolean column
long_flights2 = flights.filter(flights.distance > 1000)

# Examine the data to check they're equal
print(long_flights1.show())
print(long_flights2.show())
```

Selecting

```
# 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)
```

Selecting II

```
# 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_sp
```

Aggregating

```
flights.describe()
```

```
flights = flights.withColumn("distance", flights.distance.cast("float"))
```

```
flights = flights.withColumn("air_time", flights.air_time.cast("float"))
```

```
flights.describe('air_time', 'distance').show()
```

```
# 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 duration
flights.filter(flights.origin == "SEA").groupBy().max("air_time").show()
```

Aggregating II

```
# 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
```

Grouping and Aggregating I

```
flights.show()
```

```
# Group by tailnum
by_plane = flights.groupBy("tailnum")

# N
```

```
# Number of flights each plane made
by_plane.count().show()

# Group by origin
by_origin = flights.groupBy("origin")

# Average duration of flights from PDX and SEA
by_origin.avg("air_time").show()
```

Grouping and Aggregating II

```
flights = flights.withColumn("dep_delay", flights.dep_delay.cast("float"))


# 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
by_month_dest.agg(F.stddev("dep_delay")).show()
```

Joining II

```
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 data again
print(flights_with_airports.show())
```

Getting started with machine learning pipelines

Join the DataFrames

```
from google.colab import files
files.upload()
planes = spark.read.csv('planes.csv', header=True)
planes.show()
```

```
planes.show()

# Rename year column
planes = planes.withColumnRenamed("year", "plane_year")

# Join the DataFrames
model_data = flights.join(planes, on="tailnum", how="leftouter")

model_data.show()
```

It's important to know that Spark only handles numeric data. That means all of the columns in your DataFrame must be either integers or decimals (called 'doubles' in Spark)

you can use the .cast() method in combination with the .withColumn() method. It's important to note that .cast() works on columns, while .withColumn() works on DataFrames.

The only argument you need to pass to .cast() is the kind of value you want to create, in string form. For example, to create integers, you'll pass the argument "integer" and for decimal numbers you'll use "double".

String to integer

```
# 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 a new column

```
# Create the column plane_age
model_data = model_data.withColumn("plane_age", model_data.year - model_data.plane_year)
```

Making a Boolean

```
# 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_ti
```

Strings and factors

All you have to remember is that you need to create a StringIndexer and a OneHotEncoder, and the Pipeline will take care of the rest.

Carrier

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder


# Create a StringIndexer
carr_indexer = StringIndexer(inputCol="carrier", outputCol="carrier_index")

# Create a OneHotEncoder
carr_encoder = OneHotEncoder(inputCol="carrier_index", outputCol="carrier_fact")
```

Destination

```
# Create a StringIndexer
dest_indexer = StringIndexer(inputCol="dest", outputCol="dest_index")

# Create a OneHotEncoder
dest_encoder = OneHotEncoder(inputCol="dest_index", outputCol="dest_fact")
```

Assemble a vector

```
from pyspark.ml.feature import VectorAssembler


# Make a VectorAssembler
vec_assembler = VectorAssembler(inputCols=["month", "air_time", "carrier_fact", "dest_fact
```

Create the pipeline

```
# Import Pipeline
from pyspark.ml import Pipeline

# Make the pipeline
flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr_indexer, carr_encoder, ve
```

Test vs Train

This never-before-seen data will give you a much more realistic idea of your model's performance in the real world when you're trying to predict or classify new data.

A test set approximates the 'real world error' of your model.

Transform the data

```
# Fit and transform the data
piped_data = flights_pipe.fit(model_data).transform(model_data)
```

```
piped_data.show()
```

Split the data

```
# Split the data into training and test sets
training, test = piped_data.randomSplit([.6, .4])
```

Model tuning and selection

Logistic regression is very similar to a linear regression, but instead of predicting a numeric variable, it predicts the probability (between 0 and 1) of an event.

A hyperparameter is just a value in the model that's not estimated from the data, but rather is supplied by the user to maximize performance.

Create the modeler

```
# Import LogisticRegression
from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression Estimator
lr = LogisticRegression()
```

Cross validation

The cross validation error is an estimate of the model's error on the test set.

Create the evaluator

```
# Import the evaluation submodule
import pyspark.ml.evaluation as evals

# Create a BinaryClassificationEvaluator
evaluator = evals.BinaryClassificationEvaluator(metricName="areaUnderROC")
# the curve is the ROC, or receiver operating curve.
```

Make a grid

```
# 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()
```

Make the validator

```
# Create the CrossValidator
cv = tune.CrossValidator(estimator=lr,
               estimatorParamMaps=grid,
               evaluator=evaluator)
```

Fit the model(s)

```
training.show()
```

```
# Call lr.fit()
best_lr = lr.fit(training)

# Print best_lr
print(best_lr)
```

```
# Fit cross validation models
models = cv.fit(training)
```

```
# Extract the best model
best_lr = models.bestModel
```

Evaluate the model

```
# Use the model to predict the test set
test_results = best_lr.transform(test)

# Evaluate the predictions
print(evaluator.evaluate(test_results))
```