Exercise Overview

In this exercise we will play with Spark Datasets & Dataframes (https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes), some Spark SQL (https://spark.apache.org/docs/latest/sql-programming-guide.html#sql), and build a couple of binary classifiaction models using Spark ML (https://spark.apache.org/docs/latest/ml-guide.html) (with some MLlib (https://spark.apache.org/mllib/) too).

The set up and approach will not be too dissimilar to the standard type of approach you might do in Sklearn (http://scikit-learn.org/stable/index.html). Spark has matured to the stage now where for 90% of what you need to do (when analysing tabular data) should be possible with Spark dataframes, SQL, and ML libraries. This is where this exercise is mainly trying to focus.

Feel free to adapt this exercise to play with other datasets readily availabe in the Databricks enviornment (they are listed in a cell below).

Getting Started

To get started you will need to create and attach a databricks spark cluster to this notebook. This notebook was developed on a cluster created with:

- Databricks Runtime Version 4.0 (includes Apache Spark 2.3.0, Scala 2.11)
- Python Version 3

Links & References

Some useful links and references of sources used in creating this exercise:

Note: Right click and open as new tab!

1. Latest Spark Docs (https://spark.apache.org/docs/latest/index.html)
2. Databricks Homepage (https://databricks.com/)
3. Databricks Community Edition FAQ (https://databricks.com/product/faq/community-edition)

4. Databricks Self Paced Training (https://databricks.com/training-overview/training-self-paced)
5. Databricks Notebook Guide (https://docs.databricks.com/user-guide/notebooks/index.html)
6. Databricks Binary Classification Tutorial (https://docs.databricks.com/spark/latest/mllib/binary-classification-mllib-pipelines.html#binary-classification)

Get Data

Here we will pull in some sample data that is already pre-loaded onto all databricks clusters.

Feel free to adapt this notebook later to play around with a different dataset if you like (all available are listed in a cell below).

```
# display datasets already in databricks
display(dbutils.fs.ls("/databricks-datasets"))
```

	path	name	size
1	dbfs:/databricks-datasets/	databricks-datasets/	0
2	dbfs:/databricks-datasets/COVID/	COVID/	0
3	dbfs:/databricks-datasets/README.md	README.md	976
4	dbfs:/databricks-datasets/Rdatasets/	Rdatasets/	0
5	dbfs:/databricks-datasets/SPARK_README.md	SPARK_README.md	3359
6	dbfs:/databricks-datasets/adult/	adult/	0
7	dbfs:/databricks-datasets/airlines/	airlines/	0

Showing all 50 rows.

Lets take a look at the '**adult**' dataset on the filesystem. This is the typical US Census data you often see online in tutorials. Here (https://archive.ics.uci.edu/ml/datasets/adult) is the same data in the UCI repository.

As an aside: here (https://github.com/GoogleCloudPlatform/cloudml-samples/tree/master/census) this same dataset is used as a quickstart example for Google CLoud ML & Tensorflow Estimator API (in case youd be interested in playing with tensorflow on the same dataset as here).

```
%fs ls databricks-datasets/adult/adult.data
```

	path	name	size
1	dbfs:/databricks-datasets/adult/adult.data	adult.data	3974305

Showing all 1 rows.

Note: Above %fs is just some file system cell magic that is specific to databricks. More info here (https://docs.databricks.com/user-guide/notebooks/index.html#mix-languages).

Spark SQL

Below we will use Spark SQL to load in the data and then register it as a Dataframe aswell. So the end result will be a Spark SQL table called *adult* and a Spark Dataframe called *df_adult*.

This is an example of the flexibility in Spark in that you could do lots of you ETL and data wrangling using either Spark SQL or Dataframes and pyspark. Most of the time it's a case of using whatever you are most comfortable with.

When you get more advanced then you might looking the pro's and con's of each and when you might favour one or the other (or operating direclty on RDD's), here (https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html) is a good article on the issues. For now, no need to overthink it!

```
%sql
-- drop the table if it already exists
DROP TABLE IF EXISTS adult
```

OK

```
%sql
-- create a new table in Spark SQL from the datasets already loaded in the
underlying filesystem.
-- In the real world you might be pointing at a file on HDFS or a hive table
etc.
CREATE TABLE adult (
  age DOUBLE,
  workclass STRING,
  fnlwgt DOUBLE,
  education STRING,
  education_num DOUBLE,
  marital_status STRING,
  occupation STRING,
  relationship STRING,
  race STRING,
  sex STRING,
  capital_gain DOUBLE,
  capital_loss DOUBLE,
  hours_per_week DOUBLE,
  native_country STRING,
  income STRING)
USING com.databricks.spark.csv
OPTIONS (path "/databricks-datasets/adult/adult.data", header "true")
```

OK

```
# look at the data
#spark.sql("SELECT * FROM adult LIMIT 5").show()
# this will look prettier in Databricks if you use display() instead
display(spark.sql("SELECT * FROM adult LIMIT 5"))
```

	age	workclass	fnlwgt	education	education_num	m
1	50	Self-emp-not-inc	83311	Bachelors	13	M
2	38	Private	215646	HS-grad	9	D
3	53	Private	234721	11th	7	M
4	28	Private	338409	Bachelors	13	M
5	37	Private	284582	Masters	14	M

Showing all 5 rows.

If you are more comfortable with SQL then as you can see below, its very easy to just get going with writing standard SQL type code to analyse your data, do data wrangling and create new dataframes.

```
# Lets get some summary marital status rates by occupation
result = spark.sql(
  """
  SELECT
    occupation,
    SUM(1) as n,
    ROUND(AVG(if(LTRIM(marital_status) LIKE 'Married-%',1,0)),2) as
married_rate,
    ROUND(AVG(if(lower(marital_status) LIKE '%widow%',1,0)),2) as widow_rate,
    ROUND(AVG(if(LTRIM(marital_status) = 'Divorced',1,0)),2) as divorce_rate,
    ROUND(AVG(if(LTRIM(marital_status) = 'Separated',1,0)),2) as
separated_rate,
    ROUND(AVG(if(LTRIM(marital_status) = 'Never-married',1,0)),2) as
bachelor_rate
  FROM
    adult
  GROUP BY 1
  ORDER BY n DESC
  """)
display(result)
```

	occupation	n	married_rate	widow_rate	divorce_rate
1	Prof-specialty	4140	0.53	0.02	0.13
2	Craft-repair	4099	0.64	0.01	0.11
3	Exec-managerial	4066	0.61	0.02	0.15
4	Adm-clerical	3769	0.28	0.04	0.22
5	Sales	3650	0.47	0.03	0.12
6	Other-service	3295	0.24	0.05	0.15
7	Machine-op-inspct	2002	0.51	0.03	0.14

Showing all 15 rows.

You can easily register dataframes as a table for Spark SQL too. So this way you can easily move between Dataframes and Spark SQL for whatever reason.

```
# register the df we just made as a table for spark sql
sqlContext.registerDataFrameAsTable(result, "result")
spark.sql("SELECT * FROM result").show(5)
```

```
+----------------+----+------------+----------+------------+--------------+----
---------+
|      occupation|   n|married_rate|widow_rate|divorce_rate|separated_rate|bach
elor_rate|
+----------------+----+------------+----------+------------+--------------+----
---------+
|  Prof-specialty|4140|        0.53|      0.02|        0.13|          0.02|
0.3|
|    Craft-repair|4099|        0.64|      0.01|        0.11|          0.03|
0.21|
| Exec-managerial|4066|        0.61|      0.02|        0.15|          0.02|
0.2|
|    Adm-clerical|3769|        0.28|      0.04|        0.22|          0.04|
0.42|
|           Sales|3650|        0.47|      0.03|        0.12|          0.03|
0.36|
+----------------+----+------------+----------+------------+--------------+----
---------+
only showing top 5 rows
```

Question 1

1. Write some spark sql to get the top 'bachelor_rate' by 'education' group?

```
### Question 1.1 Answer ###

result = spark.sql(
  """
  SELECT
    education,
    ROUND(AVG(if(LTRIM(marital_status) = 'Never-married',1,0)),2) as
bachelor_rate
  FROM
    adult
  GROUP BY 1
  ORDER BY 2 DESC
  """)
result.show()
```

```
+-------------+-------------+
|    education|bachelor_rate|
+-------------+-------------+
|         12th|         0.54|
|         11th|          0.5|
|    Preschool|         0.43|
| Some-college|          0.4|
|         10th|         0.39|
|    Bachelors|         0.34|
|   Assoc-acdm|         0.32|
|          9th|          0.3|
|      HS-grad|         0.29|
|      5th-6th|         0.27|
|    Assoc-voc|         0.26|
|      Masters|         0.23|
|      1st-4th|         0.23|
|    Doctorate|         0.18|
|      7th-8th|         0.17|
|  Prof-school|         0.16|
+-------------+-------------+
```

Spark DataFrames

Below we will create our DataFrame from the SQL table and do some similar analysis as we did with Spark SQL but using the DataFrames API.

```
# register a df from the sql df
df_adult = spark.table("adult")
cols = df_adult.columns # this will be used much later in the notebook, ignore
for now


# look at df schema
df_adult.printSchema()
```

```
root
 |-- age: double (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: double (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: double (nullable = true)
 |-- marital_status: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: double (nullable = true)
 |-- capital_loss: double (nullable = true)
 |-- hours_per_week: double (nullable = true)
 |-- native_country: string (nullable = true)
 |-- income: string (nullable = true)
```

```
# look at the df
display(df_adult)
#df_adult.show(5)
```

	age	workclass	fnlwgt	education	education_num	r
1	50	Self-emp-not-inc	83311	Bachelors	13	I
2	38	Private	215646	HS-grad	9	I
3	53	Private	234721	11th	7	I
4	28	Private	338409	Bachelors	13	I
5	37	Private	284582	Masters	14	I
6	49	Private	160187	9th	5	I
7	52	Self-emp-not-inc	209642	HS-grad	9	I

Showing the first 1000 rows.

Below we will do a similar calculation to what we did above but using the DataFrames API

```
# import what we will need
from pyspark.sql.functions import when, col, mean, desc, round

# wrangle the data a bit
df_result = df_adult.select(
  df_adult['occupation'],
  # create a 1/0 type col on the fly
  when( col('marital_status') == ' Divorced' , 1
).otherwise(0).alias('is_divorced')
)
# do grouping (and a round)
df_result =
df_result.groupBy('occupation').agg(round(mean('is_divorced'),2).alias('divorce
d_rate'))
# do ordering
df_result = df_result.orderBy(desc('divorced_rate'))
# show results
df_result.show(5)
```

```
+----------------+-------------+
|      occupation|divorced_rate|
+----------------+-------------+
|    Adm-clerical|         0.22|
| Priv-house-serv|         0.19|
|    Tech-support|         0.15|
| Exec-managerial|         0.15|
|   Other-service|         0.15|
+----------------+-------------+
only showing top 5 rows
```

As you can see the dataframes api is a bit more verbose then just expressing what you want to do in standard SQL.

But some prefer it and might be more used to it, and there could be cases where expressing what you need to do might just be better using the DataFrame API if it is too complicated for a simple SQL expression for example of maybe involves recursion of some type.

Question 2

1. Write some pyspark to get the top 'bachelor_rate' by 'education' group using DataFrame operations?

```
### Question 2.1 Answer ###

# wrangle the data a bit
df_result = df_adult.select(
  df_adult['education'],
  # create a 1/0 type col on the fly
  when( col('marital_status') == ' Never-married' , 1
).otherwise(0).alias('is_bachelor')
)
# do grouping (and a round)
df_result =
df_result.groupBy('education').agg(round(mean('is_bachelor'),2).alias('bachelor
_rate'))
df_result = df_result.orderBy(desc('bachelor_rate'))
df_result.show(1)

+---------+-------------+
|education|bachelor_rate|
+---------+-------------+
|     12th|         0.54|
+---------+-------------+
only showing top 1 row
```

Explore & Visualize Data

It's very easy to collect() (https://spark.apache.org/docs/latest/rdd-programming-guide.html#printing-elements-of-an-rdd) your Spark DataFrame data into a Pandas df and then continue to analyse or plot as you might normally.

Obviously if you try to collect() a huge DataFrame then you will run into issues, so usually you would only collect aggregated or sampled data into a Pandas df.

```
import pandas as pd

# do some analysis
result = spark.sql(
  """
  SELECT
    occupation,
    AVG(IF(income = ' >50K',1,0)) as plus_50k
  FROM
    adult
  GROUP BY 1
  ORDER BY 2 DESC
  """)

# collect results into a pandas df
df_pandas = pd.DataFrame(
  result.collect(),
  columns=result.schema.names
)

# look at df
print(df_pandas.head())

         occupation  plus_50k
0   Exec-managerial  0.484014
1    Prof-specialty  0.449034
2   Protective-serv  0.325116
3      Tech-support  0.304957
4             Sales  0.269315

print(df_pandas.describe())

        plus_50k
count  15.000000
mean    0.197357
std     0.143993
min     0.006711
25%     0.107373
50%     0.134518
75%     0.287136
max     0.484014

print(df_pandas.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15 entries, 0 to 14
```

```
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   occupation  15 non-null     object
 1   plus_50k    15 non-null     float64
dtypes: float64(1), object(1)
memory usage: 368.0+ bytes
None
```

Here we will just do some very basic plotting to show how you might collect what you are interested in into a Pandas DF and then just plot any way you normally would.

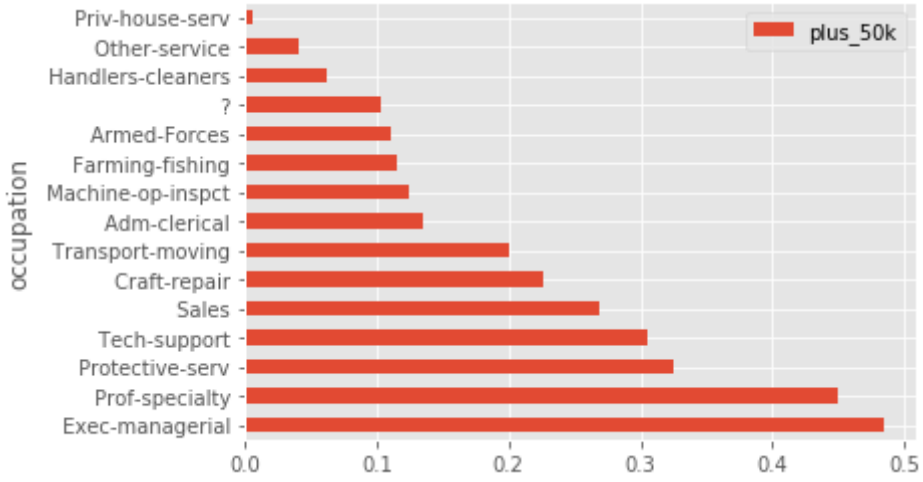
For simplicity we are going to use the plotting functionality built into pandas (you could make this a pretty as you want).

```
import matplotlib.pyplot as plt

# i like ggplot style
plt.style.use('ggplot')

# get simple plot on the pandas data
myplot = df_pandas.plot(kind='barh', x='occupation', y='plus_50k')

# display the plot (note - display() is a databricks function -
# more info on plotting in Databricks is here:
https://docs.databricks.com/user-guide/visualizations/matplotlib-and-
ggplot.html)
display(myplot.figure)
```



You can also easily get summary stats on a Spark DataFrame like below. Here (https://databricks.com/blog/2015/06/02/statistical-and-mathematical-functions-with-dataframes-in-spark.html) is a nice blog post that has more examples.

So this is an example of why you might want to move from Spark SQL into DataFrames API as being able to just call describe() on the Spark DF is easier then trying to do the equivilant in Spark SQL.

```
# describe df
df_adult.select(df_adult['age'],df_adult['education_num']).describe().show()
```

```
+-------+------------------+------------------+
|summary|               age|     education_num|
+-------+------------------+------------------+
|  count|             32560|             32560|
|   mean|38.581633906633904| 10.08058968058968|
| stddev|13.640641827464002|2.5727089681052058|
|    min|              17.0|               1.0|
|    max|              90.0|              16.0|
+-------+------------------+------------------+
```

ML Pipeline - Logistic Regression vs Random Forest

Below we will create two Spark ML Pipelines (https://spark.apache.org/docs/latest/ml-pipeline.html) - one that fits a logistic regression and one that fits a random forest. We will then compare the performance of each.

Note: A lot of the code below is adapted from this example (https://docs.databricks.com/spark/latest/mllib/binary-classification-mllib-pipelines.html).

```
spark.version
```

```
Out[25]: '3.0.1'
```

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

categoricalColumns = ["workclass", "education", "marital_status", "occupation",
"relationship", "race", "sex", "native_country"]
stages = [] # stages in our Pipeline

for categoricalCol in categoricalColumns:
    # Category Indexing with StringIndexer
    stringIndexer = StringIndexer(inputCol=categoricalCol,
outputCol=categoricalCol + "Index")
    # Use OneHotEncoder to convert categorical variables into binary
SparseVectors
    # encoder = OneHotEncoderEstimator(inputCol=categoricalCol + "Index",
outputCol=categoricalCol + "classVec")
    encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()],
outputCols=[categoricalCol + "classVec"])
    # Add stages.  These are not run here, but will run all at once later on.
    stages += [stringIndexer, encoder]


# Convert label into label indices using the StringIndexer
label_stringIdx = StringIndexer(inputCol="income", outputCol="label")
stages += [label_stringIdx]


# Transform all features into a vector using VectorAssembler
numericCols = ["age", "fnlwgt", "education_num", "capital_gain",
"capital_loss", "hours_per_week"]
assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
stages += [assembler]


# Create a Pipeline.
pipeline = Pipeline(stages=stages)
# Run the feature transformations.
#  - fit() computes feature statistics as needed.
#  - transform() actually transforms the features.
pipelineModel = pipeline.fit(df_adult)
dataset = pipelineModel.transform(df_adult)
# Keep relevant columns
selectedcols = ["label", "features"] + cols
dataset = dataset.select(selectedcols)
display(dataset)
```

	label	features
1	0	▸ {"vectorType": "sparse", "length": 100, "indices": [1, 10, 23, 31, 43, 48, 52, 53, 50, 83311, 13, 13]}
2	0	▸ {"vectorType": "sparse", "length": 100, "indices": [0, 8, 25, 38, 44, 48, 52, 53, 9 215646, 9, 40]}
3	0	▸ {"vectorType": "sparse", "length": 100, "indices": [0, 13, 23, 38, 43, 49, 52, 53, 53, 234721, 7, 40]}
4	0	▸ {"vectorType": "sparse", "length": 100, "indices": [0, 10, 23, 29, 47, 49, 62, 94, 338409, 13, 40]}

Showing the first 1000 rows.

```
### Randomly split data into training and test sets. set seed for
reproducibility
(trainingData, testData) = dataset.randomSplit([0.7, 0.3], seed=100)
print(trainingData.count())
print(testData.count())
```

```
22831
9729
```

```
from pyspark.sql.functions import avg

# get the rate of the positive outcome from the training data to use as a
threshold in the model
training_data_positive_rate =
trainingData.select(avg(trainingData['label'])).collect()[0][0]

print("Positive rate in the training data is
{}".format(training_data_positive_rate))
```

```
Positive rate in the training data is 0.2398931277648811
```

Logistic Regression - Train

```
from pyspark.ml.classification import LogisticRegression

# Create initial LogisticRegression model
lr = LogisticRegression(labelCol="label", featuresCol="features", maxIter=10)

# set threshold for the probability above which to predict a 1
lr.setThreshold(training_data_positive_rate)
# lr.setThreshold(0.5) # could use this if knew you had balanced data

# Train model with Training Data
lrModel = lr.fit(trainingData)

# get training summary used for eval metrics and other params
lrTrainingSummary = lrModel.summary

# Find the best model threshold if you would like to use that instead of the
empirical positve rate
fMeasure = lrTrainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-
Measure)').head()
lrBestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-
Measure)']) \
    .select('threshold').head()['threshold']

print("Best threshold based on model performance on training data is
{}".format(lrBestThreshold))

Best threshold based on model performance on training data is 0.382027978317939
```

GBM - Train

Question 3

1. Train a GBTClassifier on the training data, call the trained model 'gbModel'

```
### Question 3.1 Answer ###

from pyspark.ml.classification import GBTClassifier

# Create initial LogisticRegression model
gb = GBTClassifier(labelCol="label", featuresCol="features", maxIter=10)

# Train model with Training Data
gbModel = gb.fit(trainingData)
```

Logistic Regression - Predict

```
# make predictions on test data
lrPredictions = lrModel.transform(testData)

# display predictions
display(lrPredictions.select("label", "prediction", "probability"))
#display(lrPredictions)
```

Showing the first 1000 rows.

GBM - Predict

Question 4

1. Get predictions on the test data for your GBTClassifier. Call the predictions df 'gbPredictions'.