databricks Spark DF, SQL, ML Exercise

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
dbfs:/databricks-datasets/README.md
dbfs:/databricks-datasets/Rdatasets/
dbfs:/databricks-datasets/SPARK_README.md
dbfs:/databricks-datasets/adult/
dbfs:/databricks-datasets/airlines/
dbfs:/databricks-datasets/amazon/
dbfs:/databricks-datasets/asa/
dbfs:/databricks-datasets/atlas_higgs/
dbfs:/databricks-datasets/bikeSharing/

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
dbfs:/databricks-datasets/adult/adult.data

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	marital_status	occupatic
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-manageria
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-spec
37	Private	284582	Masters	14	Married-civ-spouse	Exec-manageria

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
Prof-specialty	4140	0.53	0.02
Craft-repair	4099	0.64	0.01
Exec-managerial	4066	0.61	0.02
Adm-clerical	3769	0.28	0.04
Sales	3650	0.47	0.03
Other-service	3295	0.24	0.05
Machine-op-inspct	2002	0.51	0.03
?	1843	0.36	0.08
Transport-moving	1597	0.63	0.02

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|bac
helor_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 bachelor_rate DESC
  """)

result.show(1)
```

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

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	marital_status	occupatic
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-manageria
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-spec
37	Private	284582	Masters	14	Married-civ-spouse	Exec-manageria

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|
|   Other-service|         0.15|
| Exec-managerial|         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'))
# do ordering
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):
```

```
occupation    15 non-null object
plus_50k      15 non-null float64
dtypes: float64(1), object(1)
memory usage: 320.0+ 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.

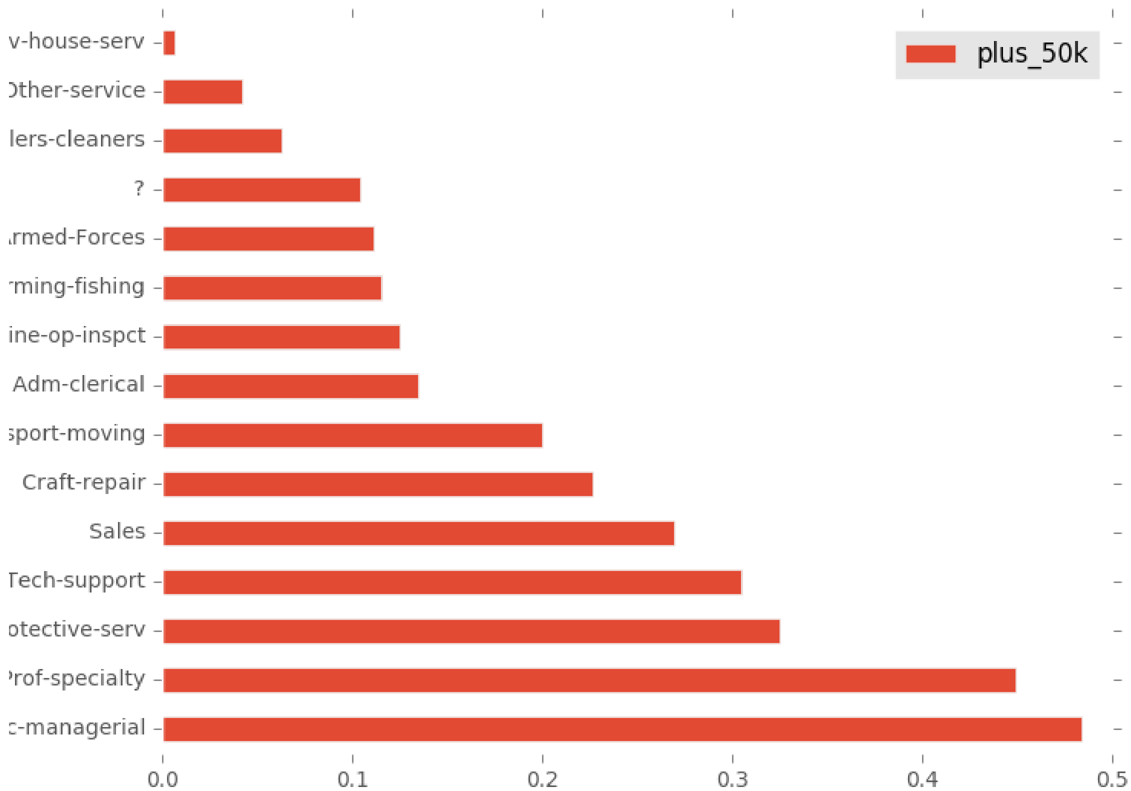
For simplicity we are going to use the plotting functionality built into pandas (you could make this a pretty as you want).

```
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).

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoderEstimator, 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 = OneHotEncoderEstimator(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	age	workclass	fnlwgt	education
0	[0,100, [1,10,23,31,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,50,83311,13,13]]	50	Self-emp-not-inc	83311	Bachelors
0	[0,100, [0,8,25,38,44,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,38,215646,9,40]]	38	Private	215646	HS-grad
0	[0,100, [0,13,23,38,43,49,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,53,234721,7,40]]	53	Private	234721	11th
0	[0,100,	28	Private	338409	Bachelors

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())
```

```
22837
9723
```

```
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 0.23934842580023646
```

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 0.34989688768486
854
```

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)
```

label	prediction	probability
0	1	▸ [1,2,[],[0.6912640989186452,0.3(
0	1	▸ [1,2,[],[0.621373486515505,0.378
0	1	▸ [1,2,[],[0.6586287948600472,0.34
0	1	▸ [1,2,[],[0.658995851028984,0.34
0	1	▸ [1,2,[],[0.6157704934546708,0.38
0	1	▸ [1,2,[],[0.5446870779706692,0.45
0	1	▸ [1,2,[],[0.6048473508705532,0.39
0	1	▸ [1,2,[],[0.59444809510805,0.405
0	1	▸ [1,2,[],[0.5875090690375259,0.4

Showing the first 1000 rows.

GBM - Predict

Question 4

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

```
### Question 4.1 Answer ###

# make predictions on test data
gbPredictions = gbModel.transform(testData)

display(gbPredictions)
```

label	features	age	workclass	fnlwgt	education	e
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,26,58426,9,50]]	26	Private	58426	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,30,83253,9,55]]	30	Private	83253	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,31,62374,9,50]]	31	Private	62374	HS-grad	9
0	[0,100,	32	Private	32732	HS-grad	9

Showing the first 1000 rows.

Logistic Regression - Evaluate

Question 5

1. Complete the print_performance_metrics() function below to also include measures of F1, Precision, Recall, False Positive Rate and True Positive Rate.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics,
MulticlassMetrics

def print_performance_metrics(predictions):
  # Evaluate model
  evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")
  auc = evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderROC"})
  aupr = evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderPR"})
  print("auc = {}".format(auc))
  print("aupr = {}".format(aupr))

  # get rdd of predictions and labels for mllib eval metrics
  predictionAndLabels = predictions.select("prediction","label").rdd

  # Instantiate metrics objects
  binary_metrics = BinaryClassificationMetrics(predictionAndLabels)
  multi_metrics = MulticlassMetrics(predictionAndLabels)

  # Area under precision-recall curve
  print("Area under PR = {}".format(binary_metrics.areaUnderPR))
  # Area under ROC curve
  print("Area under ROC = {}".format(binary_metrics.areaUnderROC))
  # Accuracy
  print("Accuracy = {}".format(multi_metrics.accuracy))
  # Confusion Matrix
  print(multi_metrics.confusionMatrix())

  ### Question 5.1 Answer ###

  # F1
  print("F1 = {}".format(multi_metrics.fMeasure()))
  # Precision
  print("Precision = {}".format(multi_metrics.precision()))
  # Recall
  print("Recall = {}".format(multi_metrics.recall()))
  # FPR
  print("FPR of label 0 = {}".format(multi_metrics.falsePositiveRate(0.0)))
  print("FPR of label 1 = {}".format(multi_metrics.falsePositiveRate(1.0)))
  # TPR
  print("TPR of label 0 = {}".format(multi_metrics.truePositiveRate(0.0)))
  print("TPR of label 1 = {}".format(multi_metrics.truePositiveRate(1.0)))

print_performance_metrics(lrPredictions)
```

```
auc = 0.9032867661805299
aupr = 0.7627830907418989
```

```
Area under PR = 0.5366100314564946
Area under ROC = 0.8159794860040686
Accuracy = 0.80067880283863
DenseMatrix([[ 5776.,  1572.],
             [  366.,  2009.]])
F1 = 0.80067880283863
Precision = 0.80067880283863
Recall = 0.80067880283863
FPR of label 0 = 0.15410526315789475
FPR of label 1 = 0.21393576483396842
TPR of label 0 = 0.7860642351660315
TPR of label 1 = 0.8458947368421053
```

GBM - Evaluate

```
print_performance_metrics(gbPredictions)
```

```
auc = 0.9033201157493619
aupr = 0.7737811350633501
Area under PR = 0.6526571258109415
Area under ROC = 0.7558290691344585
Accuracy = 0.8518975624807158
DenseMatrix([[ 6934.,   414.],
             [ 1026.,  1349.]])
F1 = 0.8518975624807158
Precision = 0.8518975624807158
Recall = 0.8518975624807158
FPR of label 0 = 0.432
FPR of label 1 = 0.05634186173108329
TPR of label 0 = 0.9436581382689168
TPR of label 1 = 0.568
```

Cross Validation

For each model you can run the below comand to see its params and a brief explanation of each.

```
print(lr.explainParams())
```

```
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha =
```

```
0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty. (default:
0.0)
family: The name of family which is a description of the label distribution to
be used in the model. Supported options: auto, binomial, multinomial (default:
auto)
featuresCol: features column name. (default: features, current: features)
fitIntercept: whether to fit an intercept term. (default: True)
labelCol: label column name. (default: label, current: label)
lowerBoundsOnCoefficients: The lower bounds on coefficients if fitting under b
ound constrained optimization. The bound matrix must be compatible with the sh
ape (1, number of features) for binomial regression, or (number of classes, nu
mber of features) for multinomial regression. (undefined)
lowerBoundsOnIntercepts: The lower bounds on intercepts if fitting under bound
constrained optimization. The bounds vector size must beequal with 1 for binom
ial regression, or the number oflasses for multinomial regression. (undefined)
maxIter: max number of iterations (>= 0). (default: 100, current: 10)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities. Not
```

```
print(gb.explainParams())
```

```
cacheNodeIds: If false, the algorithm will pass trees to executors to match in
stances with nodes. If true, the algorithm will cache node IDs for each instan
ce. Caching can speed up training of deeper trees. Users can set how often sho
uld the cache be checkpointed or disable it by setting checkpointInterval. (de
fault: False)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1).
E.g. 10 means that the cache will get checkpointed every 10 iterations. Note:
this setting will be ignored if the checkpoint directory is not set in the Spa
rkContext. (default: 10)
featuresCol: features column name. (default: features, current: features)
labelCol: label column name. (default: label, current: label)
lossType: Loss function which GBT tries to minimize (case-insensitive). Suppor
ted options: logistic (default: logistic)
maxBins: Max number of bins for discretizing continuous features.  Must be >=2
and >= number of categories for any categorical feature. (default: 32)
maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; d
epth 1 means 1 internal node + 2 leaf nodes. (default: 5)
maxIter: max number of iterations (>= 0). (default: 20, current: 10)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. If too
small, then 1 node will be split per iteration, and its aggregates may exceed
this size. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tree n
ode. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have after sp
lit. If a split causes the left or right child to have fewer than minInstances
PerNode, the split will be discarded as invalid. Should be >= 1. (default: 1)
```

```
predictionCol: prediction column name. (default: prediction)
seed: random seed. (default: -4419286272089722612)
stepSize: Step size (a.k.a. learning rate) in interval (0, 1] for shrinking th
e contribution of each estimator. (default: 0.1)
subsamplingRate: Fraction of the training data used for learning each decision
tree, in range (0, 1]. (default: 1.0)
```

Logisitic Regression - Param Grid

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Create ParamGrid for Cross Validation
lrParamGrid = (ParamGridBuilder()
             .addGrid(lr.regParam, [0.01, 0.5, 2.0])
             .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
             .addGrid(lr.maxIter, [2, 5])
             .build())
```

GBM - Param Grid

Question 6

1. Build out a param grid for the gb model, call it 'gbParamGrid'.

```
### Question 6.1 Answer ###

# Create ParamGrid for Cross Validation
gbParamGrid = (ParamGridBuilder()
             .addGrid(gb.maxDepth, [2,5])
             .addGrid(gb.maxIter, [10])
             .build())
```

Logistic Regression - Perform Cross Validation

```
# set up an evaluator
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")

# Create CrossValidator
lrCv = CrossValidator(estimator=lr, estimatorParamMaps=lrParamGrid,
evaluator=evaluator, numFolds=2)

# Run cross validations
lrCvModel = lrCv.fit(trainingData)
# this will likely take a fair amount of time because of the amount of models
that we're creating and testing


# below approach to getting at the best params from the best cv model taken
from:
# https://stackoverflow.com/a/46353730/1919374

# look at best params from the CV
print(lrCvModel.bestModel._java_obj.getRegParam())
print(lrCvModel.bestModel._java_obj.getElasticNetParam())
print(lrCvModel.bestModel._java_obj.getMaxIter())
```

```
0.01
0.0
5
```

GBM - Perform Cross Validation

Question 7

1. Perform cross validation of params on your 'gb' model.
2. Print out the best params you found.

```
### Question 7.1 Answer ###

# Create CrossValidator
gbCv = CrossValidator(estimator=gb, estimatorParamMaps=gbParamGrid,
evaluator=evaluator, numFolds=2)

# Run cross validations
gbCvModel = gbCv.fit(trainingData)
```

```
### Question 7.2 Answer ###

# look at best params from the CV
print(gbCvModel.bestModel._java_obj.getMaxDepth())
print(gbCvModel.bestModel._java_obj.getMaxIter())
```

```
5
10
```

Logistic Regression - CV Model Predict

```
# Use test set to measure the accuracy of our model on new data
lrCvPredictions = lrCvModel.transform(testData)

display(lrCvPredictions)
```

label	features	age	workclass	fnlwgt	education	e
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,26,58426,9,50]]	26	Private	58426	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,30,83253,9,55]]	30	Private	83253	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,31,62374,9,50]]	31	Private	62374	HS-grad	9
0	[0,100,	32	Private	32732	HS-grad	9

Showing the first 1000 rows.

GBM - CV Model Predict

```
gbCvPredictions = gbCvModel.transform(testData)

display(gbCvPredictions)
```

label	features	age	workclass	fnlwgt	education	e

0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,26,58426,9,50]]	26	Private	58426	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,30,83253,9,55]]	30	Private	83253	HS-grad	9
0	[0,100, [0,8,23,29,43,48,52,53,94,95,96,99], [1,1,1,1,1,1,1,1,31,62374,9,50]]	31	Private	62374	HS-grad	9

Showing the first 1000 rows.

Logistic Regression - CV Model Evaluate

```
print_performance_metrics(lrCvPredictions)
```

```
auc = 0.8857251239148407
aupr = 0.7241684115655654
Area under PR = 0.5010764347874727
Area under ROC = 0.798374810188236
Accuracy = 0.773423840378484
DenseMatrix([[ 5508.,  1840.],
             [  363.,  2012.]])
F1 = 0.773423840378484
Precision = 0.773423840378484
Recall = 0.773423840378484
FPR of label 0 = 0.1528421052631579
FPR of label 1 = 0.25040827436037016
TPR of label 0 = 0.7495917256396298
TPR of label 1 = 0.8471578947368421
```

GBM - CV Model Evaluate

```
print_performance_metrics(gbCvPredictions)
```

```
auc = 0.9033201157493619
aupr = 0.7737811350633501
Area under PR = 0.6526571258109415
Area under ROC = 0.7558290691344585
```

```
Accuracy = 0.8518975624807158
DenseMatrix([[ 6934.,   414.],
             [ 1026.,  1349.]])
F1 = 0.8518975624807158
Precision = 0.8518975624807158
Recall = 0.8518975624807158
FPR of label 0 = 0.432
FPR of label 1 = 0.05634186173108329
TPR of label 0 = 0.9436581382689168
TPR of label 1 = 0.568
```

Logistic Regression - Model Explore

```
print('Model Intercept: ', lrCvModel.bestModel.intercept)
```

```
Model Intercept:  -1.2479134417997437
```

```
lrWeights = lrCvModel.bestModel.coefficients
lrWeights = [(float(w),) for w in lrWeights]  # convert numpy type to float, and to tuple
lrWeightsDF = sqlContext.createDataFrame(lrWeights, ["Feature Weight"])
display(lrWeightsDF)
```

Feature Weight
-0.22413336981436807
-0.345555382229601
-0.13203533849479399
-0.4680986801474524
-0.2455358869288455
0.4322838456317811
0.4075811047761166
-1.1597488763666135
-0.4380095204656965

Feature Importance

Question 8

1. Print out a table of feature_name and feature_coefficient from the Logistic Regression model.

   Hint: Adapt the code from here: https://stackoverflow.com/questions/42935914/how-to-map-features-from-the-output-of-a-vectorassembler-back-to-the-column-name (https://stackoverflow.com/questions/42935914/how-to-map-features-from-the-output-of-a-vectorassembler-back-to-the-column-name)

```
### Question 8.1 Answer ###

# from: https://stackoverflow.com/questions/42935914/how-to-map-features-from-
the-output-of-a-vectorassembler-back-to-the-column-name
# fill in here
from itertools import chain

transformed = lrCvModel.bestModel.transform(trainingData)
attrs = sorted((attr["idx"], attr["name"]) for attr in
(chain(*transformed.schema["features"].metadata["ml_attr"]["attrs"].values())))



gbCvFeatureImportance = pd.DataFrame([(name,
gbCvModel.bestModel.featureImportances[idx]) for idx, name in attrs],columns=
['feature_name','feature_importance'])

print(gbCvFeatureImportance.sort_values(by=['feature_importance'],ascending
=False))
```

```
                                  feature_name  feature_importance
23     marital_statusclassVec_ Married-civ-spouse            0.213425
97                                 capital_gain            0.140075
94                                          age            0.134091
96                                education_num            0.100899
99                               hours_per_week            0.087607
98                                 capital_loss            0.086795
31            occupationclassVec_ Exec-managerial            0.063332
29             occupationclassVec_ Prof-specialty            0.024545
1             workclassclassVec_ Self-emp-not-inc            0.022247
34              occupationclassVec_ Other-service            0.020495
```

39	occupationclassVec_ Farming-fishing	0.015995
40	occupationclassVec_ Tech-support	0.015653
52	sexclassVec_ Male	0.013727
6	workclassclassVec_ Federal-gov	0.011737
10	educationclassVec_ Bachelors	0.010395
33	occupationclassVec_ Sales	0.005770
43	relationshipclassVec_ Husband	0.004145
21	educationclassVec_ 5th-6th	0.002799
47	relationshipclassVec_ Wife	0.002646

Question 9

1. Build and train a RandomForestClassifier and print out a table of feature importances from it.

	feature_name	feature_importance
43	relationshipclassVec_ Husband	0.223539
96	education_num	0.156350
24	marital_statusclassVec_ Never-married	0.103739
23	marital_statusclassVec_ Married-civ-spouse	0.088150
97	capital_gain	0.078490
52	sexclassVec_ Male	0.052675
99	hours_per_week	0.048638
44	relationshipclassVec_ Not-in-family	0.035927
94	age	0.035832
47	relationshipclassVec_ Wife	0.025968
11	educationclassVec_ Masters	0.022330
29	occupationclassVec_ Prof-specialty	0.019274
34	occupationclassVec_ Other-service	0.017977
46	relationshipclassVec_ Unmarried	0.016186
10	educationclassVec_ Bachelors	0.015504
45	relationshipclassVec_ Own-child	0.009791
98	capital_loss	0.009307
31	occupationclassVec_ Exec-managerial	0.008882
8	educationclassVec_ HS-grad	0.008132
27	marital_statusclassVec_ Widowed	0.004711