databricks DataBricks Finished running online

(https://databricks.com)

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"))
```

Table

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/COVID/	COVID/	0	1725169514125
2	dbfs:/databricks-datasets/README.md	README.md	976	1532468253000
3	dbfs:/databricks-datasets/Rdatasets/	Rdatasets/	0	1725169514125
4	dbfs:/databricks-datasets/SPARK_README.md	SPARK_README.md	3359	1455043490000
5	dbfs:/databricks-datasets/adult/	adult/	0	1725169514125
6	dbfs:/databricks-datasets/airlines/	airlines/	0	1725169514125
7	dbfs:/databricks-datasets/amazon/	amazon/	0	1725169514125
8	dbfs:/databricks-datasets/asa/	asa/	0	1725169514125
9	dbfs:/databricks-datasets/atlas_higgs/	atlas_higgs/	0	1725169514125

10	dbfs:/databricks-datasets/bikeSharing/	bikeSharing/	0	1725169514125
11	dbfs:/databricks-datasets/cctvVideos/	cctvVideos/	0	1725169514125
12	dbfs:/databricks-datasets/credit-card-fraud/	credit-card-fraud/	0	1725169514125
13	dbfs:/databricks-datasets/cs100/	cs100/	0	1725169514125
14	dbfs:/databricks-datasets/cs110x/	cs110x/	0	1725169514125
15	dbfs:/databricks-datasets/cs190/	cs190/	0	1725169514125

55 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
```

Table

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/adult/adult.data	adult.data	3974305	1444260537000

1 row

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
-- Cambia al catálogo de Spark SQL por defecto (no gestionado por Unity Catalog)
USE CATALOG spark_catalog;

-- Crear una base de datos si es necesario
CREATE DATABASE IF NOT EXISTS my_database;

-- Usar esa base de datos
USE my_database;

-- Luego crear la tabla como antes
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 CSV
OPTIONS (
  path "dbfs:/databricks-datasets/adult/adult.data",
  header "true",
  inferSchema "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"))
```

Table

	1.2 age	ABC workclass	1.2 fnlwgt	ABC education	1.2 education_num	ABC marital_status	ABC occupation
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty
5	37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial

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)
```

Table

	occupation	n	married_rate	widow_rate	divorce_rate	separated_rate	bachel
1	Prof-specialty	4140	0.53	0.02	0.13	0.02	
2	Craft-repair	4099	0.64	0.01	0.11	0.03	
3	Exec-managerial	4066	0.61	0.02	0.15	0.02	
4	Adm-clerical	3769	0.28	0.04	0.22	0.04	
5	Sales	3650	0.47	0.03	0.12	0.03	
6	Other-service	3295	0.24	0.05	0.15	0.06	
7	Machine-op-inspct	2002	0.51	0.03	0.14	0.04	
8	?	1843	0.36	0.08	0.1	0.04	
9	Transport-moving	1597	0.63	0.02	0.11	0.02	
10	Handlers-cleaners	1370	0.36	0.01	0.09	0.03	
11	Farming-fishing	994	0.6	0.02	0.06	0.02	
12	Tech-support	928	0.44	0.02	0.15	0.03	
13	Protective-serv	649	0.6	0.01	0.12	0.02	
14	Priv-house-serv	149	0.13	0.15	0.19	0.08	
15	Armed-Forces	9	0.33	0	0	0	

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|bachelor_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,
    SUM(1) as n,
    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|  n|bachelor_rate|
+---------+---+-------------+
|     12th|433|         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)
```

Table

	1.2 age	workclass	1.2 fnlwgt	education	1.2 education_num	marital_status	occupation
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaner
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaner
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty
5	37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial
6	49	Private	160187	9th	5	Married-spouse-absent	Other-service
7	52	Self-emp-not-inc	209642	HS-grad	9	Married-civ-spouse	Exec-managerial
8	31	Private	45781	Masters	14	Never-married	Prof-specialty
9	42	Private	159449	Bachelors	13	Married-civ-spouse	Exec-managerial
10	37	Private	280464	Some-college	10	Married-civ-spouse	Exec-managerial
11	30	State-gov	141297	Bachelors	13	Married-civ-spouse	Prof-specialty
12	23	Private	122272	Bachelors	13	Never-married	Adm-clerical
13	32	Private	205019	Assoc-acdm	12	Never-married	Sales
14	40	Private	121772	Assoc-voc	11	Married-civ-spouse	Craft-repair
15	34	Private	245487	7th-8th	4	Married-civ-spouse	Transport-moving

10,000+ rows | Truncated data due to row limit

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('divorced_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|
| Exec-managerial|         0.15|
|    Tech-support|         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

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):
 #   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.

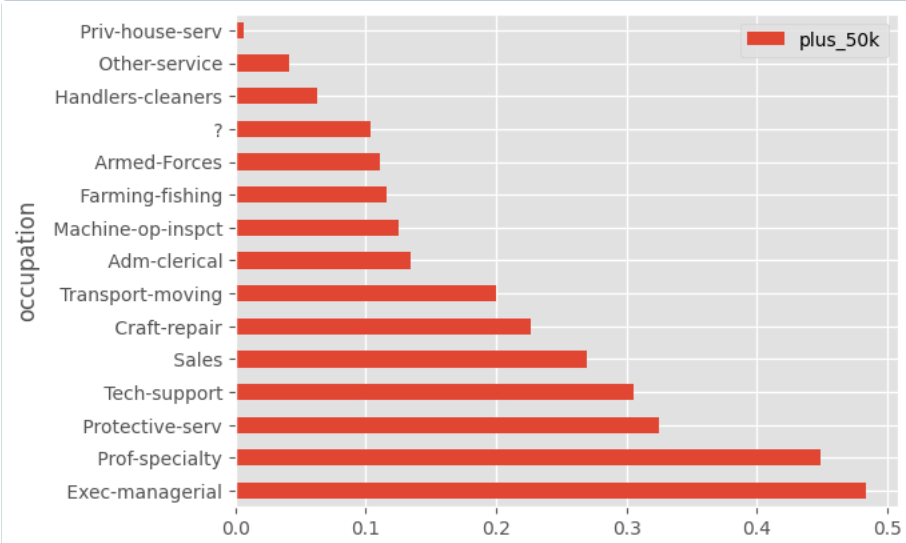
For simplicity we are going to use the plotting functionality built into pandas (you could make this a pretty as you want).

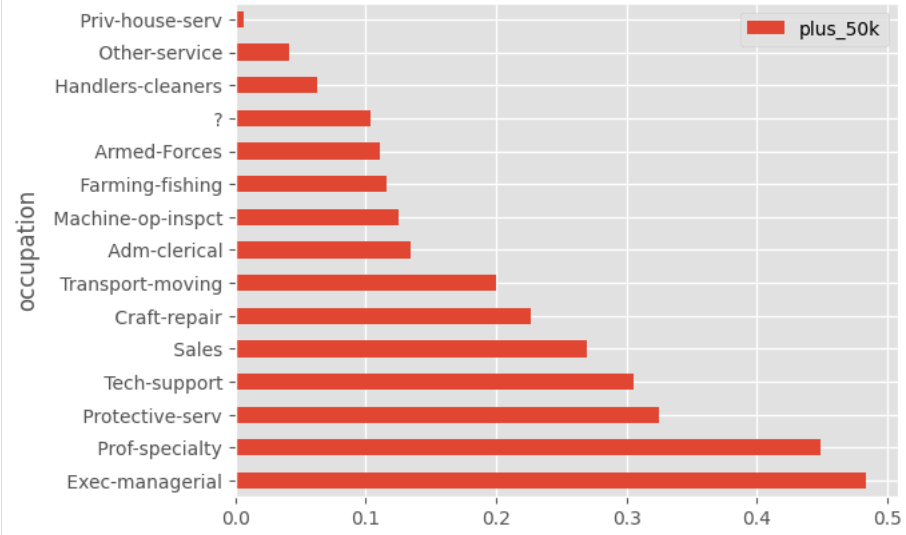
```
import matplotlib.pyplot as plt

# 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-ggp
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|
+-------+------------------+------------------+
```

```
(df_adult.count(), len(df_adult.columns))
```

```
(32560, 15)
```

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 OneHotEncoder, StringIndexer, VectorAssembler

categoricalColumns = ["workclass", "education", "marital_status", "occupation", "relationship", "race", "sex", "native_
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 = OneHotEncoder(inputCol=stringIndexer.getOutputCol(), outputCol=categoricalCol + "classVec")
    # Add stages. These are not run here, but will run all at once later on.
    stages += [stringIndexer, encoder]

# Now you can create the pipeline and fit the model
pipeline = Pipeline(stages=stages)
```

```
# 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)
```

Table

	1.2 label	features	1.2 ag
1	0	{"vectorType":"sparse","length":100,"indices":[1,10,23,31,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,50,83311,1...	
2	0	{"vectorType":"sparse","length":100,"indices":[0,8,25,38,44,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,38,215646,9,...	
3	0	{"vectorType":"sparse","length":100,"indices":[0,13,23,38,43,49,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,53,234721,...	
4	0	{"vectorType":"sparse","length":100,"indices":[0,10,23,29,47,49,62,94,95,96,99],"values":[1,1,1,1,1,1,1,28,338409,13,40]}	
5	0	{"vectorType":"sparse","length":100,"indices":[0,11,23,31,47,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,37,284582,14,40]}	
6	0	{"vectorType":"sparse","length":100,"indices":[0,18,28,34,44,49,64,94,95,96,99],"values":[1,1,1,1,1,1,1,49,160187,5,16]}	
7	1	{"vectorType":"sparse","length":100,"indices":[1,8,23,31,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,52,209642,9,...	
8	1	{"vectorType":"sparse","length":100,"indices":[0,11,24,29,44,48,53,94,95,96,97,99],"values":[1,1,1,1,1,1,1,31,45781,14,...	
9	1	{"vectorType":"sparse","length":100,"indices":[0,10,23,31,43,48,52,53,94,95,96,97,99],"values":[1,1,1,1,1,1,1,1,42,1594...	
10	1	{"vectorType":"sparse","length":100,"indices":[0,9,23,31,43,49,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,37,280464,1...	
11	1	{"vectorType":"sparse","length":100,"indices":[4,10,23,29,43,50,52,61,94,95,96,99],"values":[1,1,1,1,1,1,1,1,30,141297,...	
12	0	{"vectorType":"sparse","length":100,"indices":[0,10,24,32,45,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,23,122272,13,30]}	
13	0	{"vectorType":"sparse","length":100,"indices":[0,14,24,33,44,49,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,32,205019,...	
14	1	{"vectorType":"sparse","length":100,"indices":[0,12,23,30,43,50,52,55,94,95,96,99],"values":[1,1,1,1,1,1,1,1,40,121772,...	
15	0	{"vectorType":"sparse","length":100,"indices":[0,16,23,37,43,51,52,54,94,95,96,99],"values":[1,1,1,1,1,1,1,1,34,245487,...	

6,816+ rows | Truncated data due to byte limit

```
### 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 0.3178139130754516
```

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)
```

Table

	1.2 label	1.2 prediction	probability
83	0	1	{"vectorType":"dense","length":2,"values":[0.7589416976824479,0.2410583023175521]}
84	0	0	{"vectorType":"dense","length":2,"values":[0.7844306934645349,0.2155693065354651...
85	0	0	{"vectorType":"dense","length":2,"values":[0.7656281743002852,0.2343718256997148...
86	0	0	{"vectorType":"dense","length":2,"values":[0.7649655653949222,0.2350344346050777...
87	0	0	{"vectorType":"dense","length":2,"values":[0.7850951078266734,0.2149048921733266...
88	0	1	{"vectorType":"dense","length":2,"values":[0.7283171398962889,0.2716828601037111]}
89	0	1	{"vectorType":"dense","length":2,"values":[0.7413766064735635,0.2586233935264365]}

90	0	0	{"vectorType":"dense","length":2,"values":[0.7956711872334553,0.2043288127665447]}
91	0	0	{"vectorType":"dense","length":2,"values":[0.774026369592778,0.22597363040722196]}
92	0	0	{"vectorType":"dense","length":2,"values":[0.8030151713871715,0.1969848286128285]}
93	0	1	{"vectorType":"dense","length":2,"values":[0.7220321595014013,0.2779678404985987]}
94	0	1	{"vectorType":"dense","length":2,"values":[0.7549269504778195,0.2450730495221804...
95	0	0	{"vectorType":"dense","length":2,"values":[0.7756117691404524,0.2243882308595476...
96	0	0	{"vectorType":"dense","length":2,"values":[0.7735397057623353,0.2264602942376646...
97	0	1	{"vectorType":"dense","length":2,"values":[0.7580417690926209,0.2419582309073791]}
98	0	0	{"vectorType":"dense","length":2,"values":[0.7616381528872328,0.2383618471127672]}

9,729 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.select("label", "prediction", "probability"))
```

Table

	1.2 label	1.2 prediction	probability
1	0	1	{"vectorType":"dense","length":2,"values":[0.24926130440745373,0.750738695592546...
2	0	0	{"vectorType":"dense","length":2,"values":[0.7149309049405317,0.2850690950594683]}
3	0	0	{"vectorType":"dense","length":2,"values":[0.6719301883509003,0.3280698116490996...
4	0	0	{"vectorType":"dense","length":2,"values":[0.6433580713409047,0.3566419286590953]}
5	0	0	{"vectorType":"dense","length":2,"values":[0.6339521110230912,0.3660478889769087...
6	0	0	{"vectorType":"dense","length":2,"values":[0.6297487225219812,0.3702512774780188]}
7	0	0	{"vectorType":"dense","length":2,"values":[0.6297487225219812,0.3702512774780188]}
8	0	0	{"vectorType":"dense","length":2,"values":[0.7366400516415794,0.2633599483584206]}
9	0	0	{"vectorType":"dense","length":2,"values":[0.701704923328239,0.2982950766717610 3]}
10	0	0	{"vectorType":"dense","length":2,"values":[0.8312569877866142,0.1687430122133858...
11	0	0	{"vectorType":"dense","length":2,"values":[0.5384283521870047,0.4615716478129953]}
12	0	0	{"vectorType":"dense","length":2,"values":[0.6433580713409047,0.3566419286590953]}
13	0	0	{"vectorType":"dense","length":2,"values":[0.6773644137715346,0.3226355862284654...
14	0	0	{"vectorType":"dense","length":2,"values":[0.7176989874265829,0.2823010125734171]}
15	0	0	{"vectorType":"dense","length":2,"values":[0.7176989874265829,0.2823010125734171]}

9,729 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.

```
def print_performance_metrics(predictions):
    from pyspark.ml.evaluation import BinaryClassificationEvaluator
    from pyspark.mllib.evaluation import BinaryClassificationMetrics, MulticlassMetrics
    from pyspark.sql import SparkSession

    # Create a Spark session (recommended for newer versions)
    spark = SparkSession.builder.getOrCreate()

    # 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.map(lambda row: (row[0], row[1]))

    # 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())

    # F1 for label 1 (typically the positive class in binary classification)
    print("F1 (label 1) = {}".format(multi_metrics.fMeasure(1.0)))
    # F1 for label 0 (the negative class)
    print("F1 (label 0) = {}".format(multi_metrics.fMeasure(0.0)))

    # Precision for label 1
    print("Precision (label 1) = {}".format(multi_metrics.precision(1.0)))
    # Precision for label 0
    print("Precision (label 0) = {}".format(multi_metrics.precision(0.0)))

    # Recall for label 1
    print("Recall (label 1) = {}".format(multi_metrics.recall(1.0)))
    # Recall for label 0
    print("Recall (label 0) = {}".format(multi_metrics.recall(0.0)))

    # 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 = 0.9022382581905842
aupr = 0.7639327846135503
/databricks/spark/python/pyspark/sql/context.py:164: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOr
Create() instead.
  warnings.warn(
Area under PR = 0.550904296478963
Area under ROC = 0.8214764233357801
Accuracy = 0.8119025593586185
DenseMatrix([[5913., 1452.],
             [ 378., 1986.]])
F1 (label 1) = 0.6845915201654602
F1 (label 0) = 0.8659929701230228
```

```
Precision (label 1) = 0.5776614310645725
Precision (label 0) = 0.9399141630901288
Recall (label 1) = 0.8401015228426396
Recall (label 0) = 0.8028513238289205
FPR of label 0 = 0.1598984771573604
FPR of label 1 = 0.19714867617107942
TPR of label 0 = 0.8028513238289205
TPR of label 1 = 0.8401015228426396
```

GBM - Evaluate

```
print_performance_metrics(gbPredictions)
```

```
auc = 0.9058602504413908
aupr = 0.7801471500806201
/databricks/spark/python/pyspark/sql/context.py:164: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOr
Create() instead.
  warnings.warn(
Area under PR = 0.6483663654287593
Area under ROC = 0.7396787694576833
Accuracy = 0.8478774796998664
DenseMatrix([[6998.,  367.],
             [1113., 1251.]])
F1 (label 1) = 0.628327473631341
F1 (label 0) = 0.9043680537606616
Precision (label 1) = 0.7731767614338689
Precision (label 0) = 0.8627789421772901
Recall (label 1) = 0.5291878172588832
Recall (label 0) = 0.9501697216564834
FPR of label 0 = 0.47081218274111675
FPR of label 1 = 0.04983027834351663
TPR of label 0 = 0.9501697216564834
TPR of label 1 = 0.5291878172588832
```

Cross Validation

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

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

```
than remaining data size in a partition then it is adjusted to the data size. Default 0.0 represents choosing optimal v
alue, depends on specific algorithm. Must be >= 0. (default: 0.0)
maxIter: max number of iterations (>= 0). (default: 100, current: 10)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated
probability estimates! These probabilities should be treated as confidences, not precise probabilities. (default: proba
bility)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (default: rawPrediction)
regParam: regularization parameter (>= 0). (default: 0.0)
standardization: whether to standardize the training features before fitting the model. (default: True)
threshold: Threshold in binary classification prediction, in range [0, 1]. If threshold and thresholds are both set, th
ey must match.e.g. if threshold is p, then thresholds must be equal to [1-p, p]. (default: 0.5, current: 0.239893127764
8811)
thresholds: Thresholds in multi-class classification to adjust the probability of predicting each class. Array must hav
e length equal to the number of classes, with values > 0, excepting that at most one value may be 0. The class with lar
gest value p/t is predicted, where p is the original probability of that class and t is the class's threshold. (undefin
ed)
tol: the convergence tolerance for iterative algorithms (>= 0). (default: 1e-06)
upperBoundsOnCoefficients: The upper bounds on coefficients if fitting under bound constrained optimization. The bound
matrix must be compatible with the shape (1, number of features) for binomial regression, or (number of classes, number
of features) for multinomial regression. (undefined)
```

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

```
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 ite
ration, and its aggregates may exceed this size. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have after split. If a split causes the left or right
child to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be >= 1. (default: 1)
minWeightFractionPerNode: Minimum fraction of the weighted sample count that each child must have after split. If a spl
it causes the fraction of the total weight in the left or right child to be less than minWeightFractionPerNode, the spl
it will be discarded as invalid. Should be in interval [0.0, 0.5). (default: 0.0)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated
probability estimates! These probabilities should be treated as confidences, not precise probabilities. (default: proba
bility)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (default: rawPrediction)
seed: random seed. (default: -8262332687822256406)
stepSize: Step size (a.k.a. learning rate) in interval (0, 1] for shrinking the contribution of each estimator. (defaul
t: 0.1)
subsamplingRate: Fraction of the training data used for learning each decision tree, in range (0, 1]. (default: 1.0)
thresholds: Thresholds in multi-class classification to adjust the probability of predicting each class. Array must hav
e length equal to the number of classes, with values > 0, excepting that at most one value may be 0. The class with lar
gest value p/t is predicted, where p is the original probability of that class and t is the class's threshold. (undefin
```

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 ###
import numpy as np

# Create ParamGrid for Cross Validation
gbParamGrid = (ParamGridBuilder()
#             .addGrid(gb.maxDepth, np.arange(2, 10))
#             .addGrid(gb.maxIter, [50])
             .addGrid(gb.maxDepth, np.arange(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())
```

```
4
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)
```

Table

	1.2 label	features	1.2 ag
1	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,98,99],"values":[1,1,1,1,1,1,1,1,36,37076...	
2	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,32,131224,9,...	
3	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,33,279173,9,...	
4	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,39,50700,9,4...	
5	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,39,181705,9,...	
6	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,50,81548,9,4...	
7	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,51,94432,9,4...	
8	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,60,198727,9,...	
9	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,43,48,52,55,94,95,96,99],"values":[1,1,1,1,1,1,1,1,34,241360,9,...	
10	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,45,48,55,94,95,96,99],"values":[1,1,1,1,1,1,1,20,219266,9,36]}	
11	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,47,48,53,94,95,96,97,99],"values":[1,1,1,1,1,1,1,35,145704,9,3...	
12	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,47,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,42,334783,9,40]}	
13	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,47,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,43,216685,9,35]}	
14	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,47,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,48,199763,9,8]}	
15	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,29,47,48,53,94,95,96,99],"values":[1,1,1,1,1,1,1,50,287927,9,16]}	

4,356+ rows | Truncated data due to byte limit

GBM - CV Model Predict

```
gbCvPredictions = gbCvModel.transform(testData)

display(gbCvPredictions)
```

Table

	1.2 label	features	1.2 ag
35	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,25,81286,9,4...	
36	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,25,187577,9,...	
37	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,88449,9,3...	
38	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,121559,9,...	
39	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,188626,9,...	
40	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,193304,9,...	
41	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,213258,9,...	
42	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,26,262617,9,...	
43	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,122038,9,...	
44	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,152951,9,...	
45	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,160972,9,...	
46	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,165412,9,...	
47	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,189777,9,...	
48	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,410351,9,...	
49	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,27,588905,9,...	
50	0	{"vectorType":"sparse","length":100,"indices":[0,8,23,30,43,48,52,53,94,95,96,99],"values":[1,1,1,1,1,1,1,1,28,96226,9,4...	

4,351+ rows | Truncated data due to byte limit

Logistic Regression - CV Model Evaluate

```
print_performance_metrics(lrCvPredictions)
```

```
auc = 0.8970633845772201
aupr = 0.7460869514036187
/databricks/spark/python/pyspark/sql/context.py:164: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOr
Create() instead.
  warnings.warn(
Area under PR = 0.5330158297497791
Area under ROC = 0.8134734010841509
Accuracy = 0.799568300955905
DenseMatrix([[5792., 1573.],
             [ 377., 1987.]])
F1 (label 1) = 0.6708305199189737
F1 (label 0) = 0.8559184276636619
Precision (label 1) = 0.5581460674157304
Precision (label 0) = 0.9388879883287404
Recall (label 1) = 0.8405245346869712
Recall (label 0) = 0.7864222674813306
FPR of label 0 = 0.15947546531302875
FPR of label 1 = 0.21357773251866938
TPR of label 0 = 0.7864222674813306
TPR of label 1 = 0.8405245346869712
```

GBM - CV Model Evaluate

```
print_performance_metrics(gbCvPredictions)
```

```
auc = 0.9022715994499985
aupr = 0.7700419514589649
/databricks/spark/python/pyspark/sql/context.py:164: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOr
Create() instead.
  warnings.warn(
Area under PR = 0.6448460645818491
Area under ROC = 0.7354408110799812
Accuracy = 0.8460273409394593
DenseMatrix([[7001.,  364.],
             [1134., 1230.]])
F1 (label 1) = 0.6215260232440627
F1 (label 0) = 0.9033548387096775
Precision (label 1) = 0.7716436637390214
Precision (label 0) = 0.8606023355869699
Recall (label 1) = 0.5203045685279187
Recall (label 0) = 0.9505770536320435
FPR of label 0 = 0.4796954314720812
FPR of label 1 = 0.04942294636795655
TPR of label 0 = 0.9505770536320435
TPR of label 1 = 0.5203045685279187
```

Logistic Regression - Model Explore

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

```
Model Intercept:  -5.9821391466705025
```

```
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)
```

Table

	1.2 Feature Weight

1	0.10728563291425162
2	-0.2961958701513941
3	-0.08240276890450877
4	-0.2781053241320957
5	-0.17741325409202524
6	0.3883038764029745
7	0.447508064612638
8	-1.8866287502815302
9	-0.19640038621592631
10	-0.0041308974494214746
11	0.3817931077161157
12	0.6284988311345274
13	0.09229260575747907
14	-0.550976358418968
15	-0.10623104529190863

100 rows

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-t
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"]["attr
```

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

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

```
                               feature_name  feature_importance
94                                      age            0.206924
23  marital_statusclassVec_ Married-civ-spouse            0.205673
99                             hours_per_week            0.141034
96                              education_num            0.122204
97                               capital_gain            0.109436
..                                        ...                 ...
36                     occupationclassVec_ ?            0.000000
35       occupationclassVec_ Machine-op-inspct            0.000000
33                    occupationclassVec_ Sales            0.000000
32            occupationclassVec_ Adm-clerical            0.000000
50            raceclassVec_ Asian-Pac-Islander            0.000000

[100 rows x 2 columns]
```

Question 9

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

```
                               feature_name  feature_importance
97                              capital_gain            0.234006
43              relationshipclassVec_ Husband            0.182959
23  marital_statusclassVec_ Married-civ-spouse            0.105683
24       marital_statusclassVec_ Never-married            0.101128
99                              hours_per_week            0.083681
..                                         ...                 ...
66               native_countryclassVec_ China            0.000000
67               native_countryclassVec_ Italy            0.000000
13                    educationclassVec_ 11th            0.000000
69             native_countryclassVec_ Vietnam            0.000000
0                   workclassclassVec_ Private            0.000000

[100 rows x 2 columns]
```