Big Data Analytics Techniques and Applications _ HW4

310712009 楊家碩 (Nick Yang) GMBA

○ Q1: Show the predictive framework you designed.

Hint: What features do you extract? What algorithms do you use in the framework?

The Predictive Framework

Here I will build a predictive framework for predicting whether each flight in 2005 will be delayed or not by using the data for 2003 and 2004 as training datasets.

In this framework, I utilize Spark's Pipelines and PipelineModels, which help to ensure that training and test data go through identical feature processing steps.

The overview of our training process is shown below.

Step1: Define Y

Our Target is " Each flight in 2005 will delay or not.", so I do the calculation to define another column 'delay_DepDelay', '0' means no delay. '1' means delay. Then, I set Y as 'delay_DepDelay'.

Y = 'delay_DepDelay'

```
# 0 = no delay
# +1 = delay

data_1 = data_1.withColumn("delay_ArrDelay", \
          when((data_1.ArrDelay < 0), lit(0)) \
            .otherwise(lit(1)))
data_1 = data_1.withColumn("delay_DepDelay", \
          when((data_1.DepDelay < 0), lit(0)) \
            .otherwise(lit(1)))
data_1.show()
```

Step2: Define features

We have 29 columns in our data. We divided features into numerical features and categorical features.

- **Numerical features** = ['Month', 'DayofMonth', 'DayOfWeek', 'CRSDepTime', 'CRSArrTime', 'CRSElapsedTime','Distance', 'TaxiIn', 'TaxiOut', 'Cancelled']
- **Categorical features** = ['UniqueCarrier', 'FlightNum', 'TailNum', 'Origin', 'Dest']

Step3: Data Pre-processing

I use two approaches for the Data Pre-processing

Data Pre-processing Approach 1

Check the data drop the NA value and change the type of numerical features to float.

```
def numTOdouble(df, Cols):
  for col in Cols:
    df = df.withColumn(col, df[col].cast('double'))
  df.show(10)
  return df
```

```
data2 = numTOdouble(data2,Cols)
data2.dtypes
```

```
data2 = data2.dropna()
```

```
from pyspark.ml.feature import VectorAssembler
```

```
assembler=VectorAssembler().setInputCols(InputCols)\
            .setOutputCol("features_vec")
data_3=assembler.transform(data_2)
data_3.show()
```

Data Pre-processing Approach 2

Check the column type and use the mean of the column to fill the missing value or drop NA value for both numerical features and categorical features.

```
# Impute numerical features
for col in num_cols:
    df = df.withColumn(col, df[col].cast('double'))
    mu = df.select(col).agg({col:'mean'}).collect()[0][0]
    df = df.withColumn(col, F.when(df[col].isNull(), mu)\
                       .otherwise(df[col]))
df = df.withColumn('label', df[target_col].cast('double'))
df = df.filter(df['label'].isNotNull())

# Impute categorical features
for col in cate_cols:
    frq = df.select(col).groupby(col).count().orderBy('count', ascending=False).limit(1).collect()[0][0]
    df = df.withColumn(col, F.when((df[col].isNull() | (df[col] == '')), frq).otherwise(df[col]))

# Assure there is no missing values
for col in num_cols + cate_cols + ['label']:
    assert df.filter(df[col].isNull()).count() == 0, "Column '{}' exists NULL value(s)".format(col)
    assert df.filter(df[col] == '').count() == 0, "Column '{}' exists empty string(s)".format(col)
```

Step4: Data Pre-processing (feature extractor/transformer)

Use the different approaches for categorical features, like One-hot encoding for categorical features

Do the Pre-processing of both numerical attributes and categorical attributes. The following feature extraction/transformation processes are used successively.

1. **StringIndexer:** StringIndexer encodes a string column of labels to a column of label indices.
2. **OneHotEncoder:** One-hot encoding maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms that expect continuous features, such as Logistic Regression, to use categorical features.
3. **VectorAssembler:** VectorAssembler is a transformer combines a given list of columns into a single vector column.
4. **StandardScaler:** StandardScaler transforms a dataset of Vector rows, normalizing each feature to have unit standard deviation and/or zero mean. It takes parameters:
    - WithStd: True by default. Scales the data to unit standard deviation.
    - WithMean: False by default. Centers the data with the mean before scaling. It will build a dense output, so this does not work on sparse input and will raise an exception.

After defining these feature extractors/transformers, we create a PipelineModel by concatenating them and applying it to the training data to extract features.

```
# String Indexing for categorical features
indexers = [StringIndexer(inputCol=col,
                          outputCol="{}_idx".format(col)) \
                          for col in cate_cols]

# One-hot encoding for categorical features
encoders = [OneHotEncoder(inputCol="{}_idx".format(col),
                          outputCol="{}_oh".format(col)) \
                          for col in cate_cols]

# Concat Feature Columns
assembler = VectorAssembler(inputCols = num_cols + \
                        ["{}_oh".format(col) for col in cate_cols],
                        outputCol = "_features")

# Standardize Features
scaler = StandardScaler(inputCol='_features',
                        outputCol='features',
                        withStd=True, withMean=False)

preprocessor = Pipeline(stages = indexers + encoders + \
                                 [assembler, scaler]).fit(df)
```

.

Use the pipeline to make features into vector

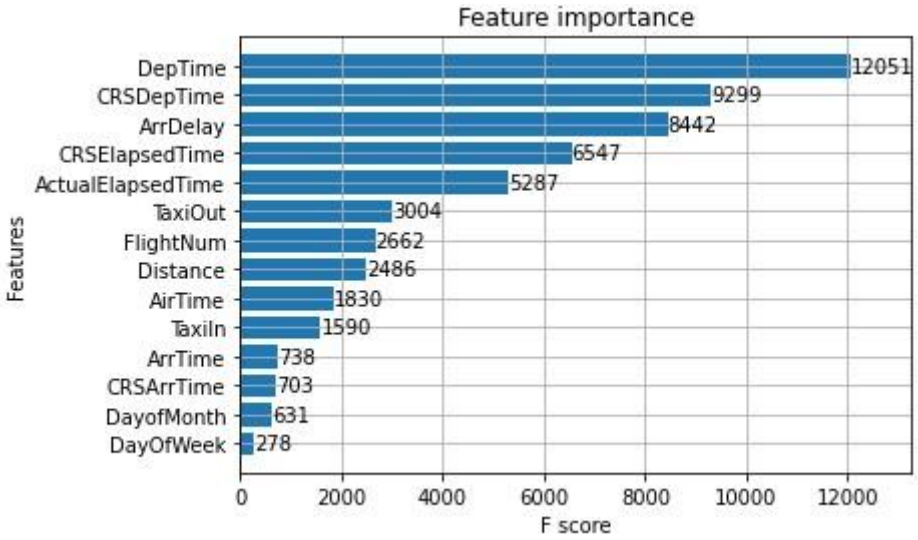
```
# Pre-Process
preprocessor = gen_preprocessor(df)
df1 = preprocessor.transform(df)
```

```
df1.show(10)
```

```
-------------------+----------------+----------------+--------------------+--------------------+
         TailNum_oh|       Origin_oh|         Dest_oh|           _features|            features|
-------------------+----------------+----------------+--------------------+--------------------+
(6163,[1853],[1.0])| (294,[1],[1.0])|(298,[40],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[1961],[1.0])| (294,[1],[1.0])|(298,[40],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[4456],[1.0])| (294,[1],[1.0])|(298,[40],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[3748],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[1755],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[3970],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[2878],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[2566],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[1933],[1.0])|(294,[29],[1.0])| (298,[1],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
(6163,[3411],[1.0])|(294,[16],[1.0])|(298,[14],[1.0])|(14760,[0,1,2,3,4...|(14760,[0,1,2,3,4...|
-------------------+----------------+----------------+--------------------+--------------------+
```

Step5: Feature importance

I use the XGboost model to find out the Feature importance of each numerical feature to see if I choose the suitable feature or not.



○ Q2: Explain the validation method you use.

Hint: Leave-one-out, Holdout, k-fold, or other methods?

I use the **Logistic Regression Model** and **Random Forest Classifier Model** to train our data. I also use k-fold cross-validation to help select the best parameters and evaluate the model's performance we trained.

Model 1: Logistic Regression Model

The first model we use in our predictive framework is the Logistic Regression Classifier, which is widely used to predict a binary response.

```
1 from pyspark.ml.classification import LogisticRegression
2 from pyspark.ml.evaluation import BinaryClassificationEvaluator
3 from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
4 import numpy as np
5 lr = LogisticRegression(featuresCol = 'features', labelCol='delay_DepDelay')
```

The logistic model is a statistical model with input (independent variable) a continuous variable and output (dependent variable) a binary variable. A unit change in the input multiplies the odds of the two possible outputs by a constant factor. For binary classification problems, the algorithm outputs a binary logistic regression model.

In spark.ml, two algorithms have been implemented to solve logistic regression: mini-batch gradient descent and L-BFGS. L-BFGS is used in our predictive framework for faster convergence.

```
paramGrid_lr = ParamGridBuilder() \
    .addGrid(lr.regParam, np.linspace(0.3, 0.01, 10)) \
    .addGrid(lr.elasticNetParam, np.linspace(0.3, 0.8, 6)) \
    .build()
```

Besides the fact that we have decided on the model to be used, we also need to find its best parameters for a given task.

We tackle this tuning task using CrossValidator, which takes an Estimator (i.e., logistic regression in this case), a set of ParamMaps (i.e., regularization parameter (>= 0) of the logistic regression model in this case), and an Evaluator (i.e. area under the precision-recall curve in this case).

```
crossval_lr = CrossValidator(estimator=lr,
                             estimatorParamMaps=paramGrid_lr,
                             evaluator=BinaryClassificationEvaluator(),
                             numFolds= 3)
cvModel_lr = crossval_lr.fit(train_df)
```

CrossValidator begins by splitting the dataset into a set of folds which are used as separate training and test datasets. For example, with k=3 folds, CrossValidator will generate 3 (training, test) dataset pairs, each of which uses 2/3 of the data for training and 1/3 for testing. For each ParamMap, CrossValidator trains the given Estimator and evaluates it using the given Evaluator. Note that we use k=10 in our predictive framework (10-fold cross-validation)

Model 2: Random Forest Classifier Model

In our predictive framework, the second model we use is the **Random Forest Classifier**, which we can compare with the first model.

```
# from pyspark.ml.regression import RandomForestRegressor
leb = 'delay_DepDelay'
rf = RandomForestClassifier(featuresCol = 'features', labelCol=leb, maxDepth=30)
```

```
rf_model = rf.fit(train_df)
```

We did not use CrossValidator this time because CrossValidator in the Random Forest Classifier Model will need more memory to run. If I want to keep the same features to put into Random Forest Classifier Model, we need to increase our memory or reduce our features.

```
rf_predictions = rf_model.transform(test_df)
rf_predictions.show()
```

```
+--------------------+--------------------+----------+
|       rawPrediction|         probability|prediction|
+--------------------+--------------------+----------+
|[10.2939272332340...|[0.51469636166170...|       0.0|
|[10.1559100711740...|[0.50779550355870...|       0.0|
|[10.1559100711740...|[0.50779550355870...|       0.0|
|[10.1559100711740...|[0.50779550355870...|       0.0|
|[9.85627391946632...|[0.49281369597331...|       1.0|
|[9.71023077254411...|[0.48551153862720...|       1.0|
```

We use two evaluators for the evaluation: Binary-ClassificationEvaluator and Multiclass-Classification Evaluator.

```
# Area under the curve for the training data
from pyspark.ml.evaluation import BinaryClassificationEvaluator
rf_evaluator = BinaryClassificationEvaluator(rawPredictionCol='prediction',
                                             labelCol='delay_DepDelay', metricName='areaUnderROC')
print('Test Area Under ROC', rf_evaluator.evaluate(rf_predictions))
```

```
Test Area Under ROC 0.6358156248580472
```

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
rf_evaluator_mul = MulticlassClassificationEvaluator(labelCol="delay_DepDelay",
                                                     predictionCol="prediction", metricName="accuracy")

rf_accuracy = rf_evaluator_mul.evaluate(rf_predictions)
print("Test accuracy = %g" % (rf_accuracy))
print("Test Error = %g" % (1.0 - rf_accuracy))
```

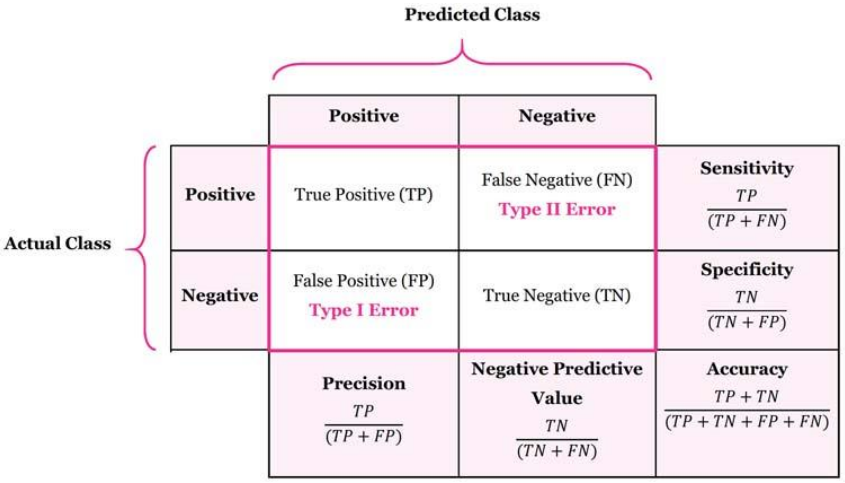
```
Test accuracy = 0.644934
Test Error = 0.355066
```

○ Q3: Explain the evaluation metric you use.

Hint: Don't just show the prediction results, you should show the effectiveness of your framework (e.g., using a confusion matrix).

I will evaluate the training and testing to see how well the are doing in the two different models.

We use Area under ROC, Confusion matrix, Area under Precision-Recall curve, f score…etc to evaluate our results. Use the confusion matrix (like the figure below) to see the predictions and actual value clearly. We also can find out the true positive, true negative, false positive, and false negative. To better understand our results and find out the precision and recall.



$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Evaluation Model 1: Logistic Regression Model

Evaluate the Model Trained with 10-fold Cross-Validation

After the Logistic Regression Classifier is trained and the best parameters are chosen, we can evaluate the performance of this model on the training data.

The evaluation metrics we use are

Evaluation for Train

- Area under ROC (receiver operating characteristic curve)
- Area under Precision-Recall curve
- Beta Coefficients

Evaluation for Test

- Test accuracy and Test Error
- Confusion matrix
- Area under ROC (receiver operating characteristic curve)
- f-MeasureByThreshold
- PrecisionByThreshold
- RecallByThreshold

Evaluation Model 2: Random Forest Classifier

Evaluate the Model Trained with 10-fold Cross-Validation

After the Random Forest Classifier is trained and the best parameters are chosen, we can evaluate the performance of this model on the training data.

The evaluation metrics we use are

Evaluation for Train

- Area under ROC (receiver operating characteristic curve)
- Area under Precision-Recall curve

Evaluation for Test

- Test accuracy and Test Error
- Confusion matrix
- Area under ROC (receiver operating characteristic curve)
- f-MeasureByThreshold
- PrecisionByThreshold
- RecallByThreshold

○ Q4: Show the validation results and give a summary of the results.

Validation results summary

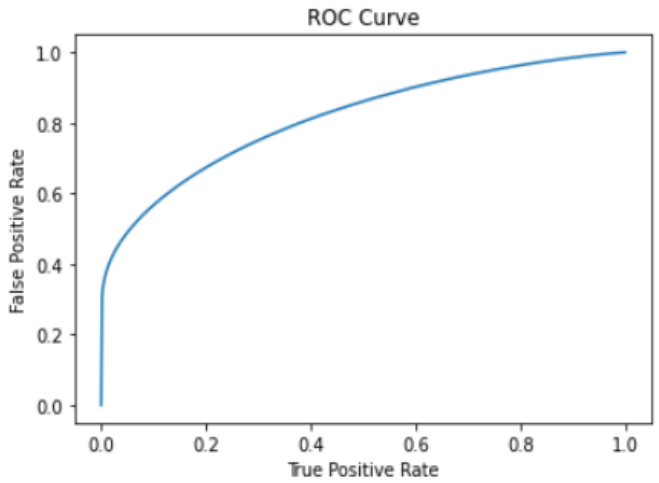
Model	Model 1: Logistic Regression Model	Model 2: Random Forest Classifier
Training Result		
Train Area under ROC	0.815208	0.780468
Testing Result		
Test Accuracy	0.667228	0.644934
Test Error	0.322772	0.355066
Test Area under ROC	0.673276	0.365815
Test Precision	0.641483	0.617303
Test Recall	0.635009	0.547518

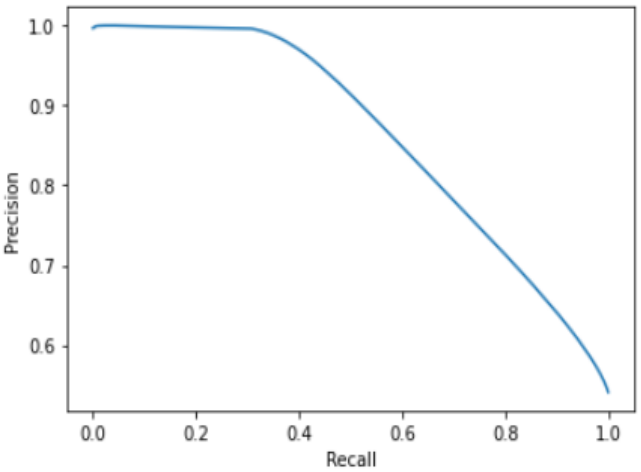
Note: Random Forest Classifier did not have a better result because the model only can maxdepth = 30. This is the reason that Random Forest Classifier cannot make the best outcome. We can fix that by adding cross-validation to the model.

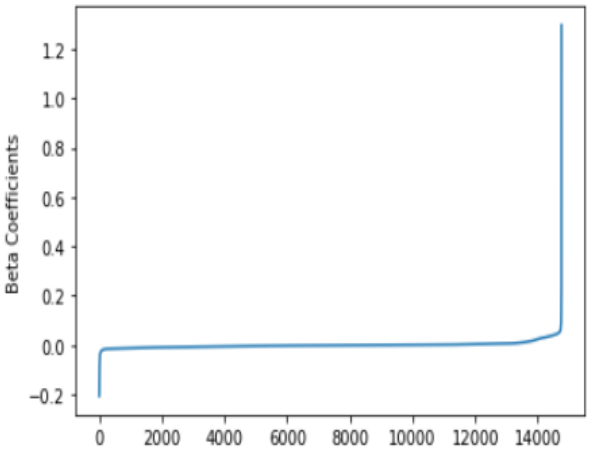
Evaluation Model 1: Logistic Regression Model

Evaluation for Train

```
Training set areaUnderROC: 0.8152082347167017
```







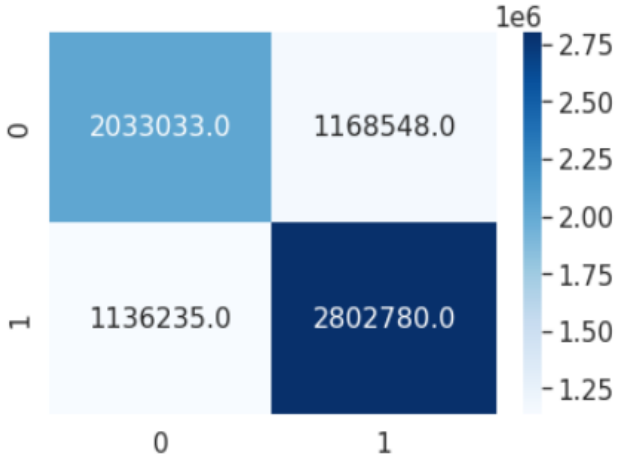
Evaluation for Test

```
5 print("Test accuracy = %g" % (lr_accuracy))
6 print("Test Error = %g" % (1.0 - lr_accuracy))
```

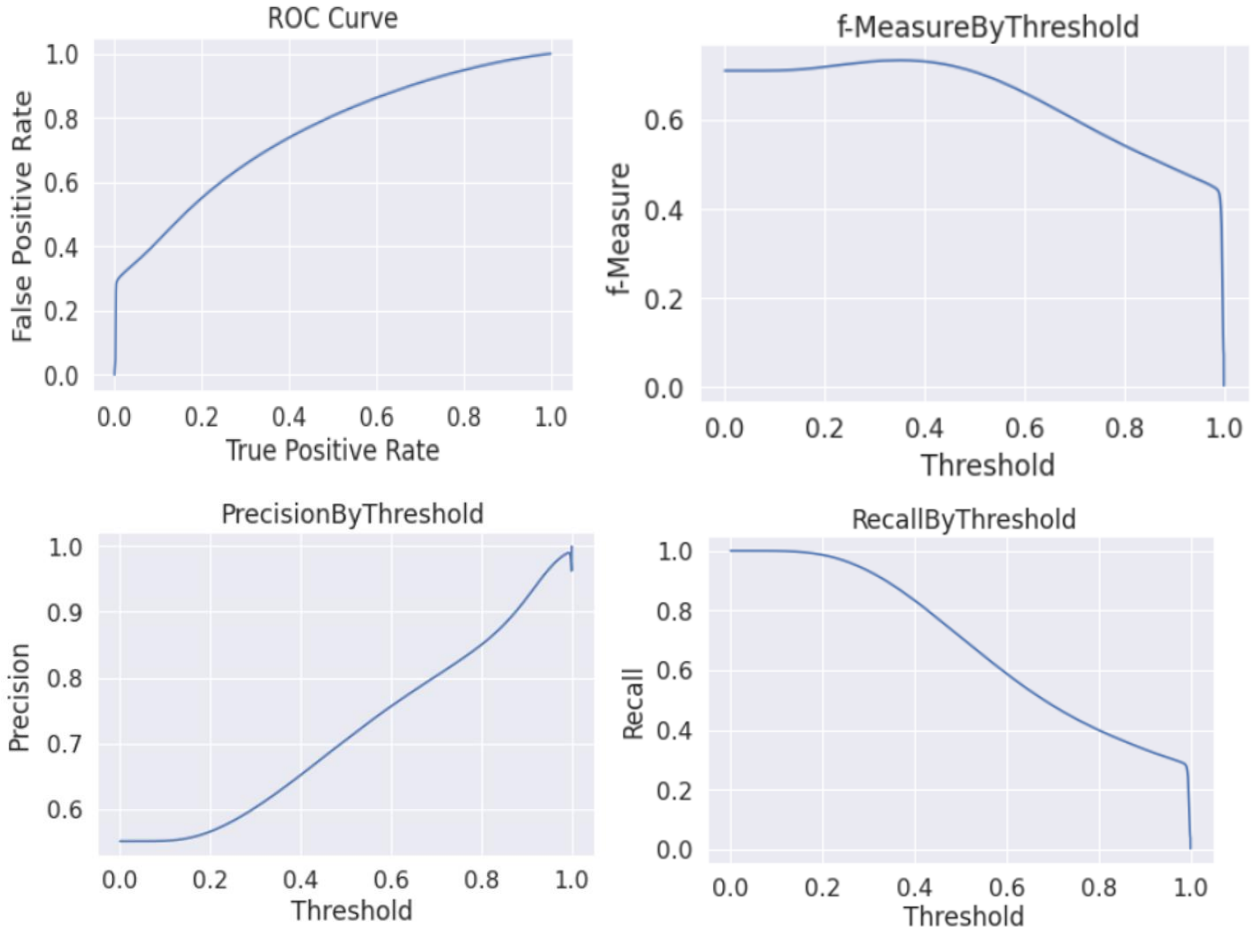
```
Test accuracy = 0.677228
Test Error = 0.322772
```

Pyspark ConfusionMatrix

```
[[2033033. 1168548.]
 [1136235. 2802780.]]
```



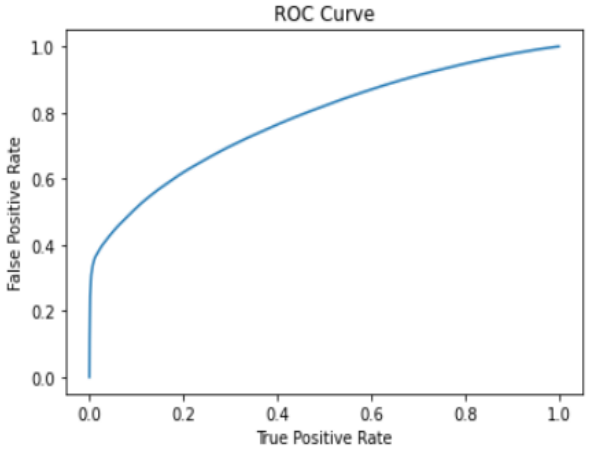

```
Test Area Under ROC 0.6732762232013803
```

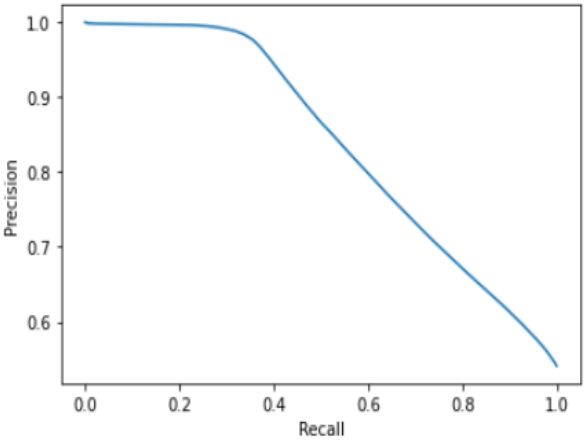


Evaluation Model 2: Random Forest Classifier

<u>Evaluation for Train</u>

```
Training set areaUnderROC: 0.7804683443604191
```



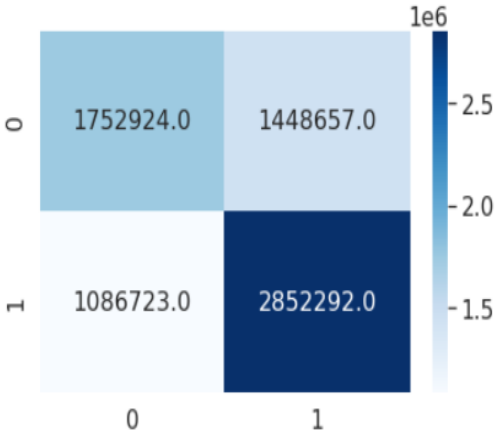


Evaluation for Test

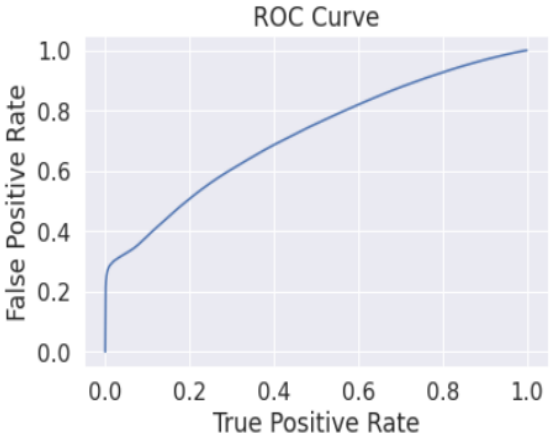
Pyspark ConfusionMatrix

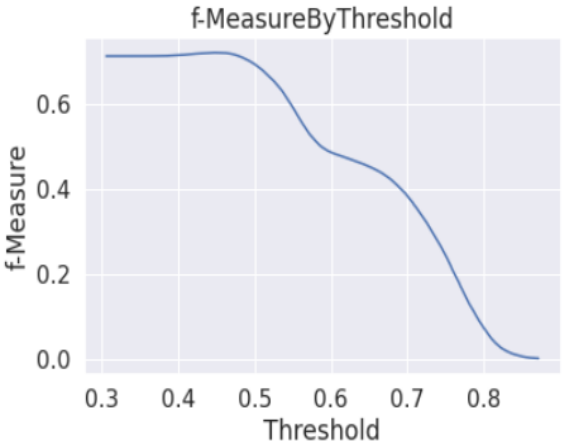
```
Test accuracy = 0.644934
Test Error = 0.355066
```

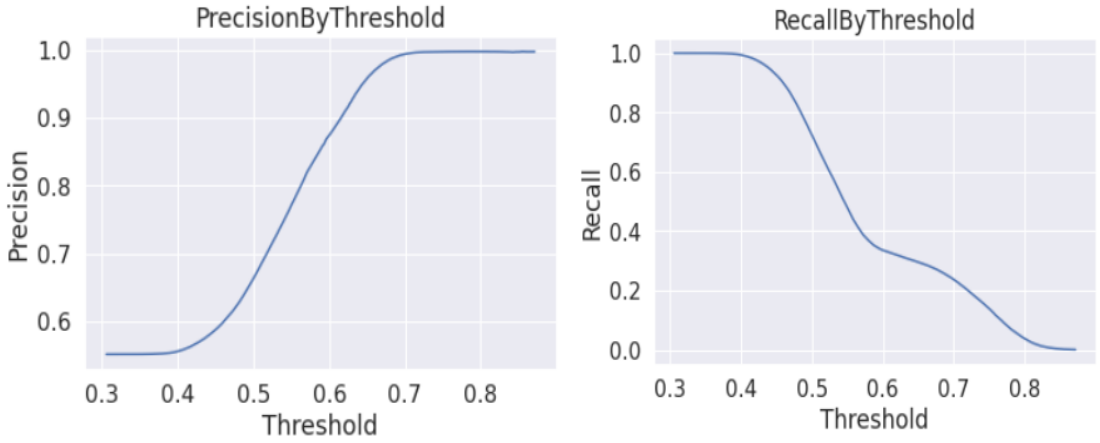
```
[[1752924. 1448657.]
 [1086723. 2852292.]]
```




```
Test Area Under ROC 0.6358156248580472
```







Anything else worth mentioning

- **Understanding the difference between the ml package and Spark mllib package**

  The first difficulty that I face is understanding the difference between the ml package and the Spark mllib package. Spark mllib is RDD-based and with different classier machine learning models (Model 1: Logistic Regression Model, Model 2: Random Forest Classifier…etc.). We need to do the transformer to deal with our features first in order to fit in the Spark milib models.

- **Diffident validation model has different problems (TrainValidationSplit, Cross-validation)**

  With different classier machine learning model (Model 1: Logistic Regression Model, Model 2: Random Forest Classifier…etc), we need to do the transformer to deal with our features first in order to fit in the Spark milib models. They also have different parameter need to be set and limit that we need to work on.

- **Out of memory Error**

```
lr_model = lrv.fit(train_df)
```

```
Py4JJavaError: An error occurred while calling o1635.fit.
: org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 171.0 faile
741) (29e8d6c7352d executor driver): java.lang.OutOfMemoryError: Java heap space

Driver stacktrace:
```

  Even with Spark mllib models, there are still some Out of memory Errors that we need to solve. We need to reduce features or make some adjustments to the model so we can successfully run the model with our training data.