Assignment 2 – Spark

Amit Kumar Gangwar - D20123666 – TU059 - 1st year MS Data Science - Programming for Big Data

Problem Statement

The dataset used for this competition is synthetic but based on a real dataset (in this case, the actual Titanic data!) and generated using a CTGAN. Task is to predict whether or not a passenger survived the sinking of the titanic (**a synthetic, much larger dataset based on the actual Titanic dataset**). For each PasengerId row in the test set, you must predict a 0 or 1 value for the Survived target.

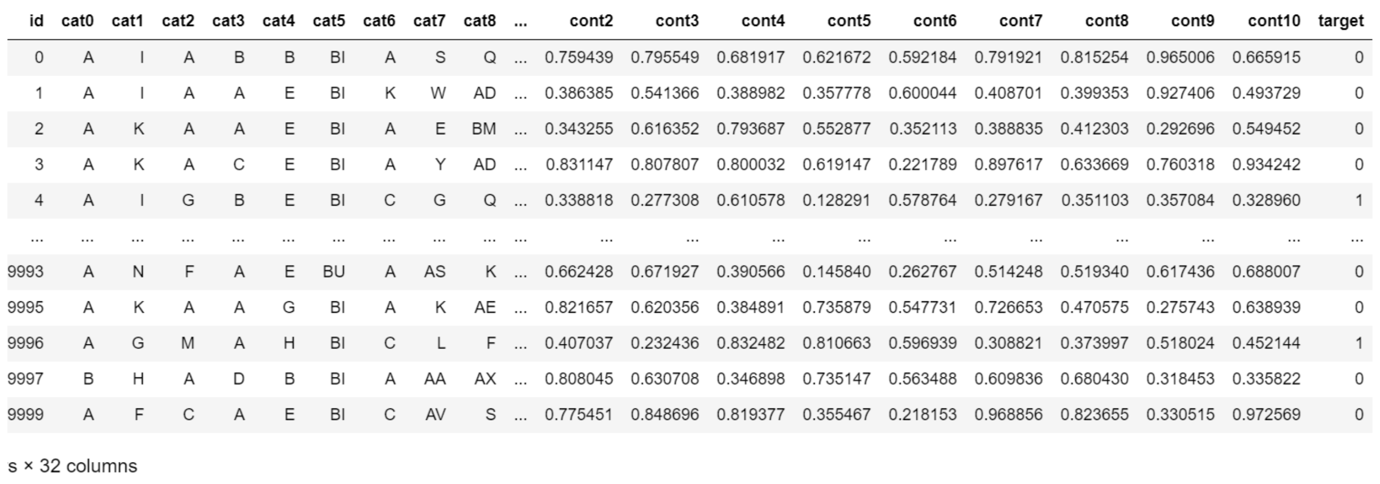
Link to the competition on Kaggle: https://www.kaggle.com/c/tabular-playground-series-apr-2021

Data Preparation

About Data

The dataset has 300000 rows and 31 columns in total excluding the 'id' (index) column. 19 categorical (nominal) columns, 11 numerical columns and 1 target column which is binary (0 for no and 1 for yes).

Numerical Features	cont0, cont1, cont2, cont3, cont4, cont5, cont6, cont7, cont8, cont9, cont10, target
Categorical columns	cat0, cat1, cat2, cat3, cat4, cat5, cat6, cat7, cat8, cat9, cat10, cat11, cat12, cat13, cat14, cat15, cat16, cat17, cat18



id	cat0	cat1	cat2	cat3	cat4	cat5	cat6	cat7	cat8	...	cont2	cont3	cont4	cont5	cont6	cont7	cont8	cont9	cont10	target
0	A	I	A	B	B	BI	A	S	Q	...	0.759439	0.795549	0.681917	0.621672	0.592184	0.791921	0.815254	0.965006	0.665915	0
1	A	I	A	A	E	BI	K	W	AD	...	0.386385	0.541366	0.388982	0.357778	0.600044	0.408701	0.399353	0.927406	0.493729	0
2	A	K	A	A	E	BI	A	E	BM	...	0.343255	0.616352	0.793687	0.552877	0.352113	0.388835	0.412303	0.292696	0.549452	0
3	A	K	A	C	E	BI	A	Y	AD	...	0.831147	0.807807	0.800032	0.619147	0.221789	0.897617	0.633669	0.760318	0.934242	0
4	A	I	G	B	E	BI	C	G	Q	...	0.338818	0.277308	0.610578	0.128291	0.578764	0.279167	0.351103	0.357084	0.328960	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9993	A	N	F	A	E	BU	A	AS	K	...	0.662428	0.671927	0.390566	0.145840	0.262767	0.514248	0.519340	0.617436	0.688007	0
9995	A	K	A	A	G	BI	A	K	AE	...	0.821657	0.620356	0.384891	0.735879	0.547731	0.726653	0.470575	0.275743	0.638939	0
9996	A	G	M	A	H	BI	C	L	F	...	0.407037	0.232436	0.832482	0.810663	0.596939	0.308821	0.373997	0.518024	0.452144	1
9997	B	H	A	D	B	BI	A	AA	AX	...	0.808045	0.630708	0.346898	0.735147	0.563488	0.609836	0.680430	0.318453	0.335822	0
9999	A	F	C	A	E	BI	C	AV	S	...	0.775451	0.848696	0.819377	0.355467	0.218153	0.968856	0.823655	0.330515	0.972569	0

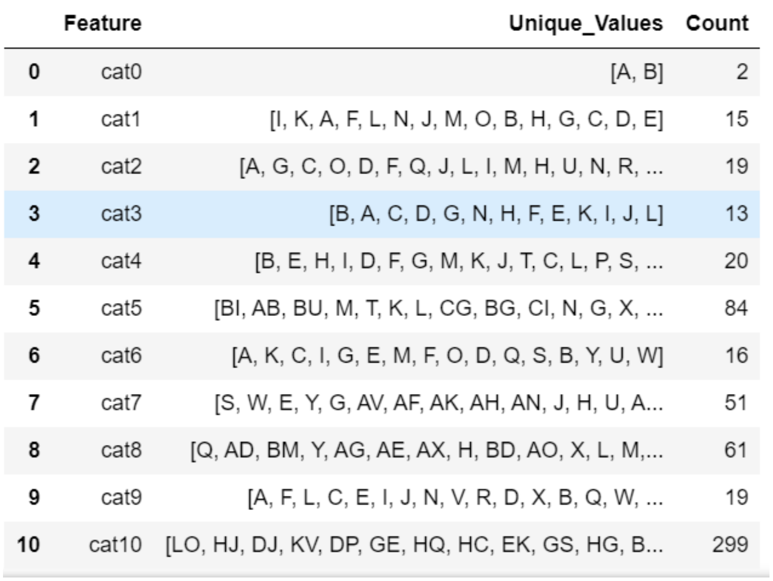
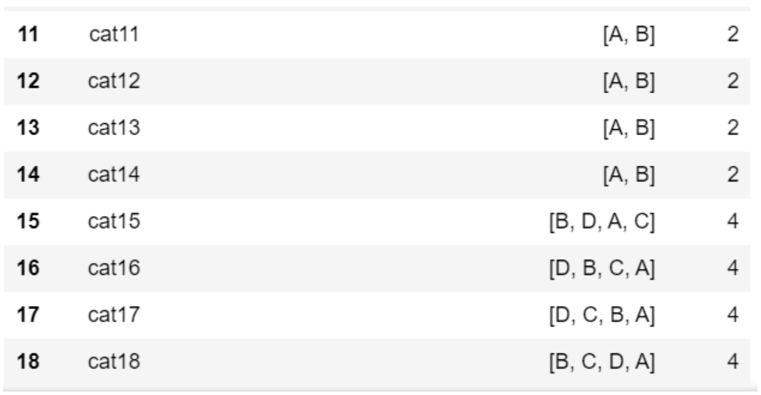
s × 32 columns

Summary statistics

Summary of Categorical Variables:

The various categorical columns have various number of distinct values from 'A' to 'Z' and their combinations.

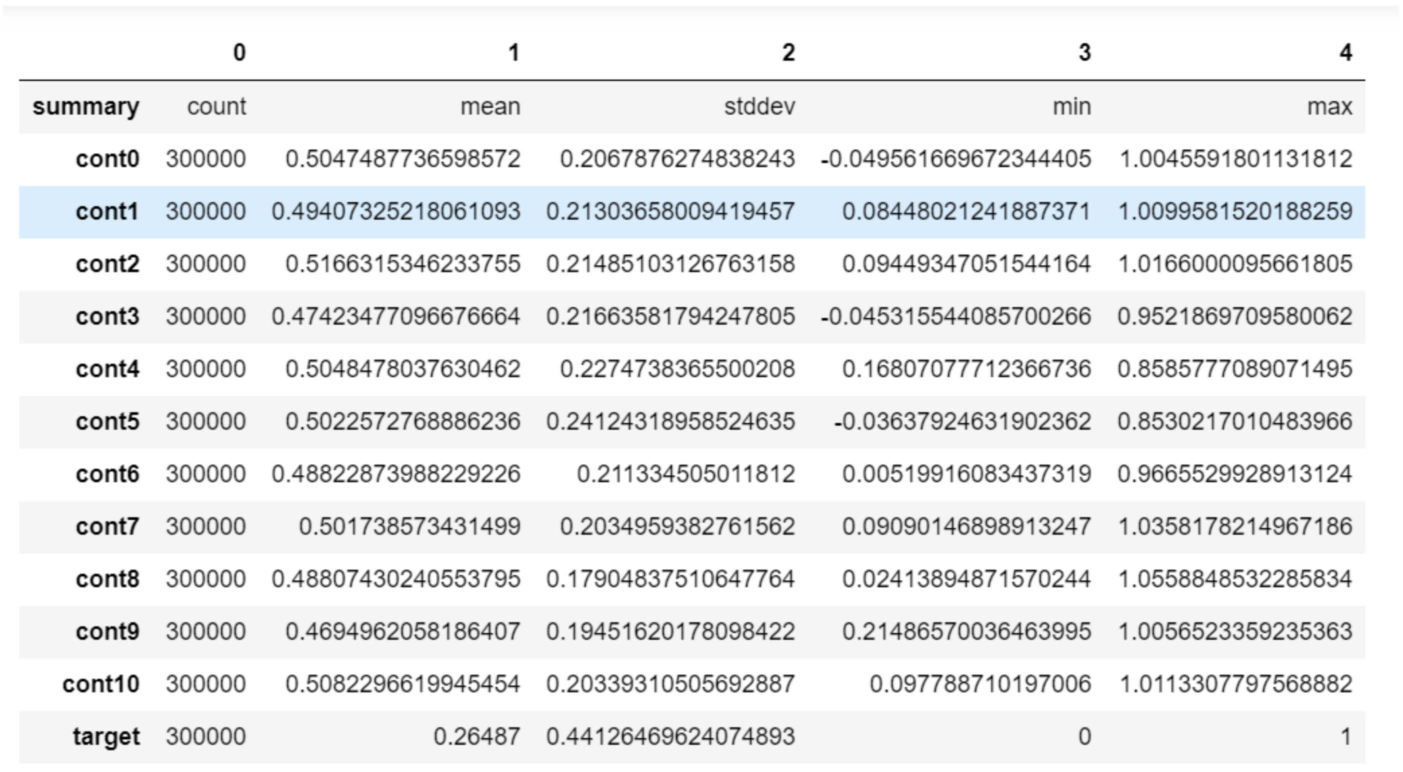
```
#summary of categorical features
d = []
df1=df[cat].toPandas()
for x in cat:
    d.append((x, df1[x].unique(), df1[x].nunique()))

df_cat=pd.DataFrame(d, columns=("Feature","Unique_Values","Count"))
df_cat
```

	Feature	Unique_Values	Count
0	cat0	[A, B]	2
1	cat1	[I, K, A, F, L, N, J, M, O, B, H, G, C, D, E]	15
2	cat2	[A, G, C, O, D, F, Q, J, L, I, M, H, U, N, R, ...	19
3	cat3	[B, A, C, D, G, N, H, F, E, K, I, J, L]	13
4	cat4	[B, E, H, I, D, F, G, M, K, J, T, C, L, P, S, ...	20
5	cat5	[BI, AB, BU, M, T, K, L, CG, BG, CI, N, G, X, ...	84
6	cat6	[A, K, C, I, G, E, M, F, O, D, Q, S, B, Y, U, W]	16
7	cat7	[S, W, E, Y, G, AV, AF, AK, AH, AN, J, H, U, A...	51
8	cat8	[Q, AD, BM, Y, AG, AE, AX, H, BD, AO, X, L, M,...	61
9	cat9	[A, F, L, C, E, I, J, N, V, R, D, X, B, Q, W, ...	19
10	cat10	[LO, HJ, DJ, KV, DP, GE, HQ, HC, EK, GS, HG, B...	299
11	cat11	[A, B]	2
12	cat12	[A, B]	2
13	cat13	[A, B]	2
14	cat14	[A, B]	2
15	cat15	[B, D, A, C]	4
16	cat16	[D, B, C, A]	4
17	cat17	[D, C, B, A]	4
18	cat18	[B, C, D, A]	4

Summary of numerical features.

```
df.select(num).describe().toPandas().transpose()
```



	0	1	2	3	4
summary	count	mean	stddev	min	max
cont0	300000	0.5047487736598572	0.2067876274838243	-0.049561669672344405	1.0045591801131812
cont1	300000	0.49407325218061093	0.21303658009419457	0.08448021241887371	1.0099581520188259
cont2	300000	0.5166315346233755	0.21485103126763158	0.09449347051544164	1.0166000095661805
cont3	300000	0.47423477096676664	0.21663581794247805	-0.045315544085700266	0.9521869709580062
cont4	300000	0.5048478037630462	0.2274738365500208	0.16807077712366736	0.8585777089071495
cont5	300000	0.5022572768886236	0.24124318958524635	-0.03637924631902362	0.8530217010483966
cont6	300000	0.48822873988229226	0.211334505011812	0.00519916083437319	0.9665529928913124
cont7	300000	0.501738573431499	0.2034959382761562	0.09090146898913247	1.0358178214967186
cont8	300000	0.48807430240553795	0.17904837510647764	0.02413894871570244	1.0558848532285834
cont9	300000	0.4694962058186407	0.19451620178098422	0.21486570036463995	1.0056523359235363
cont10	300000	0.5082296619945454	0.20339310505692887	0.097788710197006	1.0113307797568882
target	300000	0.26487	0.44126469624074893	0	1

Data Cleaning:

Checking for missing values:

```
#checking null and NaN values in train data
from pyspark.sql.functions import isnan, when, count, col
df3=df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).toPandas()
df3
```

	cat0	cat1	cat2	cat3	cat4	cat5	cat6	cat7	cat8	cat9	...	cont2	cont3	cont4	cont5	cont6	cont7	cont8	cont9	cont10	target
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

1 rows × 31 columns

We can see in the above output screenshot, there are no missing values.

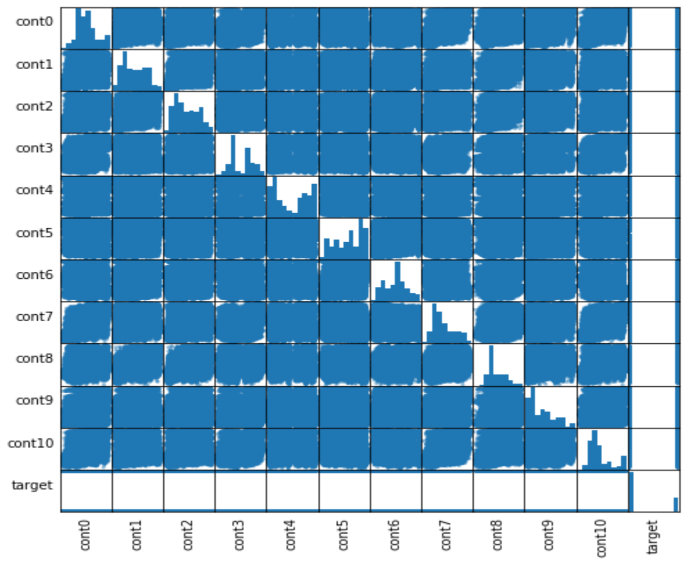
Dropping Duplicate rows:

```
#df=df.dropna()
df=df.dropDuplicates()
```

Correlation Between Numerical Features

Finding Correlation between numerical features will help us choose right machine learning algorithm. If features are highly correlated, that means they are probably not mutually independent. In that case, we cannot use Naïve Bayes algorithm because it requires conditional independence data.

```
from pandas.plotting import scatter_matrix
import pandas as pd
df2=df[num]
numeric_data = df2.select(num).toPandas()
axs = pd.plotting.scatter_matrix(numeric_data, figsize=(8, 8));
n = len(numeric_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```



It can be seen in the figure above that the numerical features are highly correlated.

Test/Train Split:

Splitting Data into test (30%) and train (70%). Since the size of data is huge, the is an ideal split ratio.

```
train, test = df.randomSplit([0.7, 0.3], seed=7)
print(f"Train set length: {train.count()} records")
print(f"Test set length: {test.count()} records")
```

```
Train set length: 209968 records
Test set length: 90032 records
```

Implementation

The machine learning models in pyspark do not accept multiple input features. They accept only one input feature in the form of a vector. To convert all the input features in one vector, categorical features need to be converted to numerical by using appropriate technique, they are as follows.

String Indexing

Stringindexing actually transform categorical data into numeric but with order. Therefore, after stringindexing we do one-hot encoding because our features are not ordinal, they are just nominal data.

```
#separating categorical and numerical features
cat_train = [x for (x, dataType) in train.dtypes if dataType == "string"]
num_train = [x for (x, dataType) in train.dtypes if ((dataType == "double") & (x
!= "target"))]
#importing OneHotEncoder and StringIndexer
from pyspark.ml.feature import (OneHotEncoder,StringIndexer,)
#creating string indexer for categorical columns
string_indexer = [StringIndexer(inputCol=x, outputCol=x + "_StringIndexer",
handleInvalid="skip") for x in cat_train]
```

OneHot Encoding

One-hot encoding maps a categorical feature, represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values. OneHotEncoder can transform multiple columns, returning a one-hot-encoded output vector column for each input column. It is common to merge these vectors into a single feature vector using Vector Assembler.

```
#creating one-hot encoder for string indexed columns
one_hot_encoder = [
    OneHotEncoder(
        inputCols=[f"{x}_StringIndexer" for x in cat_train],
        outputCols=[f"{x}_OneHotEncoder" for x in cat_train])]
```

Vector Assembling

VectorAssembler is a transformer that combines a given list of columns into a single vector column. VectorAssembler accepts the following input column types: all numeric types, boolean type, and vector type. In each row, the values of the input columns will be concatenated into a vector in the specified order.

```
#creating vector assembler
from pyspark.ml.feature import VectorAssembler
assemblerInput = [x for x in num_train]
assemblerInput += [f"{x}_OneHotEncoder" for x in cat_train]
vector_assembler = VectorAssembler(inputCols=assemblerInput,
outputCol="VectorAssembler_features")
```

Normalisation

The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. Normalization is mandatory for some algorithms like KNN but is recommended for other algorithms also as it helps in improving model performance.

```
from pyspark.ml.feature import Normalizer
#creating normalizer
normalizer = Normalizer(inputCol="VectorAssembler_features", 
outputCol="norm_VectorAssembler_features", p=1.0)
```

Creating Pipeline

A machine learning project has a lot of moving components that need to be tied together before we can successfully execute it. A pipeline allows us to maintain the data flow of all the relevant transformations that are required to reach the end result. We need to define the stages of the pipeline which act as a chain of command for Spark to run. Here, each stage is a Transformer. In the below code we are creating pipeline and transforming test and train data using pipeline. The transformed data will have all the features (original and transformed), we need to select only two for model building i.e., norm_VectorAssembler_features (the normalized vector feature), target (the target feature).

```
#Creating pipeline
#setting stages for Data preprocessing pipeline
stages = []
stages += string_indexer
stages += one_hot_encoder
stages += [vector_assembler]
stages += [normalizer]

#adding stages to pipeline
 #%%time
 from pyspark.ml import Pipeline
 pipeline = Pipeline().setStages(stages)
 model = pipeline.fit(train)

 #transforming test and train data using Data preprocessing pipeline
 train_df = model.transform(train)
 test_df = model.transform(test)

 #selecting input (norm_VectorAssembler_features) and target (target) feature 
from the transformed data
 test_data = test_df.select(
     F.col("norm_VectorAssembler_features").alias("features"),
     F.col("target").alias("label"),)

 train_data = train_df.select(
     F.col("norm_VectorAssembler_features").alias("features"),
     F.col("target").alias("label"),)

```

Model Building

Since there are plenty of input features, decision tree classifiers could lead to overfitting and is not an ideal choice. Also, the input features are highly correlated, so Naïve Bayes should not be used. KNN algorithm is not available in the Pyspark machine learning libraries. So, we will be building Logistic regression, Linear SVM and Multilayer perceptron classifier models. Later, on the basis of performance of these models, best will be picked.

1. **Logistics Regression**
   We will be building the logistic regression model using K-Fold cross validation technique with a range of parameter given in the parameter grid and would pick the best model among all. The reason behind choosing a particular hyperparameter is commented in the code.

```python
#K-FOLD CROSS VALIDATION
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

lr = LogisticRegression(maxIter=10)

#elasticNetParam is regularization parameter, to prevent overfitting
#fitIntercept is for weather we want to fit intercept for logistic or not
 paramGrid = ParamGridBuilder() \
     .addGrid(lr.regParam, [0.1, 0.01]) \
     .addGrid(lr.fitIntercept, [False, True]) \
     .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
     .build()

 crossval = CrossValidator(estimator=lr,
                           estimatorParamMaps=paramGrid,
                           evaluator=BinaryClassificationEvaluator(),
                           numFolds=3)  # use 3+ folds in practice

 # Run cross-validation, and choose the best set of parameters.
 cvModel = crossval.fit(train_data)

```

2. **Support Vector Machine Model**
Like Logistic Regression model, again we will be building the SVM model using K-Fold cross validation technique with a range of parameter given in the parameter grid and would pick the best model among all. The reason behind choosing a particular hyperparameter is commented in the code.

```python
from pyspark.ml.classification import LinearSVC
lsvc = LinearSVC(maxIter=10)

#elasticNetParam is regularization parameter, to prevent overfitting
#fitIntercept is for weather we want to fit intercept for logistic or not
paramGrid = ParamGridBuilder() \
    .addGrid(lsvc.regParam, [0.1, 0.01]) \
    .addGrid(lsvc.fitIntercept, [False, True]) \
    .build()

 crossval = CrossValidator(estimator=lsvc,
                           estimatorParamMaps=paramGrid,
                           evaluator=BinaryClassificationEvaluator(),
                           numFolds=3)  # use 3+ folds in practice

 # Run cross-validation, and choose the best set of parameters.
 lsvc_cvModel = crossval.fit(train_data)
```

3. **Multilayer perceptron classifier**
Multilayer perceptron classifier (MLPC) is a classifier based on the feedforward artificial neural network.
Parameter:
BlockSize: it is the number of inputs to be included during each iteration. The default value is 128. Smaller blockSize improves accuracy at the expense of prolonged learning time and vice versa.
Seed: This parameter ensures that the same random values are generated each time the algorithm is tested.
Layers: This parameter accepts a list of integers representing the input, hidden, and output layer. The format is as follows [a, b, c, d] ‘a’ representing the number of input variables, ‘b’ and ‘c’ representing the hidden layers and ‘d’ representing the output layers (number of classes in target variable, in our case its 2). More hidden layers will make the neural network more complex but do not ensure more accuracy so we have kept it less.
maxIter: maxIter are maximum number of iterations, describes the number of times a batch of data passed through the algorithm. Since batch size is huge, we are keeping it less.

```python
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
#x is number of input variables
x=train_data.schema["features"].metadata["ml_attr"]["num_attrs"]

layers = [x, 5, 4, 2]

# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128, seed=1234)

# train the model
model = trainer.fit(train_data)
```

Model evaluation

There are various evaluation metrics of a binary classification machine learning model and sometimes they are evaluated on the basis of the domain of the problem. Based on the nature of our problem, our primary evaluator would be accuracy. Below is the code used for calculating evaluation metrics.

```python
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator
#Make predictions on the test set.
predictions_lr = cvModel.bestModel.transform(test_data)
predictions_svm = lsvc_cvModel.bestModel.transform(test_data)
predictions_mpc = model.transform(test_data)

#making evaluator objects
evaluator_multi = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
evaluator_bi = BinaryClassificationEvaluator()
```

```python
#function to generate evaluation metrics
def evaluation(prediction, pre,name):
    df = globals()[prediction]

    tp = df[(df.label == 1) & (df.prediction == 1)].count()
    tn = df[(df.label == 0) & (df.prediction == 0)].count()
    fp = df[(df.label == 0) & (df.prediction == 1)].count()
    fn = df[(df.label == 1) & (df.prediction == 0)].count()

    if(tp + fn == 0.0):
        r = 0.0
        p = float(tp) / (tp + fp)
    elif(tp + fp == 0.0):
        r = float(tp) / (tp + fn)
        p = 0.0
    else:
        r = float(tp) / (tp + fn)
        p = float(tp) / (tp + fp)

    f1=0 if(p+r==0) else 2*((p*r)/(p+r))

    Accuracy = evaluator_multi.evaluate(pre)
    Test_Area_Under_ROC =  evaluator_bi.evaluate(pre)
    print(name)
    print("Accuracy: {} \nTest Area Under ROC: {}\nTrue Positives: {} \nTrue Negatives: {} \nFalse Positives: {} \nFalse Positives: {} \nRecall: {} \nPrecision: {} \nF1 score: {} ".
          format(Accuracy,Test_Area_Under_ROC,tp,tn,fp,fn,r,p,f1))

evaluation("predictions_mpc",predictions_mpc, "Multilayer perceptron classifier")
```

Evaluation Metrics for Multilayer perceptron classifier:

```
Multilayer perceptron classifier
Accuracy: 0.8441792039082885
Test Area Under ROC: 0.8819808777991779
True Positives: 14375
True Negatives: 61656
False Positives: 4545
False Positives: 9489
Recall: 0.6023717733825008
Precision: 0.7597780126849895
F1 score: 0.6719801795063575
```

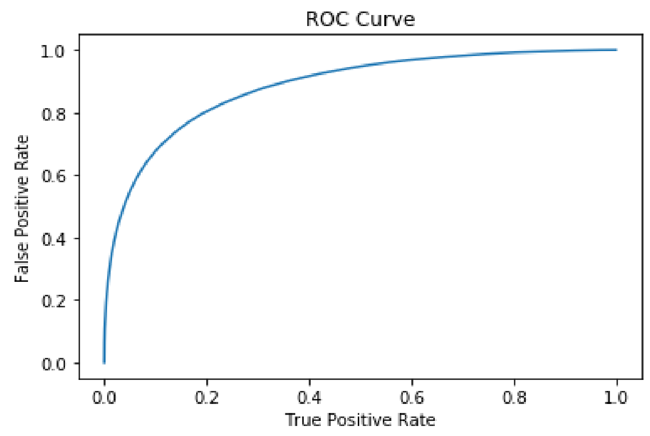
Evaluation Metrics for Logistic Regression:

```
Logistic Regression
Accuracy: 0.8452895131294066
Test Area Under ROC: 0.8835570766404409
True Positives: 14377
True Negatives: 61754
False Positives: 4447
False Positives: 9487
Recall: 0.6024555816292323
Precision: 0.7637590310242244
F1 score: 0.6735850824587706
```

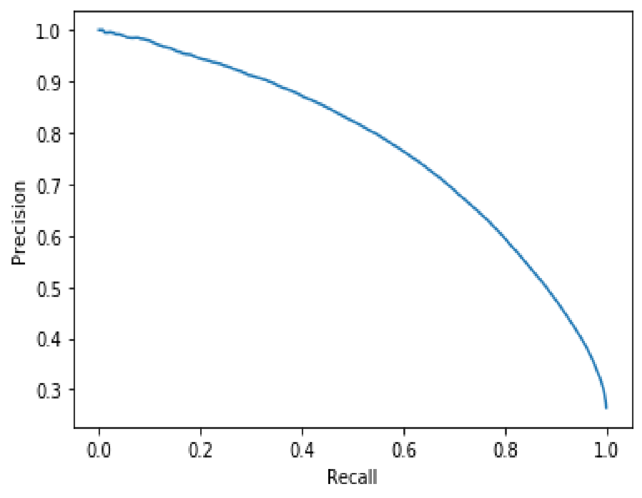
Evaluation Metrics for Linear support Vector Machines

```
Linear Support Vector Machine
Accuracy: 0.7595070227058236
Test Area Under ROC: 0.8668606600147624
True Positives: 2254
True Negatives: 66151
False Positives: 50
False Positives: 21610
Recall: 0.09445189406637614
Precision: 0.9782986111111112
F1 score: 0.17227147661265668
```

Out of all the Models, we can see that the Logistic regression model has the highest accuracy of 84.52% whereas Multilayer perceptron Classifier has performed equally well with an accuracy of 84.41%. Here is the ROC curve and Precision vs Recall curve of Logistic regression model. The AUC in the below curve is actually a weighted metric and hence is slightly different from overall AUC value we calculated above.



```
areaUnderROC: 0.883997520316556
```



Discussion

There are various machine learning models available in the pyspark libraries for classification problems and we have not considered Decision tree-based algorithms because they are not suitable for high dimension problems as we have more than 30 features in out dataset. KNN is not available in pyspark libraries and Naïve bayes was ruled out because of high correlation between numerical features and remaining algorithms have been used out of which Logistic regression is the best performer with 84.5% of accuracy. For Logistic regression and Liner SVM we have also used parameter grid with a range of hyperparameters with k-fold cross validation to get the best possible model on the basis of accuracy. For binary and linear classification problems, logistic regression is a simple and more efficient technique. In the industrial world, it is a widely used classification algorithm.