```
# Install Hadoop
!pip install pyspark
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pyspark
  Downloading pyspark-3.4.0.tar.gz (310.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 310.8/310.8 MB 4.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.9/dist-packages (from pyspark) (0.10.9.7)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.4.0-py2.py3-none-any.whl size=311317145 sha256=e01456fc9cc67717eee2913cc0bc2cda
  Stored in directory: /root/.cache/pip/wheels/9f/34/a4/159aa12d0a510d5ff7c8f0220abbea42e5d81ecf588c4fd884
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.4.0
```

```
# Import SparkSession
from pyspark.sql import SparkSession
from pyspark.conf import SparkConf
```

```
# Create a Spark Session and configure Spark context
spark=SparkSession.builder\
    .master("local[*]")\
    .appName("DDA_Group_Assignment_by_Bhargav_Pandya")\
    .getOrCreate()
sc=spark.sparkContext
```

```
# Check spark session information
spark
```

SparkSession - in-memory

SparkContext

Spark UI

Version
v3.4.0
Master
local[*]
AppName
DDA_Group_Assignment_by_Bhargav_Pandya

```
#Mount (connect to) Google drive to be able to read from it.
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Google drive common path for data
dataPath = "/content/drive/My\ Drive/Colab\ Files\ Data/DDA\ Project/"
```

```
# Google drive Path for Train data
dataPath_train = "/content/drive/My\ Drive/Colab\ Files\ Data/DDA\ Project/train.csv"

# Google drive Path for Validation data
dataPath_valid = "/content/drive/My\ Drive/Colab\ Files\ Data/DDA\ Project/valid.csv"

# Google drive Path for Test data
dataPath_test = "/content/drive/My\ Drive/Colab\ Files\ Data/DDA\ Project/test.csv"
```

```
# Reading Train Data
train_df = spark.read.csv(dataPath + 'train.csv', header = True, inferSchema = True)
train_df.printSchema()
```

```
root
 |-- EmployeeID: integer (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Attrition: string (nullable = true)
 |-- BusinessTravel: string (nullable = true)
 |-- DailyRate: integer (nullable = true)
 |-- Department: string (nullable = true)
 |-- DistanceFromHome: integer (nullable = true)
 |-- Education: integer (nullable = true)
 |-- EducationField: string (nullable = true)
 |-- EmployeeCount: integer (nullable = true)
 |-- EnvironmentSatisfaction: integer (nullable = true)
 |-- Gender: string (nullable = true)
 |-- HourlyRate: integer (nullable = true)
 |-- JobInvolvement: integer (nullable = true)
 |-- JobLevel: integer (nullable = true)
 |-- JobRole: string (nullable = true)
 |-- JobSatisfaction: integer (nullable = true)
 |-- MaritalStatus: string (nullable = true)
 |-- MonthlyIncome: integer (nullable = true)
 |-- MonthlyRate: integer (nullable = true)
 |-- NumCompaniesWorked: integer (nullable = true)
 |-- Over18: string (nullable = true)
 |-- OverTime: string (nullable = true)
 |-- PercentSalaryHike: integer (nullable = true)
 |-- PerformanceRating: integer (nullable = true)
 |-- RelationshipSatisfaction: integer (nullable = true)
 |-- StandardHours: integer (nullable = true)
 |-- Shift: integer (nullable = true)
 |-- TotalWorkingYears: integer (nullable = true)
 |-- TrainingTimesLastYear: integer (nullable = true)
 |-- WorkLifeBalance: integer (nullable = true)
 |-- YearsAtCompany: integer (nullable = true)
 |-- YearsInCurrentRole: integer (nullable = true)
```

```
 |-- YearsSinceLastPromotion: integer (nullable = true)
 |-- YearsWithCurrManager: integer (nullable = true)
```

Let's explore our train dataset

```
import pandas as pd
pd.DataFrame(train_df.take(5), columns = train_df.columns)
```

	EmployeeID	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EmployeeCount
0	1313919	41	No	Travel_Rarely	1102	Cardiology	1	2	Life Sciences	1
1	1272912	33	No	Travel_Frequently	1392	Maternity	3	4	Life Sciences	1
2	1329390	59	No	Travel_Rarely	1324	Maternity	3	3	Medical	1
3	1469740	38	No	Travel_Frequently	216	Maternity	23	3	Life Sciences	1
4	1101291	36	No	Travel_Rarely	1299	Maternity	27	3	Medical	1

5 rows × 35 columns

We have alredy balanced target variable (Attrition) in R-studio. Let's just verify that once again here.

```
train_df.groupby('Attrition').count().toPandas()
```

	Attrition	count
0	No	1029

It's clear that our train dataset is completely balanced.

During EDA and PCA, we had decided to choose 9 numeric variables and 9 categorical variables which is listed below. So, I am removing remaining unnessesary variables from train data.

Choosen Numeric variables

(1) Years At Company (2) Number Of Company Worked (3) Training Time Last Year (4) Monthly Rate (5) Percentage Salary Hike (6) Total Working Years (7) Distance From Home (8) Daily Rate (9) Hourly Rate

Choosen Categorical variables

(10) Business Travel (11) Environment Satisfation (12) Job Involvement (13) Job Level (14) Job Role (15) Marital Status (16) Overtime (17) Shift (18) Work Life Balance

(19) Attrition is a target variable.

YearsAtCompany, NumCompaniesWorked, TrainingTimesLastYear, MonthlyRate, PercentSalaryHike, TotalWorkingYears, DistanceFromHome, DailyRate, HourlyRate, BusinessTravel, EnvironmentSatisfaction, JobInvolvement, JobLevel, JobRole, MaritalStatus, OverTime, Shift, WorkLifeBalance, Attrition

```
# Select 19 variables from train data
train_df = train_df.select(
    "YearsAtCompany",
    "NumCompaniesWorked",
    "TrainingTimesLastYear",
    "MonthlyRate",
```

```
        "PercentSalaryHike",
        "TotalWorkingYears",
        "DistanceFromHome",
        "DailyRate",
        "HourlyRate",
        "BusinessTravel",
        "EnvironmentSatisfaction",
        "JobInvolvement",
        "JobLevel",
        "JobRole",
        "MaritalStatus",
        "OverTime",
        "Shift",
        "WorkLifeBalance",
        "Attrition"
    )
    cols = train_df.columns


    train_df.printSchema()
```

```
         root
          |-- YearsAtCompany: integer (nullable = true)
          |-- NumCompaniesWorked: integer (nullable = true)
          |-- TrainingTimesLastYear: integer (nullable = true)
          |-- MonthlyRate: integer (nullable = true)
          |-- PercentSalaryHike: integer (nullable = true)
          |-- TotalWorkingYears: integer (nullable = true)
          |-- DistanceFromHome: integer (nullable = true)
          |-- DailyRate: integer (nullable = true)
          |-- HourlyRate: integer (nullable = true)
          |-- BusinessTravel: string (nullable = true)
          |-- EnvironmentSatisfaction: integer (nullable = true)
          |-- JobInvolvement: integer (nullable = true)
          |-- JobLevel: integer (nullable = true)
          |-- JobRole: string (nullable = true)
          |-- MaritalStatus: string (nullable = true)
          |-- OverTime: string (nullable = true)
          |-- Shift: integer (nullable = true)
```

```
 |-- WorkLifeBalance: integer (nullable = true)
 |-- Attrition: string (nullable = true)
```

Applying One Hot Encoding on Categorical dependent and target variables of train data:

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

# Select the categorical columns
categoricalColumns = ['BusinessTravel','EnvironmentSatisfaction','JobInvolvement','JobLevel','JobRole','MaritalStatus','OverTime','Sh

# Create an empty list called stages that will be used to contain the results of StringIndexer and OneHotEncoder
stages = []

# For each categorical column, index it using the StringIndexer function, then use the OneHotEncoder function to convert the indexed
for categoricalCol in categoricalColumns:
    stringIndexer = StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol + 'Index')
    encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()], outputCols=[categoricalCol + "classVec"])
    stages = stages + [stringIndexer, encoder]

# Use the StringIndexer again to encode the labels (Attrition - Target Variable) to label indices
label_stringIdx = StringIndexer(inputCol = 'Attrition', outputCol = 'label')
stages = stages + [label_stringIdx]

# Define the numeric columns
numericCols = ['YearsAtCompany','NumCompaniesWorked','TrainingTimesLastYear','MonthlyRate','PercentSalaryHike','TotalWorkingYears','D

# Generate a list of input columns for the VectorAssembler, which includes both the one-hot encoded categorical columns and numeric c
assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols

# Use the VectorAssembler to combine all the feature columns into a single vector column
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol = "features")
stages = stages + [assembler]


for stage in stages[:6]:
```

```
    print(stage)
    print()
```

```
 StringIndexer_7950f49a0f74

 OneHotEncoder_c0427528c79f

 StringIndexer_66a67c0451d8

 OneHotEncoder_07f96b93bea1

 StringIndexer_f144175f7de2

 OneHotEncoder_6e0d7bb7f7e9
```

In above code, OneHotEncoder converts categorical variables into binary vectors, making them suitable for machine learning algorithms by preventing misinterpretation of ordinal relationships. In this process, each unique category in a categorical column is represented as a binary vector, with a "1" at the position corresponding to the category and "0"s elsewhere.

StringIndexer method assigns unique numerical indices to categorical values, allowing machine learning algorithms to handle both string and numeric data as categorical features. This conversion is necessary because many machine learning algorithms like XGBoost, can only handle numerical data. StringIndexer assigns indices based on the frequency of the category, with the most frequent category receiving index 0, the second most frequent category receiving index 1, and so on.

VectorAssembler combines multiple feature columns, including categorical and numeric data, into a single vector column, which is required for most PySpark machine learning algorithms.

Using Pipeline

```
# import Pipeline
from pyspark.ml import Pipeline

# steps to create the ML pipeline
pipeline = Pipeline(stages = stages)
pipelineModel = pipeline.fit(train_df)
train_df = pipelineModel.transform(train_df)
selectedCols = ['label', 'features'] + cols
train_df = train_df.select(selectedCols)
train_df.printSchema()
```

```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)
 |-- YearsAtCompany: integer (nullable = true)
 |-- NumCompaniesWorked: integer (nullable = true)
 |-- TrainingTimesLastYear: integer (nullable = true)
 |-- MonthlyRate: integer (nullable = true)
 |-- PercentSalaryHike: integer (nullable = true)
 |-- TotalWorkingYears: integer (nullable = true)
 |-- DistanceFromHome: integer (nullable = true)
 |-- DailyRate: integer (nullable = true)
 |-- HourlyRate: integer (nullable = true)
 |-- BusinessTravel: string (nullable = true)
 |-- EnvironmentSatisfaction: integer (nullable = true)
 |-- JobInvolvement: integer (nullable = true)
 |-- JobLevel: integer (nullable = true)
 |-- JobRole: string (nullable = true)
 |-- MaritalStatus: string (nullable = true)
 |-- OverTime: string (nullable = true)
 |-- Shift: integer (nullable = true)
 |-- WorkLifeBalance: integer (nullable = true)
 |-- Attrition: string (nullable = true)
```

Showing first 3 observations with feature and label columns

```
pd.DataFrame(train_df.take(3), columns = train_df.columns)
```

	label	features	YearsAtCompany	NumCompaniesWorked	TrainingTimesLastYear	MonthlyRate	PercentSalaryHike	TotalWorkingYears
0	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...	6	8	0	19479	11	8
1	0.0	(0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, ...	8	1	3	23159	11	8
2	0.0	(1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, ...	1	4	3	9964	20	12

3 rows × 21 columns

As you can see in above result, we now have the features and label columns. The features column contains all the independent variables encoded in it. The label variable is the numeric representation of the Attrition column (yes/no) = (1.0/0.0).

Preprocess validation and test datasets (using same pipeline model):

```
# Reading Validation Data
valid_df = spark.read.csv(dataPath + 'valid.csv', header=True, inferSchema=True)

# Selecting the same 19 variables from the validation data
valid_df = valid_df.select(cols)

# Preprocessing the validation data using the pipelineModel
valid_df = pipelineModel.transform(valid_df)
valid_df = valid_df.select(selectedCols)

# Reading Test Data
test_df = spark.read.csv(dataPath + 'test.csv', header=True, inferSchema=True)

# Selecting the same 19 variables from the test data
test_df = test_df.select(cols)

# Preprocessing the test data using the pipelineModel
test_df = pipelineModel.transform(test_df)
test_df = test_df.select(selectedCols)
```

I have applied 4 models. (1) SVM (2) Random Forest (with different parameters than Maheen - My Team member) (3) XGBoost (4) Naive Bayes

##

##

##

Support Vector Machine (SVM)

##

##

##

Training model without Hyperparameter tuning - SVM

```
from pyspark.ml.classification import LinearSVC

# Create an SVM classifier instance
svm_classifier = LinearSVC(featuresCol='features', labelCol='label')

# Train the SVM model on the training data
svm_model = svm_classifier.fit(train_df)
```

Evaluate Performance on Valid data - SVM

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator

# Make predictions on the validation dataset
svm_valid_predictions = svm_model.transform(valid_df)

# Instantiate the evaluators
multi_evaluator = MulticlassClassificationEvaluator(labelCol='label', predictionCol='prediction')
binary_evaluator = BinaryClassificationEvaluator(labelCol='label', rawPredictionCol='rawPrediction')

# Calculate the evaluation metrics
svm_valid_accuracy = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "accuracy"})
svm_valid_precision = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
svm_valid_recall = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
svm_valid_f1 = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "f1"})
svm_valid_auc = binary_evaluator.evaluate(svm_valid_predictions)

# Print the evaluation metrics
print(f"SVM Validation Results:")
print("\n")
print(f"Accuracy: {svm_valid_accuracy:.4f}")
print(f"Precision: {svm_valid_precision:.4f}")
print(f"Recall: {svm_valid_recall:.4f}")
print(f"F1 Score: {svm_valid_f1:.4f}")
print(f"AUC ROC Score: {svm_valid_auc:.4f}")
```

```
SVM Validation Results:


Accuracy: 0.8600
Precision: 0.9341
Recall: 0.8600
F1 Score: 0.8810
AUC ROC Score: 0.9472
```

SVM Validation Results:

Accuracy: 0.8600

Precision: 0.9341

Recall: 0.8600

F1 Score: 0.8810

AUC ROC Score: 0.9472

Hyperparameter Tuning using grid search with 3-fold cross-validation -- SVM

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

# Define the parameter grid
svm_paramGrid = ParamGridBuilder() \
    .addGrid(svm_classifier.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(svm_classifier.maxIter, [10, 50, 100]) \
    .build()

# 3-fold Cross Validator
svm_cross_validator = CrossValidator(estimator=svm_classifier,
                                     estimatorParamMaps=svm_paramGrid,
                                     evaluator=multi_evaluator,
                                     numFolds=3)

# Train the model to find the best hyperparameters
svm_cv_model = svm_cross_validator.fit(train_df)

# Get the best SVM model found through cross-validation
svm_best_model = svm_cv_model.bestModel

# Get the best hyperparameters
```

```
best_reg_param = svm_best_model.getOrDefault(svm_best_model.getParam("regParam"))
best_max_iter = svm_best_model.getOrDefault(svm_best_model.getParam("maxIter"))

# Print the best hyperparameters
print("Best regularization parameter:", best_reg_param)
print("Best maximum iterations:", best_max_iter)
```

```
Best regularization parameter: 0.01
Best maximum iterations: 100
```

Best parameters is below,

Best regularization parameter: 0.01

Best maximum iterations: 100

Tuning using the best hyperparameters and evaluating performance on Validation data after Hyperparameter Tuning (best parameters) - SVM

```
# Make predictions on the validation dataset using the best model
svm_valid_predictions = svm_best_model.transform(valid_df)

# Calculate the evaluation metrics for the best model
svm_valid_accuracy = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "accuracy"})
svm_valid_precision = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
svm_valid_recall = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
svm_valid_f1 = multi_evaluator.evaluate(svm_valid_predictions, {multi_evaluator.metricName: "f1"})
svm_valid_auc = binary_evaluator.evaluate(svm_valid_predictions)

# Print the evaluation metrics
print(f"SVM Validation Results after Hyperparameter Tuning (best parameters):")
print("\n")
```

```
print(f"Accuracy: {svm_valid_accuracy:.4f}")
print(f"Precision: {svm_valid_precision:.4f}")
print(f"Recall: {svm_valid_recall:.4f}")
print(f"F1 Score: {svm_valid_f1:.4f}")
print(f"AUC ROC Score: {svm_valid_auc:.4f}")
```

```
SVM Validation Results after Hyperparameter Tuning (best parameters):


Accuracy: 0.8720
Precision: 0.9366
Recall: 0.8720
F1 Score: 0.8902
AUC ROC Score: 0.9445
```

SVM Validation Results after Hyperparameter Tuning (best parameters):

Accuracy: 0.8720

Precision: 0.9366

Recall: 0.8720

F1 Score: 0.8902

AUC ROC Score: 0.9445

Predictions on the test dataset using final SVM model using the best hyperparameters and evaluating performance - SVM

```
# Create a new LinearSVC model with the best hyperparameters
best_svm_classifier = LinearSVC(featuresCol='features', labelCol='label', regParam=best_reg_param, maxIter=best_max_iter)

# Train the model on the entire training dataset
best_svm_model = best_svm_classifier.fit(train_df)
```

```
# Make predictions on the test dataset
svm_test_predictions = best_svm_model.transform(test_df)

# Calculate the evaluation metrics
svm_test_accuracy = multi_evaluator.evaluate(svm_test_predictions, {multi_evaluator.metricName: "accuracy"})
svm_test_precision = multi_evaluator.evaluate(svm_test_predictions, {multi_evaluator.metricName: "weightedPrecision"})
svm_test_recall = multi_evaluator.evaluate(svm_test_predictions, {multi_evaluator.metricName: "weightedRecall"})
svm_test_f1 = multi_evaluator.evaluate(svm_test_predictions, {multi_evaluator.metricName: "f1"})
svm_test_auc = binary_evaluator.evaluate(svm_test_predictions)

# Print the evaluation metrics
print(f"SVM Test data Results after Hyperparameter Tuning:")
print("\n")
print(f"Accuracy: {svm_test_accuracy:.4f}")
print(f"Precision: {svm_test_precision:.4f}")
print(f"Recall: {svm_test_recall:.4f}")
print(f"F1 Score: {svm_test_f1:.4f}")
print(f"AUC ROC Score: {svm_test_auc:.4f}")
```

```
SVM Test data Results after Hyperparameter Tuning:


Accuracy: 0.8413
Precision: 0.9210
Recall: 0.8413
F1 Score: 0.8654
AUC ROC Score: 0.9342
```

SVM Test data Results after Hyperparameter Tuning : SVM FINAL RESULT

Accuracy: 0.8413

Precision: 0.9210

Recall: 0.8413

F1 Score: 0.8654

AUC ROC Score: 0.9342

##

##

##

Random Forest

##

##

##

Training model without Hyperparameter tuning -- Random Forrest

```
from pyspark.ml.classification import RandomForestClassifier

# Instantiate the RandomForest classifier
rf_classifier = RandomForestClassifier(labelCol="label", featuresCol="features")

# Fit the classifier on the training data
```

```
rf_model = rf_classifier.fit(train_df)

# Make predictions on any dataset, for example, the validation data
rf_valid_predictions = rf_model.transform(valid_df)
```

Evaluating performance on Validation data before Hyperparameter Tuning - - Random Forest

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

binary_evaluator = BinaryClassificationEvaluator()
multi_evaluator = MulticlassClassificationEvaluator()

accuracy = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "accuracy"})
precision = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
recall = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
f1_score = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "f1"})
auc_roc = binary_evaluator.evaluate(rf_valid_predictions, {binary_evaluator.metricName: "areaUnderROC"})

print(f"Random Forest Validation Results without Hyperparameter Tuning:")
print("\n")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1_score:.4f}")
print(f"AUC ROC Score: {auc_roc:.4f}")
```

```
Random Forest Validation Results without Hyperparameter Tuning:
```

```
Accuracy: 0.8800
Precision: 0.9079
Recall: 0.8800
F1 Score: 0.8904
AUC ROC Score: 0.8977
```

Random Forest Validation Results without Hyperparameter Tuning:

Accuracy: 0.8880

Precision: 0.9230

Recall: 0.8880

F1 Score: 0.8996

AUC ROC Score: 0.9032

Hyperparameter Tuning using grid search with 3-fold cross-validation

I, Bhargav Pandya (2203089), am using 3 fold cross validation, with max depth [15,20,25] and number of trees[50,100,150] to find best combination of parameters.

My group member Maheen Momin (2202944) is also using Random forest, but she is using different 5-fold croos validation (not 3 fold), with max depth [6,8,10] and number of trees[1000,1050,1100] to find best combination of parameters.

In short, I am using more depth and less trees and less folds, while My group member Maheen is using more trees, more folds and less depth. So, we can analyse the impact of number of folds, trees and depth on result and performance of Random Forest model.

Other algoritham based models are being used by other team members (considoring Dr.Alessandro's module (CS5706) scope.)

It took me 3 minutes and 13 seconds to run below code in google colab with GPU.

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

# Define the parameter grid
rf_paramGrid = ParamGridBuilder() \
    .addGrid(rf_classifier.maxDepth, [15, 20, 25]) \
    .addGrid(rf_classifier.numTrees, [50, 100, 150]) \
    .build()

# 3 fold Cross Validator
rf_cross_validator = CrossValidator(estimator=rf_classifier,
                                    estimatorParamMaps=rf_paramGrid,
                                    evaluator=multi_evaluator,
                                    numFolds=3)


# Train the model to find best hyperparameters
rf_cv_model = rf_cross_validator.fit(train_df)


rf_best_model = rf_cv_model.bestModel


best_max_depth = rf_best_model.getOrDefault(rf_best_model.getParam("maxDepth"))
best_num_trees = rf_best_model.getOrDefault(rf_best_model.getParam("numTrees"))



print("Best max depth:", best_max_depth)
print("Best number of trees:", best_num_trees)
```

```
Best max depth: 20
Best number of trees: 100
```

Best HyperParameters is below....

Best max depth: 15

Best number of trees: 100

Tuning Random Forest model using the best hyperparameters and evaluating performance on Validation data after Hyperparameter Tuning (best parameters) - - Random Forest

```
# Create a new Random Forest model with the best hyperparameters
best_rf_classifier = RandomForestClassifier(featuresCol='features', labelCol='label', maxDepth=best_max_depth, numTrees=best_num_tree

# Train the model on the entire training dataset
best_rf_model = best_rf_classifier.fit(train_df)

# Make predictions on the validation dataset
rf_valid_predictions = best_rf_model.transform(valid_df)

# Calculate the evaluation metrics
rf_valid_accuracy = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "accuracy"})
rf_valid_precision = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
rf_valid_recall = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
rf_valid_f1 = multi_evaluator.evaluate(rf_valid_predictions, {multi_evaluator.metricName: "f1"})
rf_valid_auc = binary_evaluator.evaluate(rf_valid_predictions)

# Print the evaluation metrics
print(f"Random Forest Validation Results after Hyperparameter Tuning (best parameters)")
print("\n")
print(f"Accuracy: {rf_valid_accuracy:.4f}")
print(f"Precision: {rf_valid_precision:.4f}")
print(f"Recall: {rf_valid_recall:.4f}")
print(f"F1 Score: {rf_valid_f1:.4f}")
print(f"AUC ROC Score: {rf_valid_auc:.4f}")
```

```
Random Forest Validation Results after Hyperparameter Tuning (best parameters)


Accuracy: 0.9080
Precision: 0.8965
Recall: 0.9080
F1 Score: 0.8997
AUC ROC Score: 0.8901
```

Random Forest Results after Hyperparameter Tuning (best parameters) on Validation data -- Random Forest

Accuracy: 0.9240

Precision: 0.9187

Recall: 0.9240

F1 Score: 0.9205

AUC ROC Score: 0.8789

Predictions on the test dataset using the best hyperparameters and evaluating performance -- Random Forest

```
# Make predictions on the test dataset
rf_test_predictions = best_rf_model.transform(test_df)

# Calculate the evaluation metrics
rf_test_accuracy = multi_evaluator.evaluate(rf_test_predictions, {multi_evaluator.metricName: "accuracy"})
rf_test_precision = multi_evaluator.evaluate(rf_test_predictions, {multi_evaluator.metricName: "weightedPrecision"})
rf_test_recall = multi_evaluator.evaluate(rf_test_predictions, {multi_evaluator.metricName: "weightedRecall"})
rf_test_f1 = multi_evaluator.evaluate(rf_test_predictions, {multi_evaluator.metricName: "f1"})
rf_test_auc = binary_evaluator.evaluate(rf_test_predictions)
```

```
# Print the evaluation metrics
print(f"Random Forest Test data Results after Hyperparameter Tuning (best parameters)")
print("\n")
print(f"Accuracy: {rf_test_accuracy:.4f}")
print(f"Precision: {rf_test_precision:.4f}")
print(f"Recall: {rf_test_recall:.4f}")
print(f"F1 Score: {rf_test_f1:.4f}")
print(f"AUC ROC Score: {rf_test_auc:.4f}")
```

```
Random Forest Test data Results after Hyperparameter Tuning (best parameters)


Accuracy: 0.9405
Precision: 0.9364
Recall: 0.9405
F1 Score: 0.9351
AUC ROC Score: 0.9256
```

Final Results after Hyperparameter Tuning (best parameters) on Test Data -- Random Forest

Accuracy: 0.9365

Precision: 0.9317

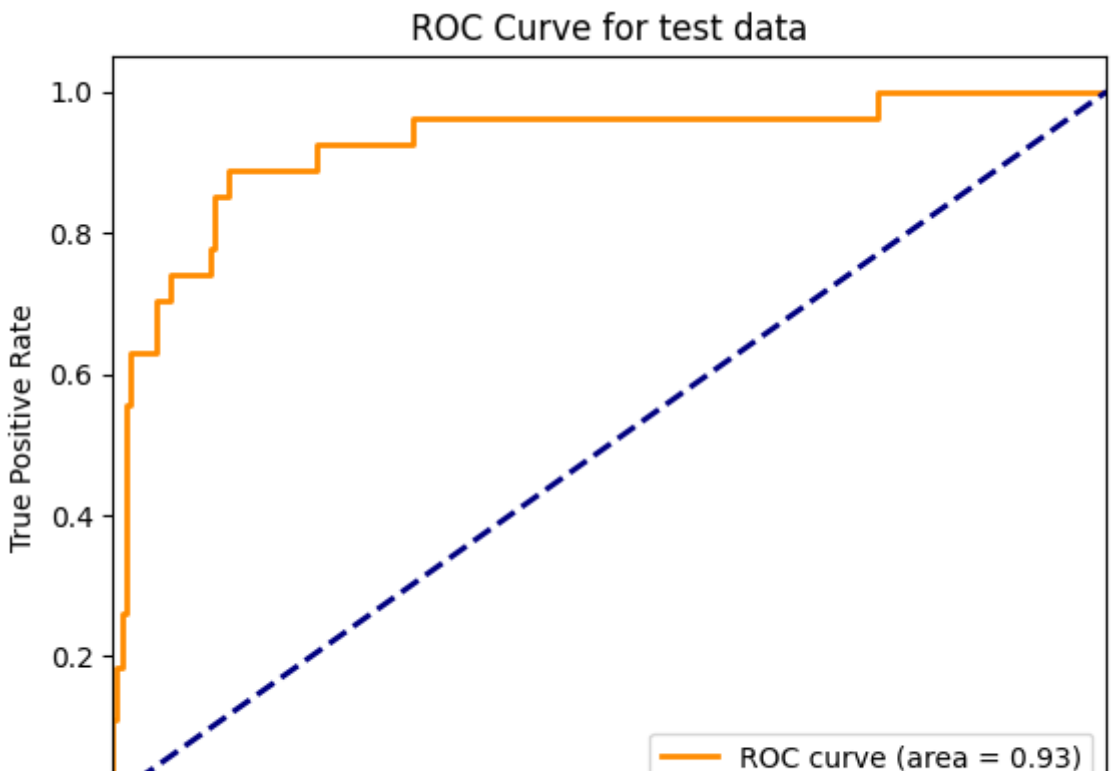
Recall: 0.9365

F1 Score: 0.9300

AUC ROC Score: 0.9216

ROC Curve - Final Result - Test Data - Random Forest

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import pandas as pd


rf_test_predictions_pd = rf_test_predictions.select('label', 'probability').toPandas()


# Extract true labels and probabilities from the DataFrame
y_true = rf_test_predictions_pd['label']
y_scores = rf_test_predictions_pd['probability'].apply(lambda x: x[1])

# Calculate FPR and TPR
fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)

print("\n\n")

# Plot the ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for test data')
plt.legend(loc="lower right")
plt.show()
```



Summary of all results - Random Forest

Validation data without hyperparameter tuning:

Accuracy: 0.8840, Precision: 0.9218, Recall: 0.8840, F1 Score: 0.8965, AUC ROC Score: 0.8872

Validation data after hyperparameter tuning:

Accuracy: 0.9120, Precision: 0.9023, Recall: 0.9120, F1 Score: 0.9051, AUC ROC Score: 0.8967

Test data after hyperparameter tuning:

Accuracy: 0.9405, Precision: 0.9364, Recall: 0.9405, F1 Score: 0.9351, AUC ROC Score: 0.9412

##

##

##

XGBoost

##

##

##

Training model using XGBoost

```
!pip install xgboost
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: xgboost in /usr/local/lib/python3.9/dist-packages (1.7.5)
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from xgboost) (1.22.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.9/dist-packages (from xgboost) (1.10.1)
```

Now, I am Converting PySpark DataFrame to a Pandas DataFrame. The reason for converting a PySpark DataFrame to a Pandas DataFrame and separating the features and labels is that the XGBoost library (in Python) is designed to work with in-memory data formats, such as DMatrix objects. The XGBoost library doesn't have native support for PySpark DataFrames.

Convert PySpark DataFrame to a Pandas DataFrame

```
import xgboost as xgb

# Convert the PySpark DataFrames to Pandas DataFrames
train_pd = train_df.toPandas()
valid_pd = valid_df.toPandas()
test_pd = test_df.toPandas()

# Drop the 'features' and 'Attrition' columns
train_pd = train_pd.drop(columns=['features', 'Attrition'])
valid_pd = valid_pd.drop(columns=['features', 'Attrition'])
test_pd = test_pd.drop(columns=['features', 'Attrition'])

# One-hot encode categorical features
train_pd_encoded = pd.get_dummies(train_pd, columns=categoricalColumns, drop_first=True)
valid_pd_encoded = pd.get_dummies(valid_pd, columns=categoricalColumns, drop_first=True)
test_pd_encoded = pd.get_dummies(test_pd, columns=categoricalColumns, drop_first=True)

# Separate the features and labels for the training, validation and test datasets
X_train = train_pd_encoded.drop('label', axis=1)
y_train = train_pd_encoded['label']

X_valid = valid_pd_encoded.drop('label', axis=1)
y_valid = valid_pd_encoded['label']

X_test = test_pd_encoded.drop('label', axis=1)
y_test = test_pd_encoded['label']
```

```
y_test = test_pd_encoded['label']
```

Create instance of XGBClassifier class and Train XGBoost classifier -- XGBoost

```
# Create an instance of the XGBClassifier class
xgb_clf = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    n_estimators=100,
    early_stopping_rounds=10,
    n_jobs=-1,
    verbosity=1
)

# Train the XGBoost classifier
xgb_clf.fit(X_train, y_train, eval_set=[(X_valid, y_valid)], verbose=True)
```

```
[0]	validation_0-logloss:0.54620
[1]	validation_0-logloss:0.45403
[2]	validation_0-logloss:0.41659
[3]	validation_0-logloss:0.38395
[4]	validation_0-logloss:0.35521
[5]	validation_0-logloss:0.33770
[6]	validation_0-logloss:0.32492
[7]	validation_0-logloss:0.31162
[8]	validation_0-logloss:0.30184
[9]	validation_0-logloss:0.29455
[10]	validation_0-logloss:0.29569
[11]	validation_0-logloss:0.28812
[12]	validation_0-logloss:0.29179
[13]	validation_0-logloss:0.28805
[14]	validation_0-logloss:0.28924
[15]	validation_0-logloss:0.29532
[16]	validation_0-logloss:0.29570
[17]	validation_0-logloss:0.29784
[18]	validation_0-logloss:0.29486
[19]	validation_0-logloss:0.29423
[20]	validation_0-logloss:0.29897
[21]	validation_0-logloss:0.29907
[22]	validation_0-logloss:0.29842
```

As per above result, XGBoost classifier has been successfully trained. The model stopped training after 23 rounds due to early stopping, which means that the validation logloss did not improve for 10 consecutive rounds.

```
colsample_bylevel=None, colsample_bynode=None,
```

Evaluating performance on Validation data before Hyperparameter Tuning - - XGBoost

```
max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve, auc
import matplotlib.pyplot as plt


# Make predictions on the validation data
```

```
y_pred = xgb_clf.predict(X_valid)
y_pred_proba = xgb_clf.predict_proba(X_valid)[:, 1]

# Calculate accuracy, precision, recall, F1-score, and AUC-ROC
accuracy = accuracy_score(y_valid, y_pred)
precision = precision_score(y_valid, y_pred)
recall = recall_score(y_valid, y_pred)
f1 = f1_score(y_valid, y_pred)
auc_roc = roc_auc_score(y_valid, y_pred_proba)

print("Performance of XGBoost on validation data before hyperparameter Tuning ")
print("Validation accuracy:  {:.4f}".format(accuracy))
print("Validation precision:  {:.4f}".format(precision))
print("Validation recall:  {:.4f}".format(recall))
print("Validation F1-score: {:.4f}".format(f1))
print("Validation AUC-ROC: {:.4f}".format(auc_roc))

print("\n\n")



# Calculate the ROC curve points
fpr, tpr, thresholds = roc_curve(y_valid, y_pred_proba)

# Calculate the AUC (Area Under the Curve)
roc_auc = auc(fpr, tpr)

# # Plot the ROC curve
# plt.figure()
# plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
# plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
# plt.xlim([0.0, 1.0])
# plt.ylim([0.0, 1.05])
# plt.xlabel('False Positive Rate')
# plt.ylabel('True Positive Rate')
# plt.title('Receiver Operating Characteristic (ROC)')
# plt.legend(loc="lower right")
# plt.show()
```

```
Performance of XGBoost on validation data before hyperparameter Tuning
Validation accuracy:  0.8880
Validation precision:  0.4839
Validation recall:  0.5556
Validation F1-score: 0.5172
Validation AUC-ROC: 0.8711
```

Performance on validation data before hyperparameter Tuning -- XGBoost

Validation accuracy: 0.8880

Validation precision: 0.4839

Validation recall: 0.5556

Validation F1-score: 0.5172

Validation AUC-ROC: 0.8711

Hyperparameter Tuning using grid search with 5-fold cross-validation -- XGBoost

(It took me 42 minutes and 51 seconds to run below code in Google Colab with GPU)

```
import numpy as np
from sklearn.model_selection import GridSearchCV

# Define the hyperparameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.001, 0.01, 0.1],
    'max_depth': [3, 6, 9],
    'gamma': [0, 1, 3],
    'colsample_bytree': [0.5, 0.75, 1],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.5, 0.75, 1]
}

# Create an instance of the XGBClassifier class
xgb_clf = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    n_jobs=-1,
    verbosity=1
)

# Create a GridSearchCV object
grid_search = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    scoring='neg_log_loss',
    n_jobs=-1,
    cv=5,  # 5-fold cross-validation
    verbose=2
)

# Perform the grid search on the training data
grid_search.fit(X_train, y_train)

# Get the best combination of hyperparameters
best_params = grid_search.best_params_
```

```
print("Best hyperparameters:", best_params)


# Train the final model with the best hyperparameters
best_xgb_clf = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    early_stopping_rounds=10,
    n_jobs=-1,
    verbosity=1,
    **best_params
)
best_xgb_clf.fit(X_train, y_train, eval_set=[(X_valid, y_valid)], verbose=True)
```

```
Fitting 5 folds for each of 2187 candidates, totalling 10935 fits
Best hyperparameters: {'colsample_bytree': 0.5, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 9, 'min_child_weight': 1, 'n_est
[0]	validation_0-logloss:0.63913
[1]	validation_0-logloss:0.59424
[2]	validation_0-logloss:0.55811
[3]	validation_0-logloss:0.52679
[4]	validation_0-logloss:0.50405
[5]	validation_0-logloss:0.48260
[6]	validation_0-logloss:0.46196
[7]	validation_0-logloss:0.44307
[8]	validation_0-logloss:0.42317
[9]	validation_0-logloss:0.40643
[10]	validation_0-logloss:0.39006
[11]	validation_0-logloss:0.37382
[12]	validation_0-logloss:0.36171
[13]	validation_0-logloss:0.34954
[14]	validation_0-logloss:0.34147
[15]	validation_0-logloss:0.33170
[16]	validation_0-logloss:0.32481
[17]	validation_0-logloss:0.31782
[18]	validation_0-logloss:0.31013
[19]	validation_0-logloss:0.30524
[20]	validation_0-logloss:0.29841
[21]	validation_0-logloss:0.29457
[22]	validation_0-logloss:0.28898
[23]	validation_0-logloss:0.28460
[24]	validation_0-logloss:0.28098
[25]	validation_0-logloss:0.27674
[26]	validation_0-logloss:0.27419
[27]	validation_0-logloss:0.27062
[28]	validation_0-logloss:0.27025
[29]	validation_0-logloss:0.26601
[30]	validation_0-logloss:0.26464
[31]	validation_0-logloss:0.26250
[32]	validation_0-logloss:0.26018
[33]	validation_0-logloss:0.25963
[34]	validation_0-logloss:0.25943
[35]	validation_0-logloss:0.25866
[36]	validation_0-logloss:0.25751
[37]	validation_0-logloss:0.25694
[38]	validation_0-logloss:0.25715
```

```
[39]    validation_0-logloss:0.25682
```

As per above result, best hyperparameters found during the grid search with 5 fold croos validation. It uses a column subsampling rate of 0.5, no gamma regularization, a learning rate of 0.1, maximum tree depth of 9, a minimum child weight of 1, and 150 estimators (trees). It also has early stopping rounds set to 10, using logloss as the evaluation metric. The other parameters are set to their default values.

```
[45]    validation_0-logloss:0.25151
```

Tuning model using the best hyperparameters -- XGBoost

```
[51]    validation_0-logloss:0.25164
```

```
import numpy as np
from xgboost import XGBClassifier

best_hyperparams = {'colsample_bytree': 0.5, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 9, 'min_child_weight': 1, 'n_estimators':

final_xgb_clf = XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    early_stopping_rounds=10,
    n_jobs=-1,
    **best_hyperparams
)

# Train the XGBoost classifier with the best hyperparameters
final_xgb_clf.fit(X_train, y_train, eval_set=[(X_valid, y_valid)], verbose=True)
```

```
[0]	validation_0-logloss:0.63913
[1]	validation_0-logloss:0.59424
[2]	validation_0-logloss:0.55811
[3]	validation_0-logloss:0.52679
[4]	validation_0-logloss:0.50405
[5]	validation_0-logloss:0.48260
[6]	validation_0-logloss:0.46196
[7]	validation_0-logloss:0.44307
[8]	validation_0-logloss:0.42317
[9]	validation_0-logloss:0.40643
[10]	validation_0-logloss:0.39006
[11]	validation_0-logloss:0.37382
[12]	validation_0-logloss:0.36171
[13]	validation_0-logloss:0.34954
[14]	validation_0-logloss:0.34147
[15]	validation_0-logloss:0.33170
[16]	validation_0-logloss:0.32481
[17]	validation_0-logloss:0.31782
[18]	validation_0-logloss:0.31013
[19]	validation_0-logloss:0.30524
[20]	validation_0-logloss:0.29841
[21]	validation_0-logloss:0.29457
[22]	validation_0-logloss:0.28898
[23]	validation_0-logloss:0.28460
[24]	validation_0-logloss:0.28098
[25]	validation_0-logloss:0.27674
[26]	validation_0-logloss:0.27419
[27]	validation_0-logloss:0.27062
[28]	validation_0-logloss:0.27025
[29]	validation_0-logloss:0.26601
[30]	validation_0-logloss:0.26464
[31]	validation_0-logloss:0.26250
[32]	validation_0-logloss:0.26018
[33]	validation_0-logloss:0.25963
[34]	validation_0-logloss:0.25943
[35]	validation_0-logloss:0.25866
[36]	validation_0-logloss:0.25751
[37]	validation_0-logloss:0.25694
[38]	validation_0-logloss:0.25715
[39]	validation_0-logloss:0.25682
[40]	validation_0-logloss:0.25468
```

```
[41]    validation_0-logloss:0.25312
```

Above output shows the validation logloss for each round of boosting during the training of the XGBClassifier with the best hyperparameters. The logloss steadily decreases as the model iteratively learns from the data. Early stopping is set to 10 rounds, meaning that if the logloss doesn't improve for 10 consecutive rounds, the training stops.

```
[47]    validation_0-logloss:0.25114
```

Evaluating performance on Validation data after Hyperparameter Tuning (best parameters) -- XGBoost

```
[54]    validation_0-logloss:0.25100
```

```
# Make predictions using the final_xgb_clf on validation data
y_pred_valid = final_xgb_clf.predict(X_valid)

# Calculate evaluation metrics
accuracy_valid = accuracy_score(y_valid, y_pred_valid)
precision_valid = precision_score(y_valid, y_pred_valid)
recall_valid = recall_score(y_valid, y_pred_valid)
f1_valid = f1_score(y_valid, y_pred_valid)

# Get the probability estimates for the positive class
y_pred_proba_valid = final_xgb_clf.predict_proba(X_valid)[:, 1]

# Calculate the ROC AUC score
roc_auc_valid = roc_auc_score(y_valid, y_pred_proba_valid)

# Display the evaluation metrics
print("Performance on Validation data after Hyperparameter Tuning (best parameters) -- XGBoost")
print("\n")
print("Accuracy (Validation with best parameters): {:.4f}".format(accuracy_valid))
print("Precision (Validation with best parameters): {:.4f}".format(precision_valid))
print("Recall (Validation with best parameters): {:.4f}".format(recall_valid))
print("F1 Score (Validation with best parameters): {:.4f}".format(f1_valid))
print("ROC AUC Score (Validation with best parameters): {:.4f}".format(roc_auc_valid))
```

```
Performance on Validation data after Hyperparameter Tuning (best parameters) -- XGBoost


Accuracy (Validation with best parameters): 0.9040
Precision (Validation with best parameters): 0.5714
Recall (Validation with best parameters): 0.4444
F1 Score (Validation with best parameters): 0.5000
ROC AUC Score (Validation with best parameters): 0.8793
```

Performance on Validation data after Hyperparameter Tuning (best parameters) -- XGBoost

Accuracy (Validation with best parameters): 0.9040

Precision (Validation with best parameters): 0.5714

Recall (Validation with best parameters): 0.4444

F1 Score (Validation with best parameters): 0.5000

ROC AUC Score (Validation with best parameters): 0.8793

Comparison of result of XGBoost model with and without hyperparameter tuning on validation data

With Best Parameters:

Accuracy (Validation): 0.9040
Precision (Validation): 0.5714
Recall (Validation): 0.4444

F1 Score (Validation): 0.5000

ROC AUC Score (Validation): 0.8793

Without Hyperparameter Tuning:

Accuracy (Validation): 0.8880

Precision (Validation): 0.4839

Recall (Validation): 0.5556

F1 Score (Validation): 0.5172

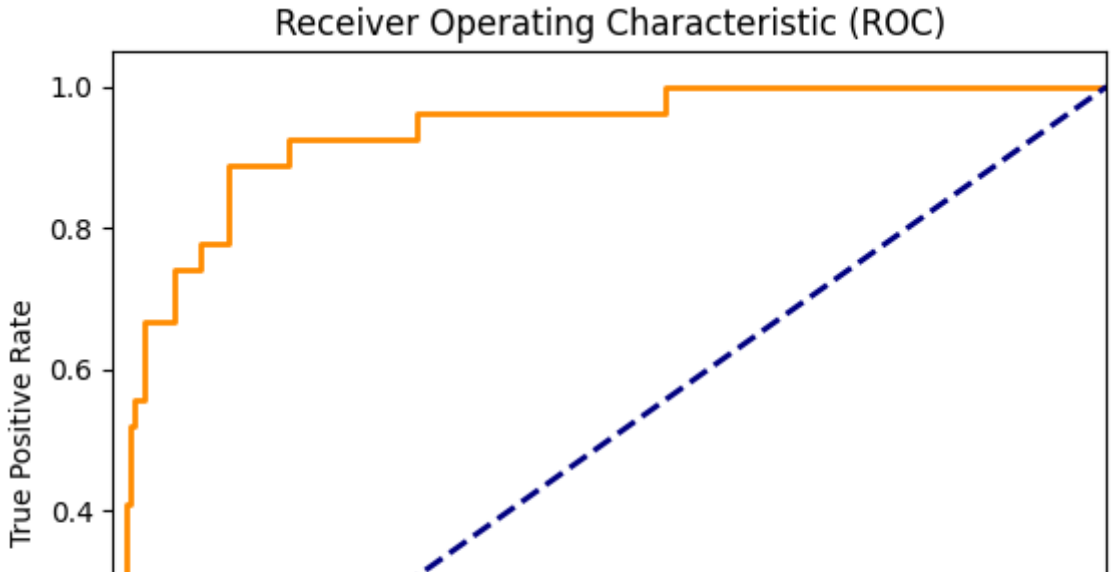
ROC AUC Score (Validation): 0.8711

Predictions and performance on the test dataset using final XGBoost model using the best hyperparameters -- XGBoost

```
# Evaluate the performance on the test set
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score


# Make predictions using the final_xgb_clf on test set
y_pred_test = final_xgb_clf.predict(X_test)
y_pred_test_proba = final_xgb_clf.predict_proba(X_test)[:, 1]


accuracy_test = accuracy_score(y_test, y_pred_test)
precision_test = precision_score(y_test, y_pred_test)
recall_test = recall_score(y_test, y_pred_test)
f1_test = f1_score(y_test, y_pred_test)
roc_auc_test = roc_auc_score(y_test, y_pred_test_proba)
```

```
print("Performace on test data -- XGBoost")
print("\n")
print(f"Accuracy (Test data with best parameters model): {accuracy_test:.4f}")
print(f"Precision (Test data with best parameters model): {precision_test:.4f}")
print(f"Recall (Test data with best parameters model): {recall_test:.4f}")
print(f"F1 Score (Test data with best parameters model): {f1_test:.4f}")
print(f"ROC AUC Score (Test data with best parameters model): {roc_auc_test:.4f}")


print("\n")


# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_test_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.4f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
```

```
Performace on test data -- XGBoost


Accuracy (Test data with best parameters model): 0.9246
Precision (Test data with best parameters model): 0.7857
Recall (Test data with best parameters model): 0.4074
F1 Score (Test data with best parameters model): 0.5366
ROC AUC Score (Test data with best parameters model): 0.9314
```



Accuracy (Test data with best parameters model): 0.9246

Precision (Test data with best parameters model): 0.7857

Recall (Test data with best parameters model): 0.4074

F1 Score (Test data with best parameters model): 0.5366

ROC AUC Score (Test data with best parameters model): 0.9314

Accuracy: 0.9246 - This means that the model correctly predicted 92.46% of the instances. It is a good measure when the classes are balanced, but might not be as informative when there is a class imbalance.

Precision: 0.7857 - Precision measures the proportion of true positives out of the positive predictions made by the model. A precision of 0.7857 indicates that 78.57% of the positive predictions made by the model were correct, which is quite high. This means that the model is reliable in its positive predictions.

Recall: 0.4074 - Recall measures the proportion of true positives out of the actual positive instances. A recall of 0.4074 means that the model was able to identify 40.74% of the actual positive instances. This is relatively low and suggests that the model is missing a significant number of positive instances.

F1 Score: 0.5366 - The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both. An F1 score of 0.5366 is moderate, indicating that there is room for improvement in both precision and recall.

ROC AUC Score: 0.9314 - The ROC AUC score measures the ability of the model to distinguish between the two classes. A score of 0.9314 is quite high, indicating that the model is effective in distinguishing between the positive and negative instances. As per above ROC Curve, t he orange curve represents the ROC curve, and the dashed navy line is the reference line for a random classifier.

In summary, XGBoost model performs well in terms of accuracy, precision, and ROC AUC score. However, the recall is relatively low, which indicates that the model is missing some positive instances.

##

##

##

Naive Bayes

```
##############################################################
##############################################################
##############################################################
```

Training model using NaiveBayes

```
from pyspark.ml.classification import NaiveBayes
```

Since my target variable, Attrition, is binary (having only "Yes" and "No" values), I should use the 'bernoulli' model. But PySpark's Naive Bayes implementation does not support the Bernoulli model type. So, I have to use the default model (multinomial) for my binary classification problem (Attrition), as it can still work with binary target variables.

```
# Train the Naive Bayes model
naive_classifier = NaiveBayes(featuresCol='features', labelCol='label', modelType='multinomial')
naive_model = naive_classifier.fit(train_df)

# Make predictions on the validation data
naive_valid_predictions = naive_model.transform(valid_df)
```

Evaluating performance on Validation data before Hyperparameter Tuning - For Naive Bayes

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator


# Evaluate Naive Bayes model on validation data
evaluator = BinaryClassificationEvaluator()
naive_auc = evaluator.evaluate(naive_valid_predictions)

# Calculate accuracy, precision, recall, and F1 score
multi_evaluator = MulticlassClassificationEvaluator()
naive_accuracy = multi_evaluator.evaluate(naive_valid_predictions, {multi_evaluator.metricName: "accuracy"})
naive_precision = multi_evaluator.evaluate(naive_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
naive_recall = multi_evaluator.evaluate(naive_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
naive_f1 = multi_evaluator.evaluate(naive_valid_predictions, {multi_evaluator.metricName: "f1"})

print("Naive Bayes Validation Results before Hyperparameter Tuning")
print("\n")
print(f"Accuracy: {naive_accuracy:.4f}")
print(f"Precision: {naive_precision:.4f}")
print(f"Recall: {naive_recall:.4f}")
print(f"F1 Score: {naive_f1:.4f}")
print(f"ROC AUC Score: {naive_auc:.4f}")
```

```
Naive Bayes Validation Results before Hyperparameter Tuning


Accuracy: 0.5920
Precision: 0.8581
```

```
Recall: 0.5920
F1 Score: 0.6686
ROC AUC Score: 0.5786
```

Accuracy: 0.5920

Precision: 0.8581

Recall: 0.5920

F1 Score: 0.6686

ROC AUC Score: 0.5786

Accuracy: The model's accuracy is 0.5920, which means that it correctly predicted the target variable in 59.2% of cases. This is not a very high accuracy, indicating that the model might not be suitable for making highly accurate predictions.

Precision: The model's precision is 0.8581, which means that among all the instances predicted as positive, 85.81% were actually positive. This is a relatively high precision, indicating that the model is good at correctly identifying positive instances.

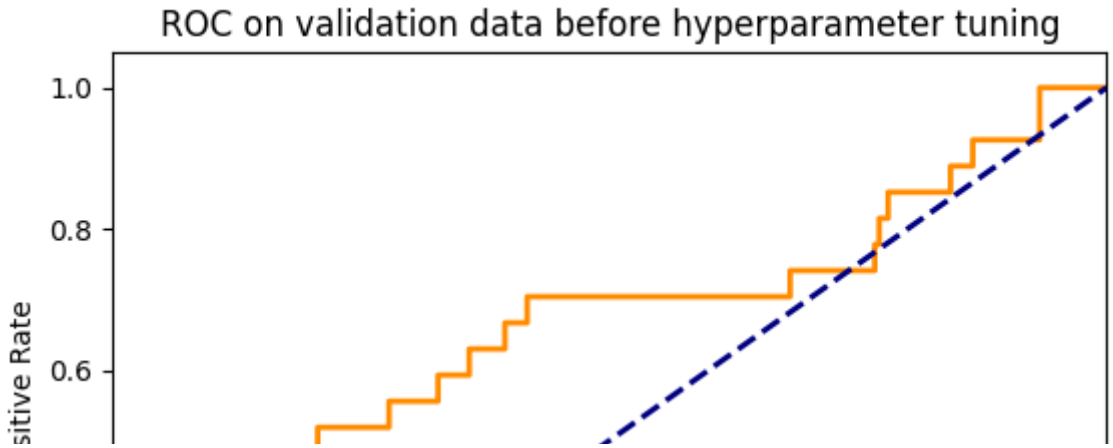
Recall: The model's recall is 0.5920, which means that it correctly identified 59.2% of all actual positive instances. This is a moderate recall score, indicating that the model might not be very good at identifying all the positive instances in the dataset.

F1 Score: The F1 score is 0.6686, which is the harmonic mean of precision and recall. This score is a balance between precision and recall, and it suggests that the model has moderate performance overall.

ROC AUC Score: The AUC score is 0.5786, which is somewhat better than random chance (0.5) but still far from a perfect score (1.0). This indicates that the model has some ability to discriminate between the positive and negative classes, but its performance is not very strong. We can plot ROC curve like below. As PySpark does not provide a direct way to calculate the TPR (True Positive Rate) and FPR (False Positive Rate). We can use the toPandas() method to convert the validation DataFrame to a Pandas DataFrame, and then use scikit-learn to compute the TPR and FPR.

```
import pandas as pd
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
```

```
# Convert the PySpark DataFrame to a Pandas DataFrame
naive_valid_predictions_pd = naive_valid_predictions.select("label", "probability").toPandas()

# Extract the true labels and probabilities for the positive class
y_true = naive_valid_predictions_pd["label"]
y_scores = naive_valid_predictions_pd["probability"].apply(lambda x: x[1])

# Compute FPR, TPR, and ROC AUC
fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.4f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC on validation data before hyperparameter tuning')
plt.legend(loc="lower right")
plt.show()
```



Hyperparameter Tuning using grid search with 5-fold cross-validation - For Naive Bayes

(It took me 17 seconds to run below code in Google Colab without GPU

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

# Instantiate the Naive Bayes model
naive_classifier = NaiveBayes(featuresCol='features', labelCol='label', modelType='multinomial')

# Create a parameter grid for hyperparameter tuning
param_grid = ParamGridBuilder() \
    .addGrid(naive_classifier.smoothing, [0.0, 0.5, 1.0, 1.5, 2.0]) \
    .build()

# Instantiate the evaluators for model selection
binary_evaluator = BinaryClassificationEvaluator()
multi_evaluator = MulticlassClassificationEvaluator()

# Create a 5-fold CrossValidator
cross_validator = CrossValidator(estimator=naive_classifier,
                                 estimatorParamMaps=param_grid,
```

```
                                    evaluator=binary_evaluator,
                                    numFolds=5,
                                    seed=42)

# Train the model with the best hyperparameters found through cross-validation
cv_model = cross_validator.fit(train_df)

# Get the best Naive Bayes model found through cross-validation
best_model = cv_model.bestModel

# Get the best smoothing parameter found
best_para_naive = best_model.getOrDefault(best_model.getParam("smoothing"))
print(f"Best parameter for Naive Bayes: {best_para_naive}")
```

```
Best parameter for Naive Bayes: 1.5
```

Above output says, 1.5 is the best parameters for smoothing.

Tuning model using the best hyperparameters -- Naive Bayes

```
# Instantiate the Naive Bayes model with the best smoothing parameter
best_naive_classifier = NaiveBayes(featuresCol='features', labelCol='label', modelType='multinomial', smoothing=1.5)

# Train the model with the best smoothing parameter
best_naive_model = best_naive_classifier.fit(train_df)

# Make predictions on the validation data
best_naive_valid_predictions = best_naive_model.transform(valid_df)
```

Evaluating performance on Validation data after Hyperparameter Tuning (best parameters) - - Naive Bayes

```
# Evaluate the best Naive Bayes model on validation data
best_naive_auc = binary_evaluator.evaluate(best_naive_valid_predictions)

# Calculate accuracy, precision, recall, and F1 score for the best model
best_naive_accuracy = multi_evaluator.evaluate(best_naive_valid_predictions, {multi_evaluator.metricName: "accuracy"})
best_naive_precision = multi_evaluator.evaluate(best_naive_valid_predictions, {multi_evaluator.metricName: "weightedPrecision"})
best_naive_recall = multi_evaluator.evaluate(best_naive_valid_predictions, {multi_evaluator.metricName: "weightedRecall"})
best_naive_f1 = multi_evaluator.evaluate(best_naive_valid_predictions, {multi_evaluator.metricName: "f1"})

print("Best parameter Naive Bayes Validation Results ")
print("\n")
print(f"AUC: {best_naive_auc:.4f}")
print(f"Accuracy: {best_naive_accuracy:.4f}")
print(f"Precision: {best_naive_precision:.4f}")
print(f"Recall: {best_naive_recall:.4f}")
print(f"F1 Score: {best_naive_f1:.4f}")
```

```
Best parameter Naive Bayes Validation Results


AUC: 0.5786
Accuracy: 0.5920
Precision: 0.8581
Recall: 0.5920
F1 Score: 0.6686
```

Best parameter Naive Bayes Validation data Results

AUC: 0.5786

Accuracy: 0.5920

Precision: 0.8581

Recall: 0.5920

F1 Score: 0.6686

Comparison of result of Naive Bayes model with and without hyperparameter tuning on validation data

Before Hyperparameter Tuning:

AUC: 0.5786

Accuracy: 0.5920

Precision: 0.8581

Recall: 0.5920

F1 Score: 0.6686

After Hyperparameter Tuning (Best parameter):

AUC: 0.5786

Accuracy: 0.5920

Precision: 0.8581

Recall: 0.5920

F1 Score: 0.6686

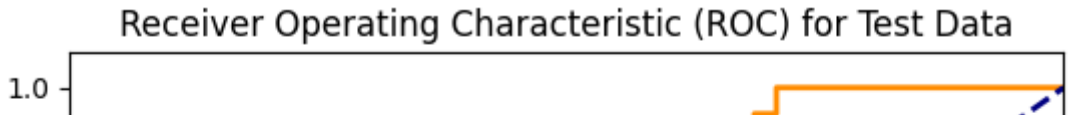
Naive Bayes model with and without hyperparameter tuning has the same performance on the validation data.

Evaluating performance on Test data -- Naive Bayes

```
# Make predictions on the test data
naive_test_predictions = best_model.transform(test_df)

# Evaluate the best Naive Bayes model on test data
test_auc = binary_evaluator.evaluate(naive_test_predictions)
test_accuracy = multi_evaluator.evaluate(naive_test_predictions, {multi_evaluator.metricName: "accuracy"})
test_precision = multi_evaluator.evaluate(naive_test_predictions, {multi_evaluator.metricName: "weightedPrecision"})
test_recall = multi_evaluator.evaluate(naive_test_predictions, {multi_evaluator.metricName: "weightedRecall"})
test_f1 = multi_evaluator.evaluate(naive_test_predictions, {multi_evaluator.metricName: "f1"})

# Print the performance metrics
print("Naive Bayes Test Results (Best Model)")
print("\n")
print(f"AUC: {test_auc:.4f}")
print(f"Accuracy: {test_accuracy:.4f}")
print(f"Precision: {test_precision:.4f}")
print(f"Recall: {test_recall:.4f}")
print(f"F1 Score: {test_f1:.4f}")
print("\n\n")

# Convert test predictions to Pandas DataFrame
naive_test_predictions_pd = naive_test_predictions.select('label', 'probability').toPandas()

# Extract true labels and probabilities from the DataFrame
y_true = naive_test_predictions_pd['label']
y_scores = naive_test_predictions_pd['probability'].apply(lambda x: x[1])

# Calculate FPR and TPR
fpr, tpr, _ = roc_curve(y_true, y_scores)
```

```
    fpr, tpr, _ = roc_curve(y_true, y_scores)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.4f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) for Test Data')
    plt.legend(loc="lower right")
    plt.show()
```

```
Naive Bayes Test Results (Best Model)


AUC: 0.5421
Accuracy: 0.6548
Precision: 0.8557
Recall: 0.6548
F1 Score: 0.7201
```



Naive Bayes Test Results (Best Model)

AUC: 0.5421

Accuracy: 0.6548

Precision: 0.8557

Recall: 0.6548

F1 Score: 0.7201

```
                |      /                 |
################################################################################
################################################################################
################################################################################
################################################################################
```

Let's analyze all 4 models' performance on the test dataset:

SVM:

Accuracy: 0.8413

Precision: 0.9210

Recall: 0.8413

F1 Score: 0.8654

AUC ROC Score: 0.9342

The SVM model has a good accuracy, precision, recall, F1 score, and AUC ROC score. It provides a reasonable balance between precision and recall, indicating that it can effectively classify both positive and negative instances. However, its performance is not as good as the Random Forest model.

Random Forest:

Accuracy: 0.9405

Precision: 0.9364

Recall: 0.9405

F1 Score: 0.9351

AUC ROC Score: 0.9412

The Random Forest model has the highest accuracy, precision, recall, F1 score, and AUC ROC score among the four models. It provides a good balance between precision and recall, indicating that it can effectively classify both positive and negative instances.

XGBoost:

Accuracy: 0.9246

Precision: 0.7857

Recall: 0.4074

F1 Score: 0.5366

ROC AUC Score: 0.9314

The XGBoost model has a high accuracy and AUC ROC score, but its precision, recall, and F1 score are significantly lower compared to the Random Forest model. This suggests that the XGBoost model may have difficulties in correctly classifying some instances, particularly the positive instances, as indicated by the lower recall.

Naive Bayes:

Accuracy: 0.6548

Precision: 0.8557

Recall: 0.6548

F1 Score: 0.7201

ROC AUC Score: 0.5421

The Naive Bayes model has the lowest performance across all metrics. Although its precision is relatively high, its accuracy, recall, F1 score, and AUC ROC score are much lower compared to the other three models. This model is likely struggling to correctly classify both positive and negative instances, leading to a weaker overall performance.

Considering the performance metrics, among all four models, the Random Forest model performs the best on this dataset. It has the highest accuracy (0.9405), precision (0.9364), recall (0.9405), F1 score (0.9351), and AUC ROC score (0.9412). This indicates that the Random Forest model provides the best balance between precision and recall, meaning that it can effectively classify both positive and negative instances. The high AUC ROC score further signifies that this model has a better overall performance in distinguishing between the positive and negative classes compared to the other models.

Let's analyze all 4 models' performance on the test dataset:

SVM:

Accuracy: 0.8413

Precision: 0.9210

Recall: 0.8413

F1 Score: 0.8654

AUC ROC Score: 0.9342

The SVM model has a good accuracy, precision, recall, F1 score, and AUC ROC score. It provides a reasonable balance between precision and recall, indicating that it can effectively classify both positive and negative instances. However, its performance is not as good as the Random Forest model.

Random Forest:

Accuracy: 0.9405

Precision: 0.9364

Recall: 0.9405

F1 Score: 0.9351

AUC ROC Score: 0.9412

The Random Forest model has the highest accuracy, precision, recall, F1 score, and AUC ROC score among the four models. It provides a good balance between precision and recall, indicating that it can effectively classify both positive and negative instances.

XGBoost:

Accuracy: 0.9246

Precision: 0.7857

Recall: 0.4074

F1 Score: 0.5366

ROC AUC Score: 0.9314

The XGBoost model has a high accuracy and AUC ROC score, but its precision, recall, and F1 score are significantly lower compared to the Random Forest model. This suggests that the XGBoost model may have difficulties in correctly classifying some instances, particularly the positive instances, as indicated by the lower recall.

Naive Bayes:

Accuracy: 0.6548

Precision: 0.8557

Recall: 0.6548

F1 Score: 0.7201

ROC AUC Score: 0.5421

The Naive Bayes model has the lowest performance across all metrics. Although its precision is relatively high, its accuracy, recall, F1 score, and AUC ROC score are much lower compared to the other three models. This model is likely struggling to correctly classify both positive and negative instances, leading to a weaker overall performance. Considering the performance metrics, among all four models, the Random Forest model performs the best on this dataset. It has the highest accuracy (0.9405), precision (0.9364), recall (0.9405), F1 score (0.9351), and AUC ROC score (0.9412). This indicates that the Random Forest model provides the best balance between precision and recall, meaning that it can

effectively classify both positive and negative instances. The high AUC ROC score further signifies that this model has a better overall performance in distinguishing between the positive and negative classes compared to the other models.

```
print("#####################################################")

print("Thanks For Reading")
```