03_Sparkify_Modelling

March 22, 2022

1 Part 3: Modelling

1.1 Load libraries, create Spark session and import data

In [1]:
```
# import libraries
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.sql.functions import countDistinct
import re
from pyspark.sql.types import StringType, DoubleType, IntegerType

import datetime
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

# Spark ML libraries
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer

from sklearn.model_selection import train_test_split
import numpy as np

from pyspark.ml.classification import LogisticRegression, RandomForestClassifier, GBTCla
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import  MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

import time
```

In [2]:
```
# create a Spark session
spark = SparkSession \
    .builder \
    .appName("Sparkify Project ML") \
    .getOrCreate()
```

```
In [3]: #change path here!
        path = "data/user_feature_data.json"
        data = spark.read.json(path)
```

```
In [4]: data.printSchema()
```

```
root
 |-- Add_Friend_vs_NextSong: double (nullable = true)
 |-- Add_to_Playlist_vs_NextSong: double (nullable = true)
 |-- Churned_User: boolean (nullable = true)
 |-- Downgrade_vs_NextSong: double (nullable = true)
 |-- Roll_Advert_vs_NextSong: double (nullable = true)
 |-- Submit_Downgrade_vs_NextSong: double (nullable = true)
 |-- Submit_Upgrade_vs_NextSong: double (nullable = true)
 |-- Thumbs_Down_vs_NextSong: double (nullable = true)
 |-- Thumbs_Up_vs_NextSong: double (nullable = true)
 |-- Upgrade_vs_NextSong: double (nullable = true)
 |-- avg_amount_songs_played_per_session: double (nullable = true)
 |-- browser: string (nullable = true)
 |-- city: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- hours_streaming_per_active_day: double (nullable = true)
 |-- level: string (nullable = true)
 |-- percentage_active_days: double (nullable = true)
 |-- platform: string (nullable = true)
 |-- state: string (nullable = true)
 |-- userId: string (nullable = true)
```

2 5. Modeling

Before we create the final dataframe to fit ML Models on we should check the features for multicollinearity. Multicollinearity happens when independent variables in the regression model are highly correlated to each other. It makes it hard to interpret of model and also creates an overfitting problem [1].

Decision Trees and boosted tree algorithmns are immune to multicollinearty by nature. When they decide to split, the tree will choose only one of the perfectly correlated features. However, other algorithms like Logistic Regression or Linear Regression are not immune to that problem and you should fix it before training the model [2].

In the Heatmap below we can see that there is a high correlation between 'avg_amount_songs_played_per_session' and 'hours_streaming_per_active_day'. This makes sense since we would assume that users who play many songs per session would also have many hours streaming per active day. As a solution we will drop the 'avg_amount_songs_played_per_session' column.

```
In [5]: data_pd = data.toPandas()
        corr = data_pd.corr()
        corr.style.background_gradient(cmap='coolwarm')
```

Out[5]: <pandas.io.formats.style.Styler at 0x7fb2398aa518>

```
In [6]: def drop_correlated_column_from_feature_data(data, name_correlated_col = 'avg_amount_son
            '''
            This function drops the correlated column (here: 'avg_amount_songs_played_per_sessio

            args:
                data (pyspark dataframe): dataframe containing the features with one column per

            returns:
                data (pyspark dataframe): dataframe without the dropped column
            '''

            data = data.drop(name_correlated_col)

            return data
```

```
In [7]: data = drop_correlated_column_from_feature_data(data)
```

As seen from above we have some columns that have numeric and some that have categorical values. Before using the categorical values in a ML model we need to use a StringIndexer Function. The String Indexer encodes a string column of labels to a column of label indices [1]. The numerical columns we need to assemble with a Vector Assembler and then scale. Both the assembled categorical column values and the numerical column values then need to be assembled to one vector. This is done in the following:

```
In [8]: #Helper functions to find the names of the numeric and categorical values so they don't
        def get_list_of_numerical_columns(data):
            '''
            This function creates an array with column names of the numeric columns (that are Do

            args:
                data (pyspark dataframe): dataframe containing the features with one column per

            returns:
                num_columns (array): array containing the names of the numerical columns
            '''

            num_columns = [f.name for f in data.schema.fields if isinstance(f.dataType, (DoubleT

            return num_columns

        def get_list_of_categorical_columns(data):
            '''
            This function creates an array with column names of the categorical columns (that ar

            args:
                data (pyspark dataframe): dataframe containing the features with one column per
```

```
        returns:
            cat_columns (array): array containing the names of the categorical columns
        '''

        cat_columns = [f.name for f in data.schema.fields if isinstance(f.dataType, (StringT
        cat_columns.remove('userId')

        return cat_columns
```

In [9]:
```
get_list_of_categorical_columns(data)
```

Out[9]:
```
['browser', 'city', 'gender', 'level', 'platform', 'state']
```

In [10]:
```
def create_feature_preprocessing_pipeline(data):
    '''
    Function to create a preprocessing pipeline to process the dataframe and make a fea
    Categorical Variables are processed with a String Indexer and numerical columns are
    a Standard Scaler

    args:
        data (pyspark dataframe): dataframe containing the features one column per feat

    returns:
        feature_preprocessing_pipeline (pyspark pipeline): a pipeline that processes nu
        features into a featurevector-column

    '''
    num_columns = get_list_of_numerical_columns(data)

    #categorical columns are: ['browser', 'city', 'gender', 'level', 'platform', 'state
    #for features/predicting we will only use gender and level

    # index categorical columns
    indexer_gender = StringIndexer(inputCol='gender', outputCol='gender_indexed')
    indexer_level = StringIndexer(inputCol='level', outputCol='level_indexed')

    #asseble the categorical columns
    assembler_categorical = VectorAssembler(inputCols = ['gender_indexed', 'level_index
                                            outputCol = 'vectorized_categorical_columns')

    # assemble and scale numerical columns
    assembler_numerical = VectorAssembler(inputCols = num_columns, outputCol = 'vectori
    scaler_standard = StandardScaler(inputCol = 'vectorized_numerical_columns', outputC

    # assemble all columns together into the features column, label will be churned/not
    assembler_all = VectorAssembler(inputCols = ['vectorized_categorical_columns'] + ['

    feature_preprocessing_pipeline = Pipeline(stages=[indexer_gender, indexer_level, as
```

```
                                            assembler_numerical, scaler_standard, assemb

            return feature_preprocessing_pipeline
```

```
In [11]: feature_preprocessing_pipeline = create_feature_preprocessing_pipeline(data)
         data_processed = feature_preprocessing_pipeline.fit(data).transform(data)
```

```
In [12]: def create_label_features_dataframe(data_processed):
            '''
            Function to create a label-feature dataframe to train the models on

            args:
                data_processed (pyspark dataframe): dataframe from function:
                        feature_preprocessing_pipeline = create_feature_preprocessing_pipeline(
                        data_processed = feature_preprocessing_pipeline.fit(data).transform(dat

            returns:
                data_model (pyspark dataframe): dataframe to train the models on with 'feature'
            '''


            data_processed = data_processed.withColumnRenamed('Churned_User','label')
            data_model = data_processed.select('label', 'features')

            data_model = data_model.withColumn('label',F.col('label').cast(IntegerType()))

            return data_model
```

```
In [13]: data_model = create_label_features_dataframe(data_processed)
```

```
In [14]: data_model.printSchema()
```

```
root
 |-- label: integer (nullable = true)
 |-- features: vector (nullable = true)
```

As we have seen in the Dataset Analysis we have an imbalance in the dataset of churned users vs stayed users:

Of total 225 users, 173 users stayed with the streaming service during the observed time and 52 users eventually churned (churn rate: 23.11%)

Therefore random sampling could lead to our model not having not many churned-samples to be trained on. Therefore we will use a stratified sampling strategy in the following. In a classification setting, Stratified sampling is often chosen to ensure that the train and test sets have approximately the same percentage of samples of each target class as the complete set [3].

```
In [15]: def create_train_test_split_stratified_sampling(data_model, percentage_training_data):
            '''
```

```
            Function to splits unbalanced data into train and test data using stratified sampli
            churn/ not churn rations in triaining and test data.

            args:
                data_processed (pyspark dataframe): dataframe containing processed data for ML
                percentage_training_data (float): percentage of training data from complete dat

            returns:
                data_training (pyspark dataframe): dataframe containing training data
                data_test (pyspark dataframe): dataframe containing test data
            '''

            # Taking the percentage_training_data of both 0's and 1's of Churned_User into trai
            data_training = data_model.sampleBy('label', fractions={1: percentage_training_data

            # Subtracting 'train' from original 'data' to get test set
            data_test = data_model.subtract(data_training)

            print('The training data contains {} observations, the test data contains {} observ
                data_training.count(), data_test.count()))

            print('The training data has the following distribution in labels: ')
            data_training.groupBy('label').count().show()

            print('The testing data has the following distribution in labels: ')
            data_test.groupBy('label').count().show()

            print('The total data has the following distribution in labels: ')
            data_model.groupBy('label').count().show()

            return data_training, data_test
```

In [16]:
```
data_train, data_test = create_train_test_split_stratified_sampling(data_model, 0.8)
```

```
The training data contains 193 observations, the test data contains 32 observations
The training data has the following distribution in labels:
+-----+-----+
|label|count|
+-----+-----+
|    1|   44|
|    0|  149|
+-----+-----+

The testing data has the following distribution in labels:
+-----+-----+
|label|count|
+-----+-----+
|    1|    8|
```

```
|    0|   24|
+-----+-----+

The total data has the following distribution in labels:
+-----+-----+
|label|count|
+-----+-----+
|    1|   52|
|    0|  173|
+-----+-----+
```

2.0.1 Fit and evaluate Models

The evaluation of the models will be through the F1 score and the ROC-AUC. The ROC-AUC might be a good metric for an imbalanced binary classification.

F-Measure / F1 score Precision and recall can be combined into a single score that seeks to balance both concerns, called the F-score or the F-measure. The F-Measure is a popular metric for imbalanced classification [4].

AUC-ROC The Receiver Operator Characteristic (ROC) is a probability curve that plots the TPR(True Positive Rate) against the FPR(False Positive Rate) at various threshold values and separates the 'signal' from the 'noise'.

The greater the AUC, the better is the performance of the model at different threshold points between positive and negative classes. This simply means that When AUC is equal to 1, the classifier is able to perfectly distinguish between all Positive and Negative class points. When AUC is equal to 0, the classifier would be predicting all Negatives as Positives and vice versa. When AUC is 0.5, the classifier is not able to distinguish between the Positive and Negative classes [5].

```
In [17]: def evaluate_model_F1_and_AUC_ROC_score(results):
             '''
             Function to evaluate prediction of model using F1 and AUC ROC score

             args:
                 results (pyspark dataframe): Results from predicting classes with trained model

             returns:
                 prints out F1-Score for given results
             '''

             f1_evaluator = MulticlassClassificationEvaluator(metricName='f1')
             f1_score = f1_evaluator.evaluate(results.select(F.col('label'), F.col('prediction')

             print('F1-score = {:.4%}'.format(f1_score))
```

```
        auc_roc_evaluator = BinaryClassificationEvaluator()
        auc_roc_score = auc_roc_evaluator.evaluate(results, {auc_roc_evaluator.metricName:

        print('Area under ROC = {:.4%}'.format(auc_roc_score))
```

```
In [18]: def train_test_model(model, data_train, data_test):
            '''
            Function to:
                    - fit the model on data_train
                    - predict on data_test
                    - print the time needed for the model to fit and predict
                    - print the evaluation scores (F1 and AUC-ROC)
            '''
            start = time.time()

            clf = model.fit(data_train)
            results = clf.transform(data_test)

            end = time.time()
            print('Time spent for training and predicting: {}'.format(round(end-start,2)))

            evaluate_model_F1_and_AUC_ROC_score(results)
```

According to Apache Spark Documentation there are multiple ML models possible for a classification prediction [6].

We will work with the following models:

- Logistic Regression
- Decision tree classifier
- Random forest classifier
- Gradient boosted tree classifier
- Linear Support Vector Machine

```
In [19]: lr = LogisticRegression()
         train_test_model(lr, data_train, data_test)
```

```
Time spent for training and predicting: 5.68
F1-score = 74.1852%
Area under ROC = 86.9792%
```

```
In [20]: dt = DecisionTreeClassifier()
         train_test_model(dt, data_train, data_test)
```

```
Time spent for training and predicting: 2.32
F1-score = 74.1852%
Area under ROC = 54.4271%
```

```
In [21]: rf = RandomForestClassifier()
         train_test_model(rf, data_train, data_test)
```

```
Time spent for training and predicting: 2.37
F1-score = 74.1852%
Area under ROC = 64.5833%
```

```
In [22]: gbt = GBTClassifier()
         train_test_model(gbt, data_train, data_test)
```

```
Time spent for training and predicting: 14.01
F1-score = 83.0317%
Area under ROC = 84.8958%
```

```
In [23]: lsvc = LinearSVC()
         train_test_model(lsvc, data_train, data_test)
```

```
Time spent for training and predicting: 14.88
F1-score = 64.2857%
Area under ROC = 86.4583%
```

2.0.2 Parameter tuning

```
In [24]: def evaluation_best_model_F1(model, data_train, data_test, paramGrid):
             '''
             Function to find the model with the best parameters based on the Area Under ROC val
             '''

             cv = CrossValidator(estimator = model,
                                 estimatorParamMaps = paramGrid,
                                 evaluator = MulticlassClassificationEvaluator(metricName='f1'
                                 numFolds = 5)

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

             #get the results of train data from best model
             results = cvModel.transform(data_test)
             evaluate_model_F1_and_AUC_ROC_score(results)

             return cvModel

In [25]: def evaluation_best_model_AUC_ROC(model, data_train, data_test, paramGrid):
             '''
             Function to find the model with the best parameters based on the Area Under ROC val
             '''
```

```
    cv = CrossValidator(estimator = model,
                        estimatorParamMaps = paramGrid,
                        evaluator = BinaryClassificationEvaluator(metricName='areaUnd
                        numFolds = 5)

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

    #get the results of train data from best model
    results = cvModel.transform(data_test)
    evaluate_model_F1_and_AUC_ROC_score(results)

    return cvModel
```

Linear Regression

```
In [26]: paramGrid_lr = ParamGridBuilder() \
                 .addGrid(lr.regParam, [0.0, 0.01, 0.5]) \
                 .addGrid(lr.elasticNetParam, [0.0, 0.5]) \
                 .addGrid(lr.maxIter, [1, 5, 10, 20, 100]) \
                 .build()
```

```
In [27]: bestmodel_lr_F1 = evaluation_best_model_F1(lr, data_train, data_test, paramGrid_lr)
```

```
F1-score = 74.1852%
Area under ROC = 86.9792%
```

```
In [28]: bestmodel_lr_F1.bestModel.extractParamMap()
         #elasticNetParam: 0.0
         #maxIter: 1
         #regParam: 0.0
```

```
Out[28]: {Param(parent='LogisticRegression_bfa1bbbaa6ba', name='aggregationDepth', doc='suggeste
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='elasticNetParam', doc='the Elast
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='family', doc='The name of family
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='featuresCol', doc='features colu
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='fitIntercept', doc='whether to f
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='labelCol', doc='label column nam
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='maxIter', doc='maximum number of
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='predictionCol', doc='prediction
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='probabilityCol', doc='Column nam
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='rawPredictionCol', doc='raw pred
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='regParam', doc='regularization p
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='standardization', doc='whether t
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='threshold', doc='threshold in bi
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='tol', doc='the convergence toler
```

```
In [29]: bestmodel_lr_AUC_ROC = evaluation_best_model_AUC_ROC(lr, data_train, data_test, paramGr
```

```
F1-score = 74.1852%
Area under ROC = 86.9792%
```

```
In [30]: bestmodel_lr_AUC_ROC.bestModel.extractParamMap()
         #elasticNetParam: 0.5
         #maxIter: 20
         #regParam: 0.01
```

```
Out[30]: {Param(parent='LogisticRegression_bfa1bbbaa6ba', name='aggregationDepth', doc='suggeste
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='elasticNetParam', doc='the Elast
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='family', doc='The name of family
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='featuresCol', doc='features colu
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='fitIntercept', doc='whether to f
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='labelCol', doc='label column nam
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='maxIter', doc='maximum number of
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='predictionCol', doc='prediction
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='probabilityCol', doc='Column nam
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='rawPredictionCol', doc='raw pred
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='regParam', doc='regularization p
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='standardization', doc='whether t
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='threshold', doc='threshold in bi
          Param(parent='LogisticRegression_bfa1bbbaa6ba', name='tol', doc='the convergence toler
```

```
In [31]: bestmodel_lr_AUC_ROC.bestModel
```

```
Out[31]: LogisticRegressionModel: uid = LogisticRegression_bfa1bbbaa6ba, numClasses = 2, numFeat
```

```
In [49]: #Out of the two best models (where one is evaluated for F1 and one for AUC-ROC), we cho
         #with the following scores:

         #F1-score = 74.1852%
         #Area under ROC = 86.9792%

         #save best model
         #bestmodel_lr_AUC_ROC.bestModel.save('best_models/best_model_Linear_Regression.pkl')
```

Decision Tree Classifier

```
In [33]: paramGrid_dt = ParamGridBuilder() \
                     .addGrid(dt.cacheNodeIds, [False, True]) \
                     .addGrid(dt.impurity, ['gini', 'entropy']) \
                     .addGrid(dt.maxDepth, [1, 5, 10]) \
                     .build()
```

```
In [34]: bestmodel_dt_F1 = evaluation_best_model_F1(dt, data_train, data_test, paramGrid_dt)
```

```
F1-score = 77.6190%
Area under ROC = 50.0000%
```

```
In [35]: bestmodel_dt_AUC_ROC = evaluation_best_model_AUC_ROC(dt, data_train, data_test, paramGr
```

```
F1-score = 81.2500%
Area under ROC = 76.3021%
```

```
In [50]: #Out of the two best models (where one is evaluated for F1 and one for AUC-ROC), we cho
         #with the following scores:

         #F1-score = 81.2500%
         #Area under ROC = 76.3021%


         #save best model
         #bestmodel_dt_AUC_ROC.bestModel.save('best_models/best_model_Decision_Tree_Classifier.p
```

Random Forest Classifier

```
In [37]: paramGrid_rf = ParamGridBuilder() \
                   .addGrid(rf.cacheNodeIds, [False, True]) \
                   .addGrid(rf.impurity, ['gini', 'entropy']) \
                   .addGrid(rf.maxDepth, [5, 10, 20]) \
                   .addGrid(rf.numTrees, [10, 20, 40]) \
                   .build()
```

```
In [38]: bestmodel_rf_F1 = evaluation_best_model_F1(rf, data_train, data_test, paramGrid_rf)
```

```
F1-score = 80.2857%
Area under ROC = 80.9896%
```

```
In [39]: bestmodel_rf_AUC_ROC = evaluation_best_model_AUC_ROC(rf, data_train, data_test, paramGr
```

```
F1-score = 76.6667%
Area under ROC = 76.5625%
```

```
In [51]: #Out of the two best models (where one is evaluated for F1 and one for AUC-ROC), we cho
         #with the following scores:

         #F1-score = 80.2857%
         #Area under ROC = 80.9896%


         #save best model
         #bestmodel_rf_F1.bestModel.save('best_models/best_model_Random_Forest_Classifier.pkl')
```

Gradient boosted tree classifier

```
In [41]: paramGrid_gbt = ParamGridBuilder() \
                     .addGrid(gbt.maxIter, [10, 20]) \
                     .addGrid(gbt.maxDepth, [5, 10]) \
                     .addGrid(gbt.stepSize, [0.1, 0.5]) \
                     .build()
```

```
In [42]: bestmodel_gbt_F1 = evaluation_best_model_F1(gbt, data_train, data_test, paramGrid_gbt)
```

```
F1-score = 83.0317%
Area under ROC = 84.8958%
```

```
In [43]: bestmodel_gbt_AUC_ROC = evaluation_best_model_AUC_ROC(gbt, data_train, data_test, param
```

```
F1-score = 72.4030%
Area under ROC = 69.5312%
```

```
In [52]: #Out of the two best models (where one is evaluated for F1 and one for AUC-ROC), we cho
         #with the following scores:

         #F1-score = 83.0317%
         #Area under ROC = 84.8958%


         #save best model
         #bestmodel_gbt_F1.save('best_models/best_model_Gradient_boosted_tree_classifier.pkl')
```

Linear Support Vector Machine

```
In [45]: paramGrid_lsvc = ParamGridBuilder() \
                     .addGrid(lsvc.aggregationDepth, [2, 3]) \
                     .addGrid(lsvc.standardization, [True, False]) \
                     .addGrid(lsvc.maxIter, [10, 20, 100]) \
                     .build()
```

```
In [46]: bestmodel_lsvc_F1 = evaluation_best_model_F1(lsvc, data_train, data_test, paramGrid_lsv
```

```
F1-score = 76.6667%
Area under ROC = 80.2083%
```

```
In [47]: bestmodel_lsvc_AUC_ROC = evaluation_best_model_AUC_ROC(lsvc, data_train, data_test, par
```

```
F1-score = 76.6667%
Area under ROC = 80.2083%
```

In [53]:
```
#Both models give the same metrics, therefore we could safe either with the following e

#F1-score = 76.6667%
#Area under ROC = 80.2083%


#save best model
#bestmodel_lsvc_AUC_ROC.bestModel.save('best_models/best_model_Linear_Vector_Machine.pk
```