02_Sparkify_Feature_Engineering

March 22, 2022

1 Part 2: Feature Engineering

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
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
import re
import time
import numpy as np



from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer
from sklearn.model_selection import train_test_split
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
```

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

In [3]:
```
path = "data/mini_sparkify_event_data.json"
data = spark.read.json(path)
```

2 4. Feature Engineering

In [106]:
```
def clean_data(data):
    '''
    Function to clean data from userIds and sessionIds from NA and empty values.

    args:
        data (pyspark dataframe) - raw data of Sparkify

    returns:
        data_clean (pyspark dataframe) - cleaned data of Sparkify
    '''
    data_clean = data.dropna(how = 'any', subset = ['userId', 'sessionId'])
    data_clean = data_clean.filter(data_clean['userId'] != '')

    print('The dataset originally contained {} rows. \nAfter cleaning there are {} row
          .format(data.count(), data_clean.count()))

    return data_clean
```

In [107]:
```
def transform_date_function(data_clean):
    '''
    Function to clean data add columns 'datetime' and 'date'
    created from ts column in human readable format

    args:
        data_clean (pyspark dataframe) - cleaned data with 'clean_data(data)' function

    returns:
        data_transformed (pyspark dataframe) - transformed data with columns 'datetime
    '''
    udf_convert_ts_to_datetime = F.udf(lambda timestamp: datetime.datetime.fromtimesta
        timestamp / 1000.0).strftime('%Y-%m-%d %H:%M:%S'))

    udf_convert_ts_to_date = F.udf(lambda timestamp: datetime.datetime.fromtimestamp(
        timestamp / 1000.0).strftime('%Y-%m-%d'))

    data_transformed = data_clean.withColumn('datetime', udf_convert_ts_to_datetime(da
        .withColumn("date", udf_convert_ts_to_date(data_clean.ts))

    return data_transformed
```

In [108]:
```
def create_cancellation_event_and_churn_label(data_transformed):
    '''
    Function to create columns that flag the Churning Event ('Cancellation Confirmatio
    and a column that labels entries from users who churn eventually with 1

    args:
```

```
        data_transformed (pyspark dataframe) - transformed data with 'transform_date_f

    returns:
        data_churn (pyspark dataframe) - dataframe with new columns 'Churning Event' a
    '''
    #create column that flags Churning Events
    data_churn = data_transformed \
        .withColumn('Churning_Event', (F.when(F.col("page")=='Cancellation Confirmatio

    #create list of churned users
    churned_users = data_churn.select('userId') \
        .filter(data_churn.Churning_Event == 1) \
        .dropDuplicates().collect()

    churned_userId = []
    for u in churned_users:
        churned_userId.append(u[0])

    data_churn = data_churn.withColumn('Churned_User', data_churn.userId.isin(churned_

    total_users = data_churn.select('userId').dropDuplicates().count()
    churned_users = data_churn.filter('Churned_User = true').select('userId').dropDupl
    stayed_users = data_churn.filter('Churned_User = false').select('userId').dropDupl

    print('Of total {} users, {} users stayed with the streaming service during the ob
          .format(total_users, stayed_users, churned_users, churned_users/total_users*

    return data_churn
```

In [109]:
```
def get_platform(x):
    """
    Checks userAgent String for possible platforms that are referenced within the stri
    """
    if 'compatible' in x:
        return 'Windows'
    elif 'iPad' in x:
        return 'iPad'
    elif 'iPhone' in x:
        return 'iPhone'
    elif 'Macintosh' in x:
        return 'Mac'
    elif 'Windows NT 5.1' in x:
        return 'Windows'
    elif 'Windows NT 6.0' in x:
        return 'Windows'
    elif 'Windows NT 6.1' in x:
        return 'Windows'
    elif 'Windows NT 6.2' in x:
```

```
            return 'Windows'
        elif 'Windows NT 6.3' in x:
            return 'Windows'
        elif 'X11' in x:
            return 'Linux'
```

In [110]:
```
def get_browser(x):
    '''
    Checks userAgent String for possible browsers that are referenced within the strin
    '''
    if 'Firefox' in x:
        return 'Firefox'
    elif 'Safari' in x:
        if 'Chrome' in x:
            return 'Chrome'
        else:
            return 'Safari'
    elif 'Trident' in x:
        return 'IE'
    else:
        return np.NaN
```

In [111]:
```
def create_browser_and_platform_columns(data_churn):
    '''
    Function to create columns for the browser and platform

    args:
        data_churn (pyspark dataframe) - data outcome from 'create_cancellation_event_

    returns:
        data_churn (pyspark dataframe) - dataframe with new columns 'browser' and 'pla
    '''
    # udfs to add columns with the browser and platform
    get_browser_udf = F.udf(get_browser, StringType())
    get_platform_udf = F.udf(get_platform, StringType())

    data_churn = data_churn.withColumn( 'browser', get_browser_udf(data_churn.userAgen
    data_churn = data_churn.withColumn( 'platform', get_platform_udf(data_churn.userAg

    #dropping userAgent
    data_churn = data_churn.drop('userAgent')

    return data_churn
```

In [112]:
```
# Doing the same with location data
def get_state(x):
    """
    Splits ccolumn values on a ", " and retrieves the state entry.
```

```
            """
            return x.split(', ')[1]

        def get_city(x):
            """
            Splits column values on a ", " and retrieves the city entry.
            """
            return x.split(', ')[0]
```

In [113]:
```
def create_state_and_city_columns(data_churn):
    '''
    Function to create columns for the state and the city

    args:
        data_churn (pyspark dataframe) - data outcome from 'create_browser_and_platfor

    returns:
        data_churn (pyspark dataframe) - dataframe with new columns 'state' and 'city'
    '''
    # udfs to add columns with the state and city

    get_state_udf = F.udf(get_state, StringType())
    get_city_udf = F.udf(get_city, StringType())

    data_churn = data_churn.withColumn( 'state', get_state_udf(data_churn.location) )
    data_churn = data_churn.withColumn( 'city', get_city_udf(data_churn.location) )

    #dropping location
    data_churn = data_churn.drop('location')

    return data_churn
```

In [114]:
```
def create_column_days_since_registration_and_percentage_active_days(data_churn):
    '''
    Function to create a column that shows the days from registration until last recor
    create a column that shows the percentage of days where the user was active throug
    until last recorded event

    args:
        data_churn (pyspark dataframe) - data outcome from 'create_state_and_city_colu

    returns:
        data_churn (pyspark dataframe) - dataframe with new column 'days_from_registra
                                         'percentage_active_days'
    '''

    #Create a str datetime column for registration timestamp
    udf_convert_ts = F.udf(lambda timestamp: datetime.datetime.fromtimestamp(timestamp
```

```
    data_churn = data_churn.withColumn('dt_registration', udf_convert_ts(data_churn.re

    #Create an int column for timedifference from the event to the registration
    data_churn = data_churn.withColumn('milliseconds_since_registration', (data_churn.
    data_churn = data_churn.withColumn('days_since_registration', (data_churn.ts - dat

    #Create Column that shows for each user the difference of days for the last event
    #for the observed timeframe
    window = Window.partitionBy('userId')
    data_churn = data_churn.withColumn('days_from_registration_until_last_event', F.ma
    data_churn = data_churn.withColumn('milliseconds_from_registration_until_last_even

    data_churn = data_churn.drop('days_since_registration')

    join_df = data_churn.groupBy('userId') \
        .agg(F.countDistinct('date').alias('days_user_active'))

    data_churn = data_churn.join(join_df, on=['userId'], how='full')

    join_df.unpersist(blocking = True)

    data_churn = data_churn \
        .withColumn('percentage_active_days', 100*(data_churn.days_user_active/data_ch

    return data_churn
```

In [115]:
```
def create_column_page_events_vs_songs_listened_per_userid(data_churn, list_page_event
    '''
    Function to create a column that shows chosen page events (from the list_page_even

    args:
        data_churn (pyspark dataframe) - data outcome from 'create_column_days_since_r
        list_page_events (list containing strings) - list of strings where each string

    returns:
        data_churn (pyspark dataframe) - dataframe with new columns for each input eve
    '''

    help_df = data_churn.select(['userId', 'page', 'Churned_User']) \
        .where((data_churn.page.isin(list_page_events)) | (data_churn.page == 'NextSon
        .groupby('userId', 'page', 'Churned_User').count()

    join_df_NextSong = help_df.where(help_df.page == 'NextSong') \
        .select(['userId', 'count']) \
        .withColumnRenamed('count', 'Total_NextSong_perUser')

    data_churn = data_churn.join(join_df_NextSong, on=['userId'], how='full')
```

```
        for page_event in list_page_events:

            page_name = page_event.replace(" ", "_")

            renamed_Column = 'Total_' + page_name + '_perUser'

            join_df_event = help_df.where(help_df.page == page_event) \
                .select(['userId', 'count']) \
                .withColumnRenamed('count', renamed_Column)

            data_churn = data_churn.join(join_df_event, on=['userId'], how='full')

            data_churn = data_churn.withColumn(page_name + '_vs_NextSong', (F.col(renamed_
            data_churn = data_churn.drop(renamed_Column)

            help_df.unpersist(blocking = True)
            join_df_event.unpersist(blocking = True)

        join_df_NextSong.unpersist(blocking = True)

        return data_churn
```

In [116]:
```
def create_column_streamingtime_per_active_days(data_churn):
    '''
    Function to add a column to a dataframe that shows the hours of streaming time per

    args:
        data_churn (pyspark dataframe) - data outcome from 'create_column_page_events_

    returns:
        data_churn (pyspark dataframe) - dataframe with new column 'hours_streaming_pe
    '''
    streaming_time_df = data_churn.filter(data_churn.page == 'NextSong') \
        .groupBy('userId', 'Churned_User') \
        .agg((F.sum('length')/3600).alias('streamingTime_h')) #'length' column is in s

    data_churn = data_churn.join(streaming_time_df.select(['userId','streamingTime_h']
                                 on=['userId'], how='full')

    streaming_time_df.unpersist(blocking = True)

    data_churn = data_churn \
        .withColumn('hours_streaming_per_active_day', data_churn.streamingTime_h/data_

    return data_churn
```

In [117]:
```
def create_column_average_amount_songs_per_session_for_every_user(data_churn):
    '''
```

```
        Function to add a column to a dataframe that shows the average amount of songs pla

        args:
            data_churn (pyspark dataframe) - data outcome from 'create_column_streamingtim

        returns:
            data_churn (pyspark dataframe) - dataframe with new column 'avg_amount_songs_p
        '''
        avg_amount_songs_played_per_session = data_churn.select('userId', 'sessionId', 'Ch
            .filter('page = "NextSong"') \
            .groupby('userId', 'sessionId', 'Churned_User').count() \
            .select('userId', 'Churned_User', 'count') \
            .groupby('userId', 'Churned_User').mean() \
            .withColumnRenamed('avg(count)', 'avg_amount_songs_played_per_session')

        avg_songs_df = avg_amount_songs_played_per_session.select(['userId', 'avg_amount_s

        data_churn = data_churn.join(avg_songs_df, on=['userId'], how='full')

        avg_amount_songs_played_per_session.unpersist(blocking = True)
        avg_songs_df.unpersist(blocking = True)

        return data_churn

In [118]: def create_labeled_userIds(raw_feature_data):
        '''
        Function to convert raw_feature_data to dataframe with one line per userId.
        The last recorded event for each user is chosen to get the 'level' column as a fea

        args:
            raw_feature_data (pyspark dataframe) - raw feature dataframe from 'create_colu

        returns:
            formatted_feature_data (pyspark dataframe) - dataframe with userIds and featur
        '''

        formatted_feature_data = raw_feature_data.where(raw_feature_data.milliseconds_sinc
                                                        raw_feature_data.milliseconds_from

        formatted_feature_data = formatted_feature_data.dropDuplicates(subset=['userId', '
                                 .fillna(value=0)

        raw_feature_data.unpersist(blocking = True)

        return formatted_feature_data

In [119]: def feature_engineering_dataframe(data):
        '''
```

```
    Function to compute all functions in order and produce the formatted feature dataf

    args:
        data (pyspark dataframe) - unprepared (original) data from Sparkify

    returns:
        user_feature_data (pyspark dataframe) - dataframe with userIds and features, c
    '''
    #remove columns that are not needed for feature engineering: artist, method, song
    data = data.drop('artist', 'method', 'song', 'auth', 'firstName', 'lastName')

    data_clean = clean_data(data)
    data.unpersist(blocking = True)
    data_clean = transform_date_function(data_clean)
    data_churn = create_cancellation_event_and_churn_label(data_clean)
    data_clean.unpersist(blocking = True)

    data_feature = create_browser_and_platform_columns(data_churn)
    data_churn.unpersist(blocking = True)
    data_feature = create_state_and_city_columns(data_feature)

    data_feature = create_column_days_since_registration_and_percentage_active_days(da
    list_page_events = ['Thumbs Up', 'Thumbs Down', 'Downgrade', 'Submit Downgrade', '
    data_feature = create_column_page_events_vs_songs_listened_per_userid(data_feature
    data_feature = create_column_streamingtime_per_active_days(data_feature)
    data_feature = create_column_average_amount_songs_per_session_for_every_user(data_


    raw_feature_data = data_feature.select('ts', 'itemInSession', 'milliseconds_since_
                                           'Churned_User', 'userId', 'gender', 'level', 'br
                                          'days_from_registration_until_last_event', 'Thumb
                                          'Downgrade_vs_NextSong', 'Submit_Downgrade_vs_Nex
                                          'Submit_Upgrade_vs_NextSong', 'Roll_Advert_vs_Nex
                                          'Add_Friend_vs_NextSong','hours_streaming_per_act
                                          'percentage_active_days', 'avg_amount_songs_playe
    data_feature.unpersist(blocking = True)

    user_feature_data = create_labeled_userIds(raw_feature_data)
    raw_feature_data.unpersist(blocking = True)

    user_feature_data = user_feature_data.drop('ts', 'itemInSession', 'milliseconds_si
                                               'milliseconds_from_registration_until_l


    return user_feature_data
```

In [120]:
```
user_feature_data = feature_engineering_dataframe(data)
```

```
The dataset originally contained 286500 rows.
After cleaning there are 278154 rows left.
```

```
Of total 225 users, 173 users stayed with the streaming service during the observed time and 52
```

```
In [121]: user_feature_data.printSchema()
```

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

```
In [1]: #safe dataframe to be used in the next notebook
        #user_feature_data.coalesce(1).write.format('json').save("user_feature_data.json")
```

This data will now be the basis for the data that our models are being trained on. For the next and finals steps towards training the ML models and evaluating the results see 03_Sparkify_Modelling