Predict customer churn in a distributed way using Spark

Naveen Setia
Feb 12 · 6 min read

I got an opportunity to work on the customer churn problem as a part of the “Udacity Data Science Nanodegree program”. It was an amazing learning experience so I am sharing my learning with the broader audience.

Before we start performing the real analysis, let’s understand the few key terms which will be used in this project:

What is the customer/subscriber churn (aka Customer Attrition)?

It occurs when customers/subscribers stop doing business with a company/service. It is sometimes also known as customer attrition.

Why it is important?

- Earning money from a new customer is a result of a lengthy process which includes taking a customer through the entire sales funnel.

- Given we have already earned the trust and loyalty of the existing customer, it is much less expensive to retain an existing customer.
- As a business, we should be able to assess customer churn in a given period of time to avoid impedes in growth.

How should wedefine customer churn for this project?

I used the *Cancellation Confirmation* events to define your churn, which happens for both paid and free users.

Why Spark?

- Dataset shared as part of this project is over 12GB so it is virtually impossible to load this into a typical development machine.
- The Spark distributed analysis capability makes it relatively easy to handle these large datasets and reduce the time it takes to analyze the data and train models.

How to save costs?

- As highlighted above, it is not possible to load over 12GB in a typical local development machine. So, I used a subset of data to explore and build a model in a local development environment.
- Once it is fully tested it can be used in a cloud-based development (e.g. on Amazon AWS, Microsoft Azure, etc.) environment to run the script on full 12GB data.

What is the outcome of the project?

- As highlighted above, the business ask is to predict whether a customer will churn or not.
- From the machine learning perspective, it is a binary classification problem i.e. a binary outcome whether a customer will churn or not.

How will you decide whether the trained customer attrition model is good or not?

- Normally, for a binary classification problem, it is fine to use accuracy (i.e. no. of correctly predicted outcomes / total outcomes).

- However, in the current scenario, it is not a good measure given the provided data set is imbalanced i.e. only 22% of the records show up as positive for churn. In other words, where only a minority of cases are positive cases.
- In the case of imbalanced class, F1 is a good measure to evaluate the model.
- How to calculate the F1 measure:

$$F1 = 2 * precision * recall / (precision + recall)$$

F1 Measure Formula

- **Precision** is the total number of actual positives out of the total number of positives identified (correctly and incorrectly).
- **The recall** is the total number of positives correctly identified out of all the true positives.

Given you have some background by now, let's start with steps I followed to analyze the data:

Explore a subset of the data

- As highlighted earlier, we are using a subset of the data for faster experimentation and cost perspective.
- It works very well as long as the subset holds true for the large dataset.
- Once the script is tested locally, we can use a cloud-based spark environment for analysis on the full data set.

```
# Read in mini sparkify dataset in the local environment
event_data = "./mini_sparkify_event_data.json"
df = spark.read.json(event_data)

# Let's see the head of the data
df.head()
```

Load mini dataset into the spark session

```
print((df.count(), len(df.columns)))
```

```
(286500, 18)
```

Check the number of records and columns in the dataset

```
df.columns
```

```
['artist',
 'auth',
 'firstName',
 'gender',
 'itemInSession',
 'lastName',
 'length',
 'level',
 'location',
 'method',
 'page',
 'registration',
 'sessionId',
 'song',
 'status',
 'ts',
 'userAgent',
 'userId']
```

List of columns which are there in the dataset

As we can see, the subset of data has 2,86,500 records and 18 columns.

Create the target variable i.e. 'Churn'

- Select the relevant columns into a cleaned data frame (i.e. df_clean)
- Created a user-defined function (UDF) i.e. churned to check if the Cancelled value exists in the auths column
- Once the churn column has been created, I joined it with a cleaned data frame (i.e. df clean) based on userId.

```
# Select relevant columns
df_clean = df.select('artist','auth','firstName','gender','lastName','length','level','location','page','song','ts'
```

```
# Combine all auths in one columns
df_churn = df_clean.groupby('userId').agg(collect_list('auth').alias("auths"))
```

```
df_churn.show(1)
```

```
+--------------------+
|               auths|
+--------------------+
|[Logged In, Logge...|
+--------------------+
only showing top 1 row
```

```
# Create a custom UDF for creating the churned column
churned = udf(lambda x: 'Cancelled' in x)
```

```
# Create the churned column and drop the auths column
df_churn = df_churn.withColumn("Churned", churned(df_churn.auths))
df_churn = df_churn.drop('auths')
```

```
# Join the churn df with the clean df
df_label = df_churn.join(df_clean,'userId')
```

Step 3: Feature Engineering

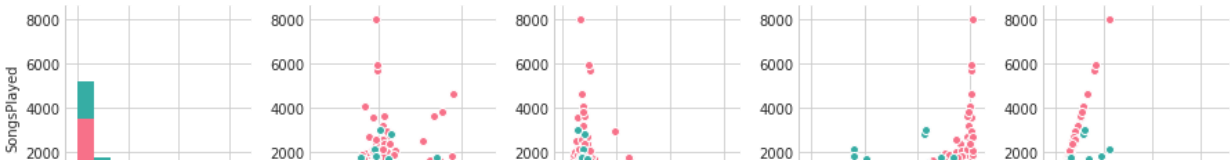
After analysis, I decided to create ***five*** features for model training:

- `Thumbs Up:` The average number of 'thumbs up' given per song
- `Thumbs Down:` The average number of 'thumbs down' given per song
- `Songs Played:` The total number of songs played
- `Number of Days:` The number of days the user had been members of the music service
- `Songs Per Hour:` How many songs in an hour was played

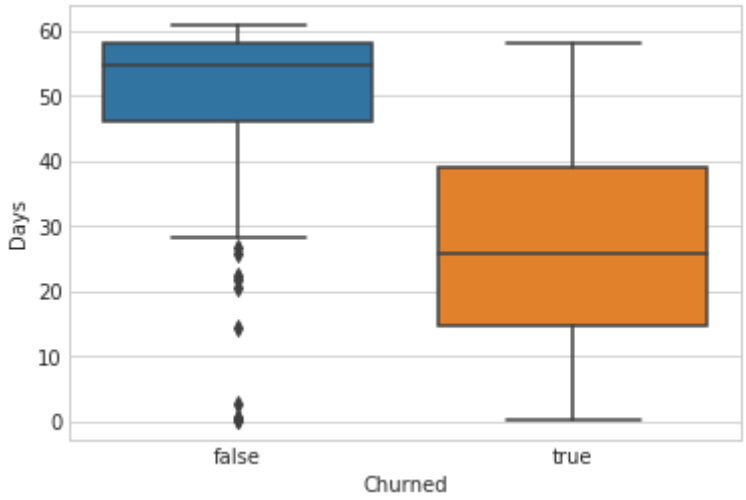
Code for calculating features:

```
songs_played = df_label.where(col('song')!='null').groupby("userId").agg(count(col('song')).alias('SongsPlayed')).orderBy('userId')

thumbs_up = df_label.where(df_label.page=='Thumbs Up').groupby("userId").agg(count(col('page')).alias('ThumbsUp')).orderBy('userId')

thumbs_down = df_label.where(df_label.page=='Thumbs Down').groupby("userId").agg(count(col('page')).alias('ThumbsDown')).orderBy('userId')

days = df_label.groupby('userId').agg(((max(col('ts')) - min(col('ts')))/86400000).alias("Days"))

udf_songs_played_per_hour = udf(lambda songsPlayed, numberOfDays: float(songsPlayed)/float((numberOfDays*24)), FloatType())

df_features = df_features.withColumn("SongsPerHour", udf_songs_played_per_hour(df_features.SongsPlayed, df_features.Days))
```

Visualisation of Features

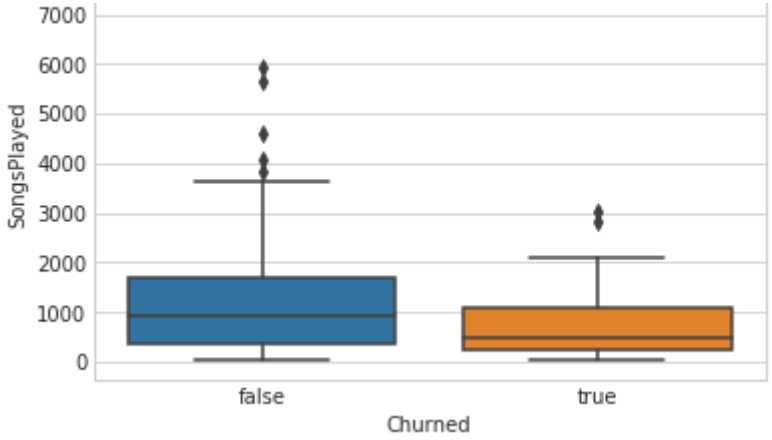


PairPlot of Important Features



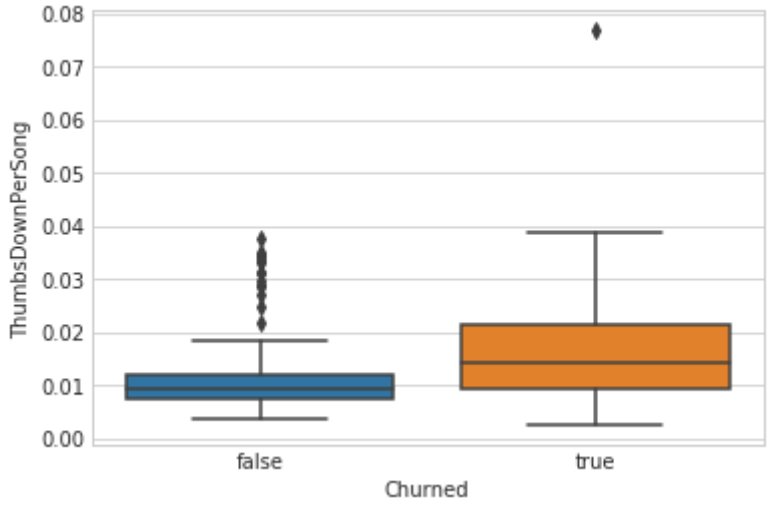
Box Blot for Days

It's evident from the above graph that users who uses the service for a shorter time are more likely to churn compared to continuing users.



Box Plot for SongsPlayed

It's evident from the above graph that users has played less songs is more likely to churn compared to use who has played more songs.



Box Plot for ThumbsDownPerSong

It's evident from the above graph that song which has got more thumbs down is more likely to churn compared to user who has less thumbs down.

Join all features for model training in a single vector and standardise it

```
assembler = VectorAssembler(inputCols=["SongsPlayed", "ThumbsUpPerSong",
"ThumbsDownPerSong", "Days", "SongsPerHour"], outputCol="FeatureVector")
df_features = assembler.transform(df_features)

scaler = StandardScaler(inputCol="FeatureVector", outputCol="ScaledFeatures",
withStd=True)
scaler_tranformer = scaler.fit(df_features)
df_features = scaler_tranformer.transform(df_features)
```

Split the data into train, test and validation

```
# Split the data into train, test and validation
train_ratio = 0.8
test_ratio = 0.2
validation_ratio = 0.2
train, test = df_features.randomSplit([train_ratio, test_ratio], seed=9999)
train, validation = train.randomSplit([(1 - validation_ratio), validation_ratio], seed=9999)
```

Split the data into train, test and validation

Train the model

> *model = RandomForestClassifier(featuresCol = 'FeatureVector', labelCol = 'label', numTrees=100)*
>
> *trained_model = model.fit(train)*

Evaluate the Performance of the Model

```
def evaluate_performance(trained_model,train,validation,test,evaluator):
    # Test the performance via evaluator on training data
    predictions = trained_model.transform(train)
    print('Train: Area Under ROC', evaluator.setMetricName("areaUnderROC").evaluate(predictions))
    print('Train: Area Under PR', evaluator.setMetricName("areaUnderPR").evaluate(predictions))

    # Test the performance via evaluator on validation data
    predictions = trained_model.transform(validation)
    print('Validation: Area Under ROC', evaluator.setMetricName("areaUnderROC").evaluate(predictions))
    print('Validation: Area Under PR', evaluator.setMetricName("areaUnderPR").evaluate(predictions))

    # Test the performance via evaluator on test data
    predictions = trained_model.transform(test)
    print('Area Under ROC', evaluator.setMetricName("areaUnderROC").evaluate(predictions))
    print('Area Under PR', evaluator.setMetricName("areaUnderPR").evaluate(predictions))
```

Function to evaluate the performance

```
evaluate_performance(trained_model,train,validation,test,evaluator)
```

Function call to evaluate the performance

```
Train: Area Under ROC 0.9985119047619048
Train: Area Under PR 0.9942257534428728
Validation: Area Under ROC 0.9523809523809523
Validation: Area Under PR 0.9027777777777777
Test: Area Under ROC 0.8428030303030303
Test: Area Under PR 0.7930118994150303
```

Performance evaluation of the model on training, validation and test data

Model training using CrossValidator

```
# Run through a cross validator
param_grid = ParamGridBuilder().addGrid(model.regParam, [0.1, 0.01]).build()
```

```
    if type(model).__name__ == 'LogisticRegression' else
    ParamGridBuilder().addGrid(model.numTrees, [2, 5, 10]).build()

crossval = CrossValidator(estimator=model,
                          estimatorParamMaps=param_grid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=3)
trained_model = crossval.fit(train)
```

Function to evaluate the performance

```
evaluate_performance(trained_model,train,validation,test,evaluator)
```

Function call to evaluate the performance

```
Train: Area Under ROC 0.9985119047619048
Train: Area Under PR 0.9942257534428728
Validation: Area Under ROC 0.9523809523809523
Validation: Area Under PR 0.9027777777777777
Test: Area Under ROC 0.8428030303030303
Test: Area Under PR 0.7930118994150303
```

Performance evaluation of the model on training, validation and test data using CrossValidator

Summary:

- I experimented with both LogisticRegression and RandomForestClassifier but RandomForestClassifier performed relatively better so I have shared the code for that.
- The difference in the performance of train and test data shows there is overfitting to some extent but it can be optimized with the help of more hyper parameter tuning.
- It seems we didn't gain much using CrossValidator. However, with more experimentation and hyper parameter tuning it can be perform better.

Scope of improvement:

- Current performance may be improved by using XGBoost and hyper parameter tuning.
- Adding more features e.g. Gender, days since last login, daily session time etc.

Reference(s)/Credit(s):

- https://en.wikipedia.org/wiki/Customer_attrition

- https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier
- https://spark.apache.org/docs/latest/ml-classification-regression.html#binomial-logistic-regression