CS724 High Performance Computing and Big Data

Course Project Report

Customer Churn Prediction using PySpark

Herambeshwar Pendyala | 01130541

Contents

1. Objective

The primary objective of the project is to build a scalable model to predict the customer churn for a music streaming service, given features such as user activity on the website, events, user history and other user related data.

2. Abstract

Sparkify is a fictitious music streaming service, created by Udacity to resemble the real-world datasets generated by companies such as Spotify or Pandora. Millions of users play their favorite songs through music streaming services daily, either through a free tier plan that plays advertisements, or by using a premium subscription model, which offers additional functionalities and is typically ad-free. Users can upgrade or downgrade their subscription plan any time, but also cancel it altogether, so it is very important to make sure they like the service.

Every time a user interacts with a music streaming app, whether playing songs, adding them to playlists, rating them with a thumbs up or down, adding a friend, logging in or out, changing settings, etc., data is generated. User activity logs contain key insights for helping the businesses understand whether their users are happy with the service.

In order to stay on track with its financial goals, it is key for a music streaming business to identify users who are likely to churn, i.e. users who are at risk of downgrading from premium to free tier or cancelling the service. If a music streaming business accurately identifies such users in advance, they can offer them discounts or other similar incentives and save millions in revenues. It is a well-known fact that it is more expensive to acquire a new customer than it is to retain an existing one.

3. Description

3.1. About Dataset

Sparkify, a fictional Music Streaming Service similar to Spotify, by Udacity,

source: http://udacitydsnd.s3.amazonaws.com/

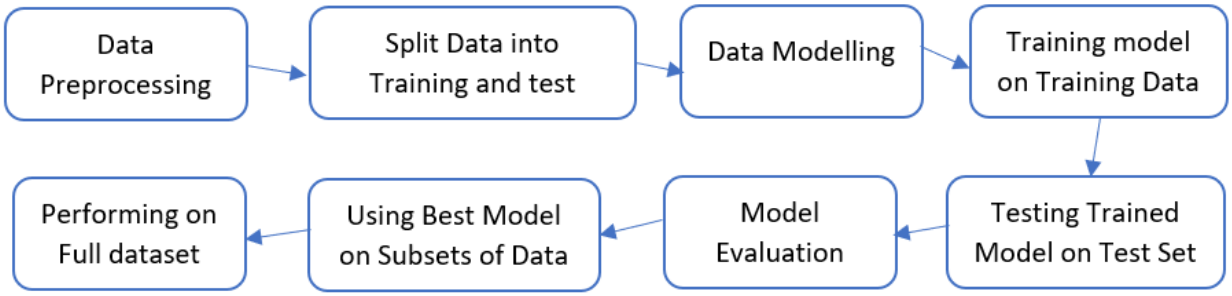
The data given is that of user events. Every interaction of every user with the application is provided. This means every time a user goes to the Home page, listens to a song, thumbs up a song, etc. we have an event in the data corresponding to the same. Each record in our data represents an event by a user. This makes the data huge as the application grows and makes tedious to process it. So I have used pyspark sql dataframe to work on this data in the preprocessing stage. Below are the columns in the dataset.

```
Root
 |-- artist: string (nullable = true)
 |-- auth: string (nullable = true)
 |-- firstName: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- itemInSession: long (nullable = true)
 |-- lastName: string (nullable = true)
 |-- length: double (nullable = true)
 |-- level: string (nullable = true)
 |-- location: string (nullable = true)
 |-- method: string (nullable = true)
 |-- page: string (nullable = true)
 |-- registration: long (nullable = true)
 |-- sessionId: long (nullable = true)
 |-- song: string (nullable = true)
 |-- status: long (nullable = true)
 |-- ts: long (nullable = true)
 |-- userAgent: string (nullable = true)
 |-- userId: string (nullable = true)
```

4. Design

For model development, most of the steps such as data understanding, feature engineering and model selection, were performed on a representative sample (1/100) of the full dataset, using Spark in local mode. Models that performed well on the smaller sample were trained and tested also other iterations of the dataset as well as on the full dataset.

Below is the flow chart showing the flow of the development process.



5. Implementation

5.1. Data Cleaning, Preprocessing and Exploratory Data Analysis

To transform the original dataset which has one row per user log to a dataset with user-level information or statistics one row per user, I have used dataframe from sparkSQL to represent the data in a table format similar to relational database which makes it easy to do data analysis. The final dataset is obtained through mapping (e.g. user's gender, start/end of the observation period, etc.) or aggregation (e.g. song count, advertisement count, etc.) using the functions of spark sql component.

Using SparkSQL dataframe for data preprocessing as follows,

I started with data cleaning as there are missing values in the columns userId and sessionId. I need at least the userId to feed the machine learning algorithm with user-specific input features and labels. So, as there is no possibility to keep these rows e.g. through imputing the values for IDs, I decided to drop the associated rows from the dataset. Below are the functions I have used for preprocessing

Udf – spark provides us with udf to write scalable functions

Window - we use this function to perform a calculation over a set of rows.

Defining Churn

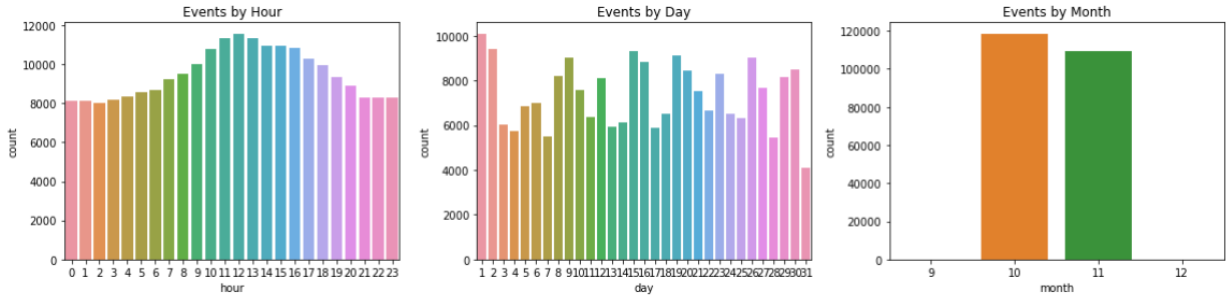
As no label specifically as 'churn' was given to predict, In this application, the label is defined by the Cancellation Confirmation or downgrade events to identify user churn, which happen for both paid and free users. All the features need to be engineered to identify churned users, e.g. aggregated statistic per unit of time, number of plan changes, songs played vs. total activity ratio, thumbs up vs. thumbs down ratio, activity trend, etc. Once churn is defined churn, I will further perform some exploratory data analysis to observe the behavior for users who stayed vs users who churned.

In the user activity log, column 'Page' stores the user activity on pages he visited for a session, I have used this column to see if the user has visited the cancellation confirmation page or downgrade page. If the user visited these pages, then they are likely to be considered as churned.

Defining and calculating the churn, I consider this as binary variable: 1 — users who cancelled their subscription, and 0 — users who kept the service throughout a time period.

Exploring the timestamp column to get insights about user behavior.

Timestamp ‘ts’ is one of the important columns in user event data as it gives us more insights about the user like daily stats. Below is the graph showing daily stats of the users went to ‘nextsong’ page in the months of October and November of 2018.



By extensively exploring this data by aggregation of pages like ‘nextSong’ on different days we can get features such as consecutive_listening_days, days_since_last_listen, total_listens.

I also took a look at user behavior from other pages like `*thumbs-up*`, specifying a like for the song, which can lead the user to add a song to a playlist, where `*add-to-playlist*` is another page. `*thumbs-down*`, specifying the dislike for a song, `*add-friends*` to the profile and share playlists and songs with other users, `*error*` page which specifies how often a user is not able to find his needs in the application. Features were computed by aggregating the page visits of the users to these pages.

From the above observations, the features calculated are consecutive_listening_days, days_since_last_listen, total_listens, total_thumbs_up, total_thumbs_down, total_errors, total_add_to_playlists, total_add_friends.

Exploring user session data

Another potentially useful indicator is the extent to which users behave within each session. Starting by taking the total sum of each flag behavior calculated above (listens, likes, dislikes, friends and playlists). Then taking the average over all each user's session to get a sense of how a user tends to behave in one session. Below image displays the result of those aggregations.

userId	avg_sess_listens	avg_sess_thU	avg_sess_thD	avg_sess_err	avg_sess_addP	avg_sess_addF
100010	39.285714285714285	2.4285714285714284	0.7142857142857143	0.0	1.0	0.5714285714285714
200002	64.5	3.5	1.0	0.0	1.3333333333333333	0.6666666666666666
125	8.0	0.0	0.0	0.0	0.0	0.0
51	211.1	10.0	2.1	0.1	5.2	2.8
124	140.6551724137931	5.896551724137931	1.4137931034482758	0.20689655172413793	4.068965517241379	2.5517241379310347
7	21.428571428571427	1.0	0.14285714285714285	0.14285714285714285	0.7142857142857143	0.14285714285714285
54	76.78378378378379	4.405405405405405	0.7837837837837838	0.02702702702702703	1.945945945945946	0.8918918918918919
15	127.6	5.4	0.9333333333333333	0.13333333333333333	3.933333333333333	2.066666666666667

This brings our features list to 15 which include gender, consecutive_listening_days, days_since_last_listen, total_listens, total_thumbs_up, total_thumbs_down, total_errors, total_add_to_playlists, total_add_friends. avg_sess_listens, avg_sess_thU, avg_sess_thD, avg_sess_err, avg_sess_addP, avg_sess_addF. The features are aggregations of user daily stats and session data.

I have performed exploratory data analysis, comparing the engineered statistics for users who stayed v s users who churned. Below are the graphs for the same.

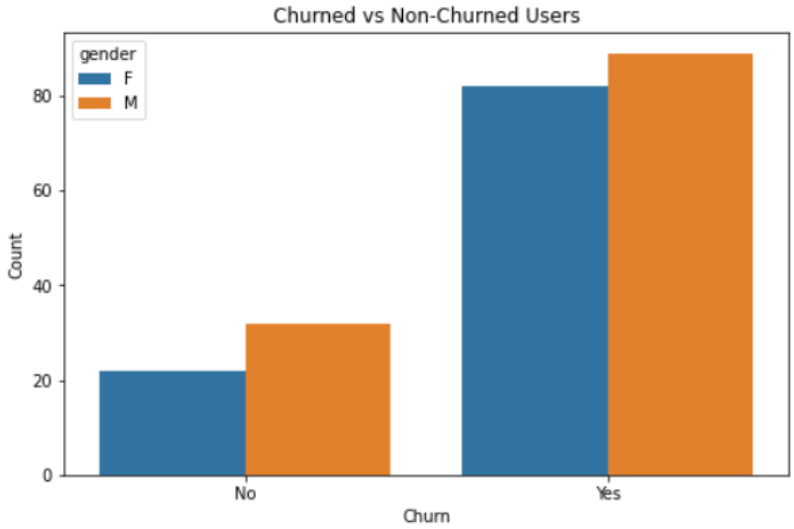


Figure: Churned vs Non-Churned users based on Gender

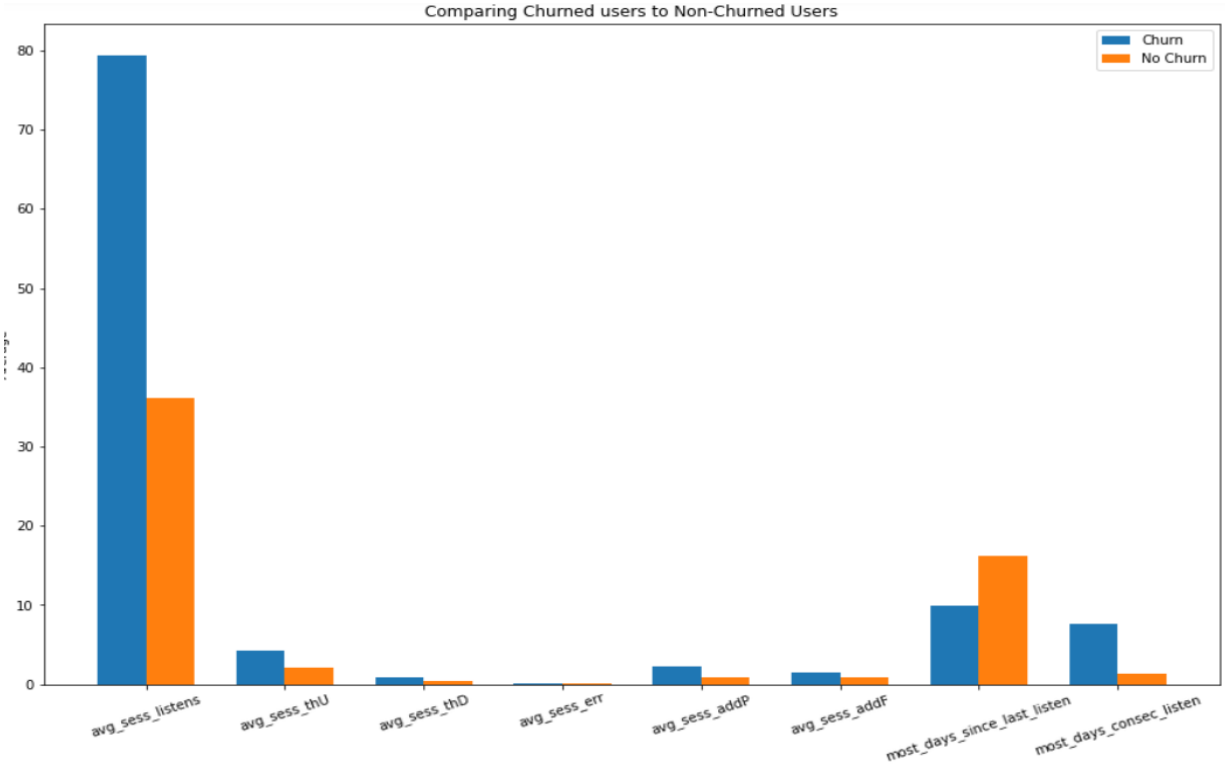


Figure: Scalability – churned vs non churned on session features.

From the preprocessing I have reduced the number of rows from 286500 to 225. Now our data has onl y one user data per row with all the engineered features. Using this data, I built a prediction model to p redict the customer churn. Steps and functions used in model building are described in the next section .

churn	count(1)
1	171
0	54

5.2. Model Building

As this problem is termed as a classification problem, the data is split into a training set and a test set. I am using Logistic regression to build prediction model and measure the performance of the algorithm on the dataset of different sizes.

Using PySpark for machine learning,

Logistic Regression

Mathematical expression of the algorithm**:**

For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)}$$
$$\widehat{y^{(i)}} = a^{(i)} = sigmoid\left(z^{(i)}\right)$$
$$\mathcal{L}\left(a^{(i)}, y^{(i)}\right) = -y^{(i)} \log\left(a^{(i)}\right) - \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(a^{(i)}, y^{(i)}\right)$$

To minimize the cost(*J*), I used gradient descent as an optimization method, the goal is to learn *w* and *b* by minimizing the cost function *J*. For a parameter θ, the update rule is $\theta = \theta - \alpha\, d\theta$, where α is the learning rate.

Using the above mathematical expressions, below is the pseudocode using RDD MapReduce to perform gradient descent.

```
For given number of iterations :
    Gradient = dataRDD.map(calculate cost).reduce(add)
    # update weights
   W = W - α * Gradient
```

After it learn the parameters, they are used to predict the labels for the dataset, by using below equation.

$$\hat{Y} = A = \sigma(w^T X)$$

Convert the entries of ***A*** into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector ***Y_prediction***.

5.3. Model Evaluation

I have used below performance metrics to evaluate my model.

Confusion Matrix

Confusion matrix is calculated from comparing true values with the predicted values. This gives us four values namely.

true positives(tp) – count of values that are true and predicted true

false positives(fp) – count of values that are false but predicted true

false negatives(fn) – count of values that are true but predicted false

true negatives(tn) – count of values that are false and predicted false

below is the confusion matrix calculated for the test dataset.

	Actual Positive	Actual Negative
Predicted True	23	0
Predicted False	14	14

Accuracy

Accuracy tells us how well out model performs on the test dataset. Below is the formula for accuracy.

$$\text{Accuracy} = TP+TN/(TP+FP+FN+TN)$$

Precision

Ration of correctly predicted positive observations to the total predicted positive observations.

$$\text{Precision} = TP/TP+FP$$

Recall

Ratio of correctly predicted positive observations to total predicted positive observations.

$$\text{Recall} = TP/TP+FN$$

F1 – Score

This is weighted average of precision and recall.

$$\text{F1 score} = 2*(\text{Precision}*\text{Recall})/(\text{Precision} + \text{Recall})$$

6. Results and Discussion

6.1. Scalability Study

To test the scalability of each model, I tested it on a small dataset with 286K rows, then on consecutive large datasets of 543K, 1087K, 2174K rows. Following is the table shows the execution time to process and model on different sizes of dataset.

As the dataset is composed of event logs, most of the data is aggregated during preprocessing and feature engineering stages reducing the number of rows to values shown in the below table based on the userIDs.

Table 1: preprocessing, modelling and total execution time for different sets of data

Data				Time for 100 iterations		
Name	DataSize	DataSize Rows	Rows Retrieved	PreProcess	Modelling	Total
Set1	128 mb	286500	225	46.87	28.11	74.98
Set2	250 mb	543705	448	63.29	29.06	92.35
Set3	490 mb	1087410	8279	113.91	37.08	150.99
set4	980 mb	2174820	10620	170.16	48.31	218.47
set5	1.9 gb	4349640	12679	185.71	47.24	232.95
set6	3.9 gb	8699280	14142	298.03	50.401	348.431
Set7	7.9 gb	17398560	21069	479.82	57.55	537.37
Set8	12 gb	25259199	22216	700.05	60.78	760.83

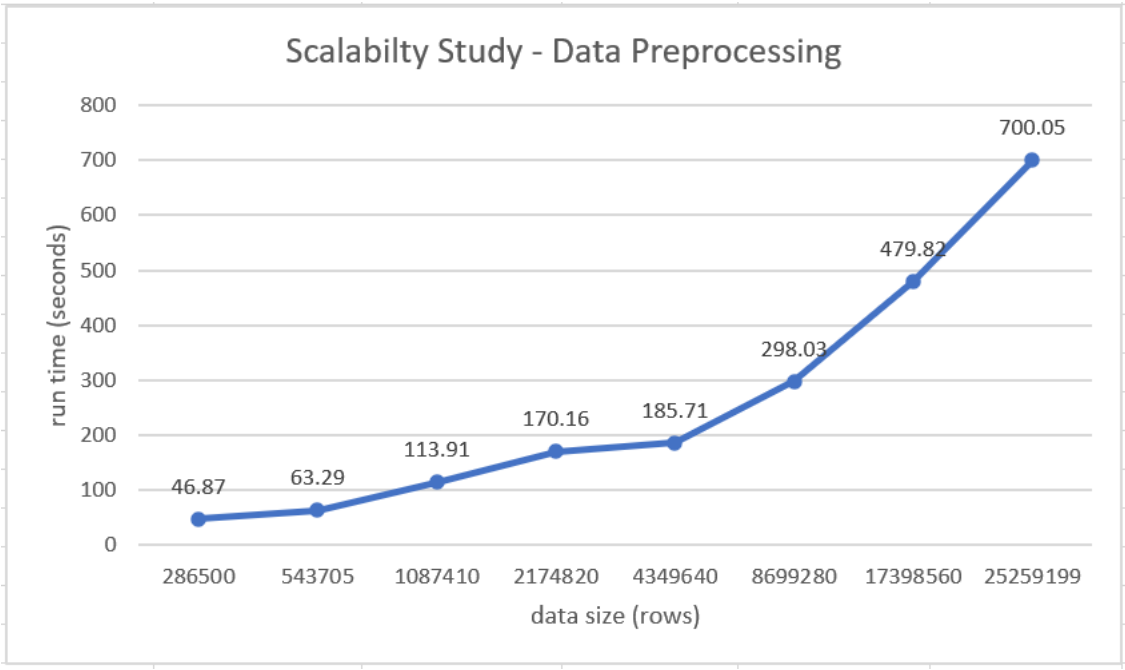


Figure: Scalability – Execution Time of Preprocessing stage for different data sizes

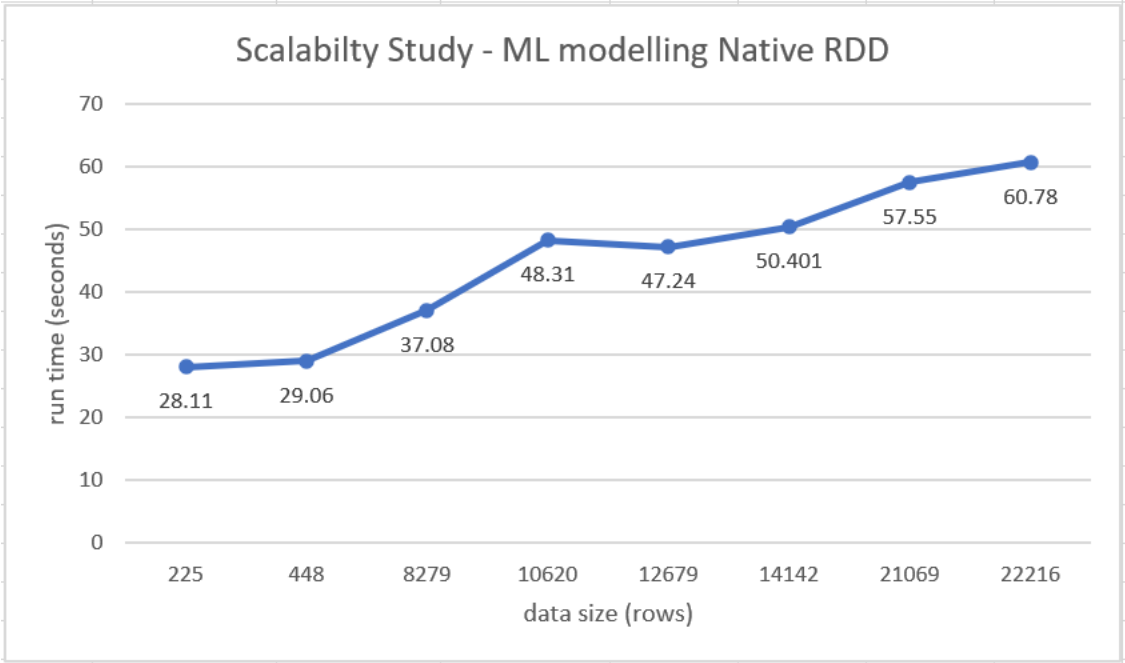


Figure: Scalability – Execution Time of Native logistic implementation stage for different data sizes

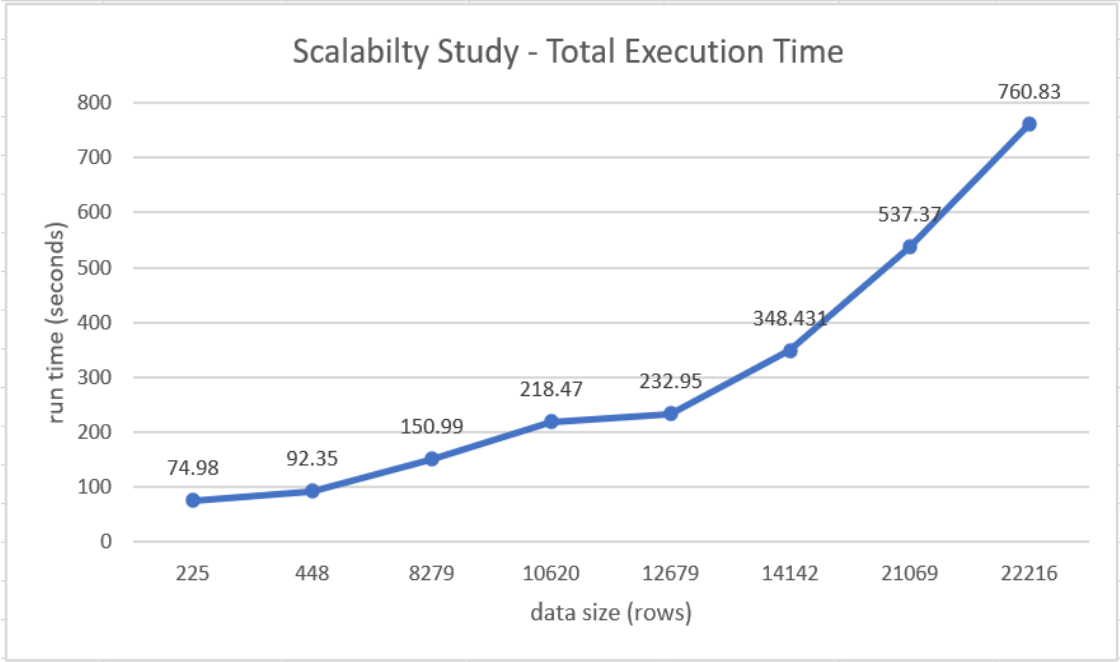


Figure: Scalability – Execution Time of preprocessing and modelling combined for different data sizes

6.2. Model Evaluation

To evaluate our model, I have compared the execution time, accuracy, f1-score of native rdd implementation with mllib library version (LogisticRegressionWithLBFGS) on different sets of data. Below tables and graphs represent the analysis of results.

Table 2: Execution Time for native and mllib version of logistic regression.

MLLIB vs custom	Execution Time(Seconds)	
Rows	Native	Mllib
225	28.11	110.84
448	29.06	116.09
8279	37.08	126.89
10620	48.31	180.29
12679	47.24	170.94
14142	50.401	185.3
21069	57.55	200.22
22216	60.78	184.07

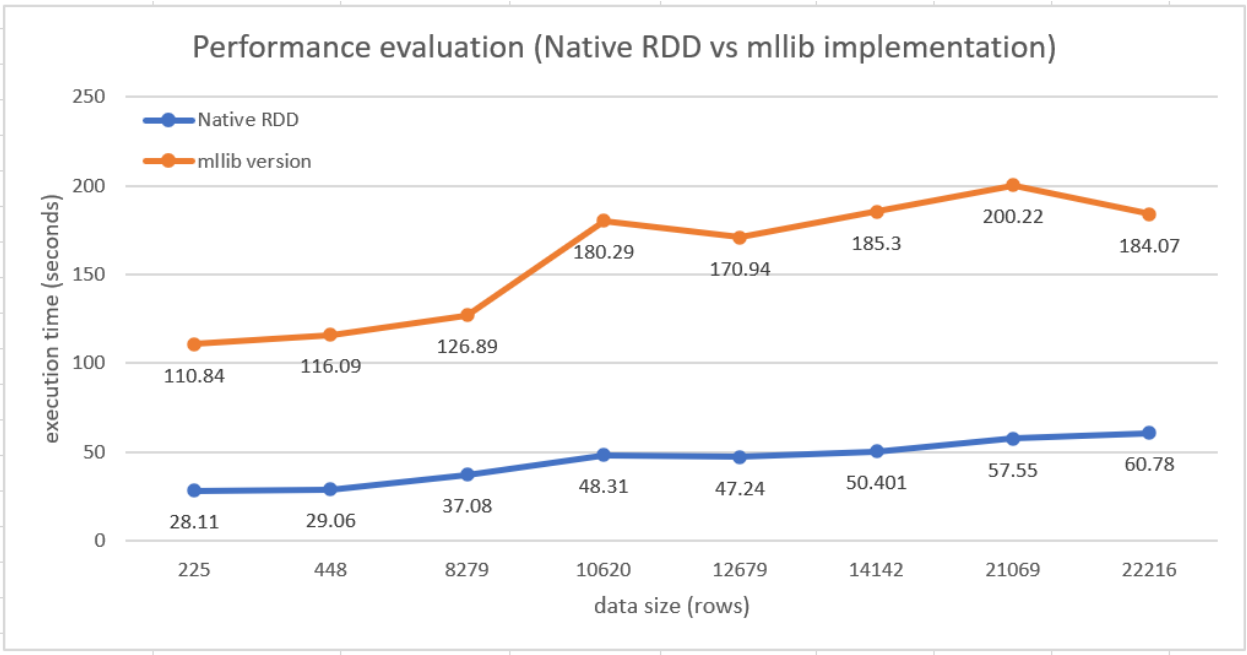


Figure: Performance evaluation – Execution Time comparison for Native RDD vs mllib on different data sizes

Table3: Scalability – Evaluation metrics comparison between native RDD vs mllib version on different data sizes

MLLIB vs custom	Native Implementation		Mllib	
Rows	Test -Accuracy	F1-Score	Accuracy	F1-Score
225	74.5	0.8433	47.05	0.4705
448	81.5	0.8793	29.06	0.536
8279	60.07	0.538	50.55	0.5055
10620	70.17	0.2964	51.42	0.5142
12679	63.9	0.837	49.89	0.4989
14142	72.22	0.813	51.15	0.5115
21069	67.32	0.805	52.61	0.5261
22216	77.58	0.837	52.26	0.5226

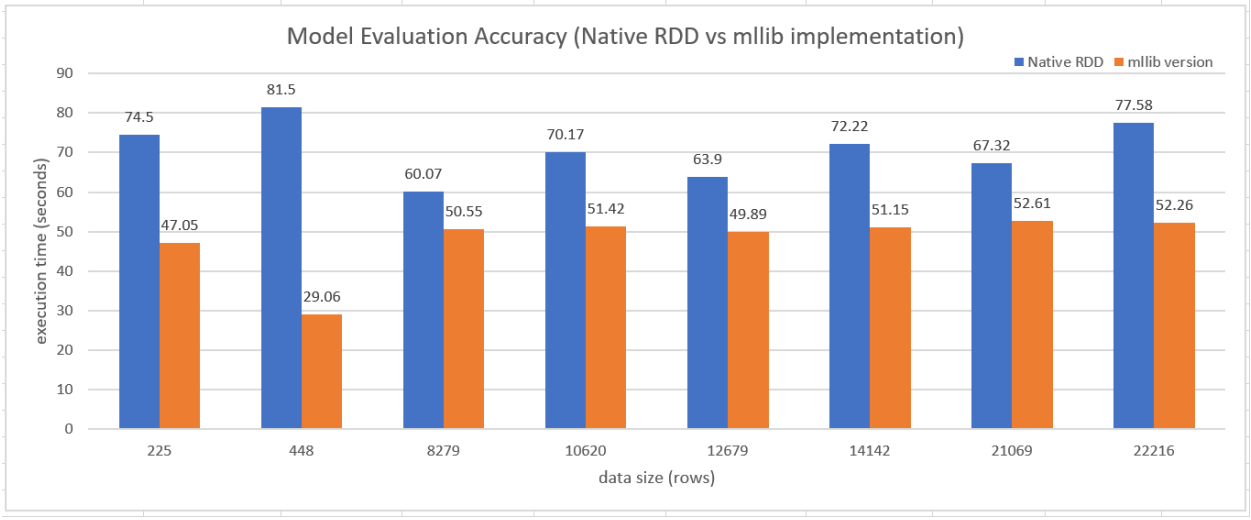


Figure: Evaluation – Accuracy of Native rdd vs mllib version on different data sizes.

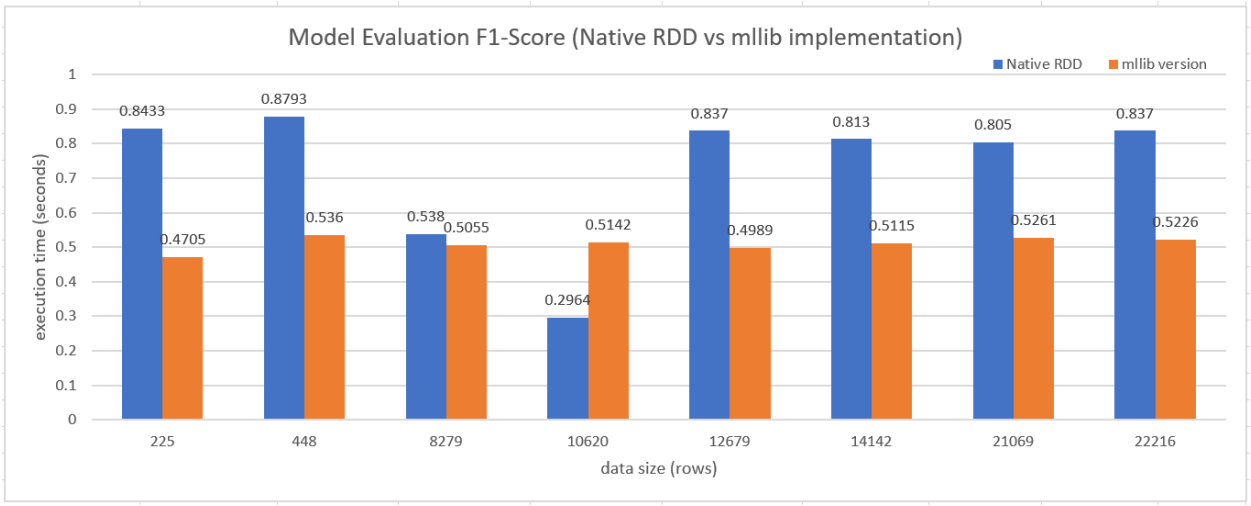


Figure: Evaluation – F1-Score of Native rdd vs mllib version on different data sizes.

Based on the above results it looks like the native rdd implementation of logistic regression performs better than the library version, but this is too good to be true. Maybe I need to do more extensive analysis of the results to get more insights about the implementation.

7. Learning Outcomes

- Learned about various spark sql functions like udf, window and their usage in big data analysis.
- While executing on larger datasets, ran into some issues as we are working with memory. It took more time to figure out these issues.

8. Future Enhancements

As part of this project, I implemented only Logistic regression to classify the churned customers, I intend to Implement other classification algorithms such as Decision tree classifier and random forest, that could improve the accuracy score of the model. Implementing the above methods can be a good learning curve

as these methods are memory extensive and writing an efficient code to manage memory is necessary. Also, during feature engineering I can use PCA to validate the important features.

9. References

[1]. Dataset info, http://udacity-dsnd.s3.amazonaws.com/

[2]. Spark Window Function, https://knockdata.github.io/spark-window-function/

[3]. Interacting with Hadoop hdfs using python, https://community.cloudera.com/t5/Community-Articles/Interacting-with-Hadoop-HDFS-using-Python-codes/ta-p/245163

[4]. Predicting Customer churn using pyspark, https://medium.com/analytics-vidhya/predicting-user-churn-with-pyspark-part-1-f13befbf04c3

[5]. Pyspark udf, https://medium.com/@QuantumBlack/spark-udf-deep-insights-in-performance-f0a95a4d8c62

[6]. Trick to normalize the exponent calculation, https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

[7]. Normalizing exponent calculation, https://stackoverflow.com/questions/26218617/runtimewarning-overflow-encountered-in-exp-in-computing-the-logistic-function

[8]. Setting up spark cluster on AWS emr, https://www.perfectlyrandom.org/2018/08/11/setup-spark-cluster-on-aws-emr/

[9]. publicKey to log on to cluster, https://stackoverflow.com/questions/41563973/git-clone-key-load-public-invalid-format-permission-denied-publickey

[10]. Setting up pyspark on local machine, https://towardsdatascience.com/how-to-use-pyspark-on-your-computer-9c7180075617.