KKBOX Churn Prediction - Final Report

Objective: Predicting whether a user will churn after the subscription expires.

Evaluation: The evaluation metric for this project is Log Loss

$$\text{log-loss}=\frac{-1}{N}\sum_{i=1}^{N}(yi\log(pi)+(1-\ yi\ )\log(1-\ pi))$$

where N is the number of observations, *log* is the natural logarithm, y_i is the binary target, and p_i is the predicted probability that y_i equals 1.

Client: KKBOX is Asia's leading music streaming service, holding the world's most comprehensive Asia-Pop music library with over 30 million tracks, supported by advertising and paid subscriptions. Currently, the company uses survival analysis techniques to determine the residual membership life time for each subscriber. By adopting different methods, KKBOX anticipates they'll discover new insights to why users leave so they can be proactive in retaining users.

Data: For this analysis, I will be using the KKBox's churn prediction dataset which was publicly available on Kaggle. Data is distributed across 4 different csv files as follows:

train_v2csv: The train set, containing the user ids and whether they have churned.
transactions.csv: The transaction set contains all the payment details till feb-2017.
transactions_v2csv: The transaction set contains all the payment details of march-2017.
members.csv: The members set contains the user information.
user_logs.csv: The user logs set contains the daily user logs describing listening behaviors of a user for the month march-2017.

Data Wrangling: All the .csv files were imported & converted as data frames. Then we concatenated the two transactions data frames ('transactions_1' & 'transactions_2') making sure we have all the transaction details in one data frame 'transactions'.

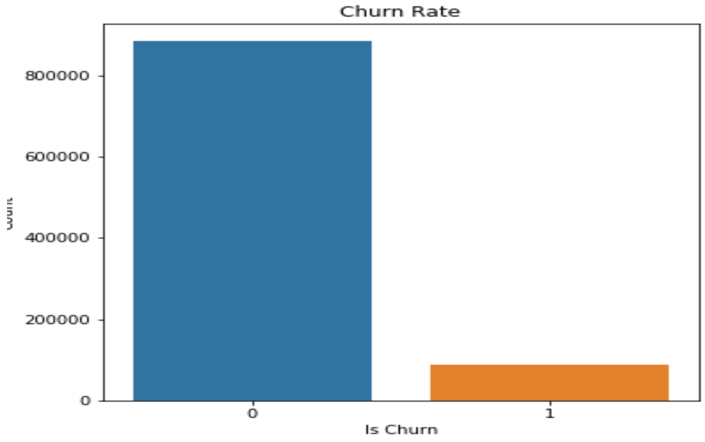
Memory Reduction: Because all the data frames are very large in size and are using a lot of memory, we performed memory reduction to reduce the memory usage by changing the data types of some columns and splitting the date column into three different columns - year, month & day columns respectively. Finally, we made sure none of the data is stripped from the original data frames and then we dropped the date columns. This helped us to reduce the memory usage to half of its usage than before.

Members data set memory-usage has been reduced from 310 MB to 155 MB.
Transactions data set memory-usage has been reduced from 1756 MB to 723 MB.
User-logs data set memory-usage has been reduced from 1265 MB to 597 MB.
Train data set memory-usage has been reduced from 15 MB to 8 MB.

We have reduced the memory of each file to its half making it convenient for the later analysis.

Data Exploration in Train

Churn Percentage: In the train data frame, we have the 'msno' – which are unique id no's for given to each customer & their churn detail are present.

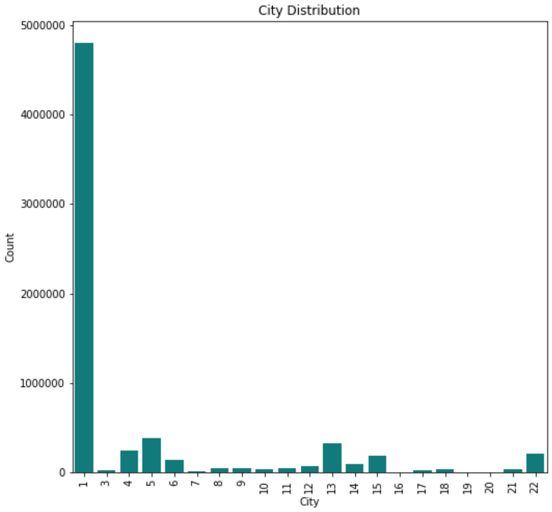


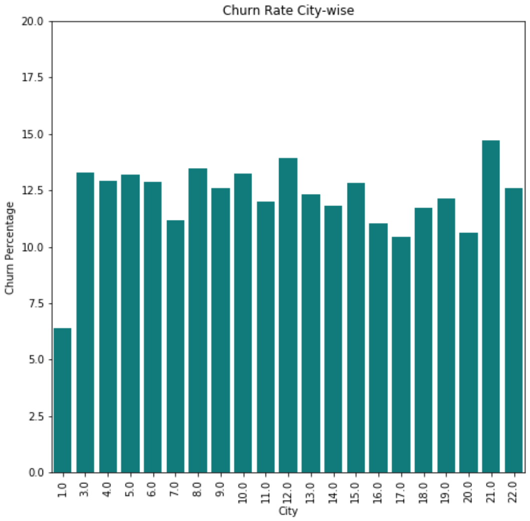
Only 9% people have churned which looks so successful, making it a highly imbalanced classification problem.

Data Exploration in Members & Train

Here we're merging train & members data sets using left join. After the merge, because not every member is present in the members data set some null values are observed.

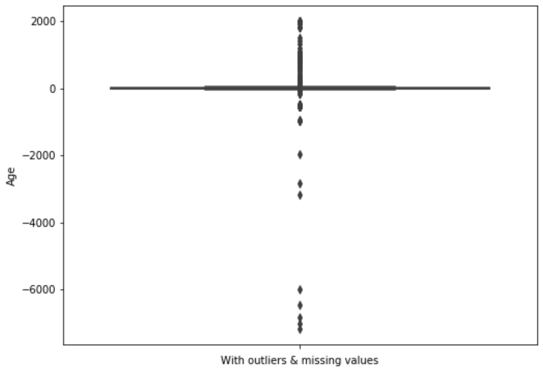
City:

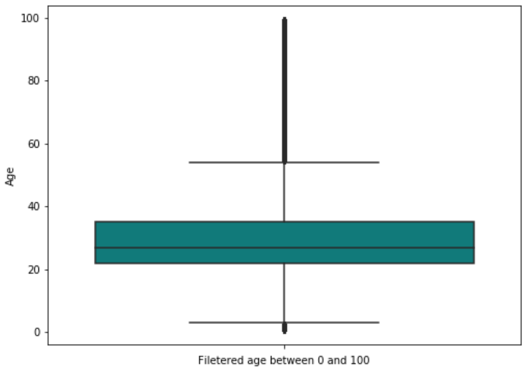




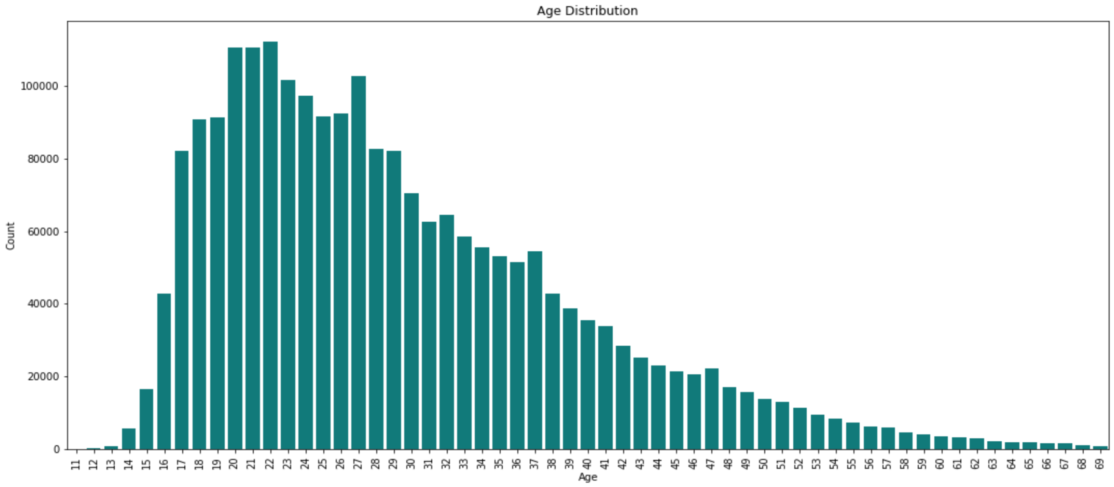
- There are total of 21 cities, there is no city '2'. We observe majority from city 1. Everything else looks similarly unpopular.
- The cities are quite similar at churn rates with the crucial exception of city 1. In this most popular city, the churn rate is significantly lower when compared to other cities. This has a big impact on the overall churn rate.

bd (age column):

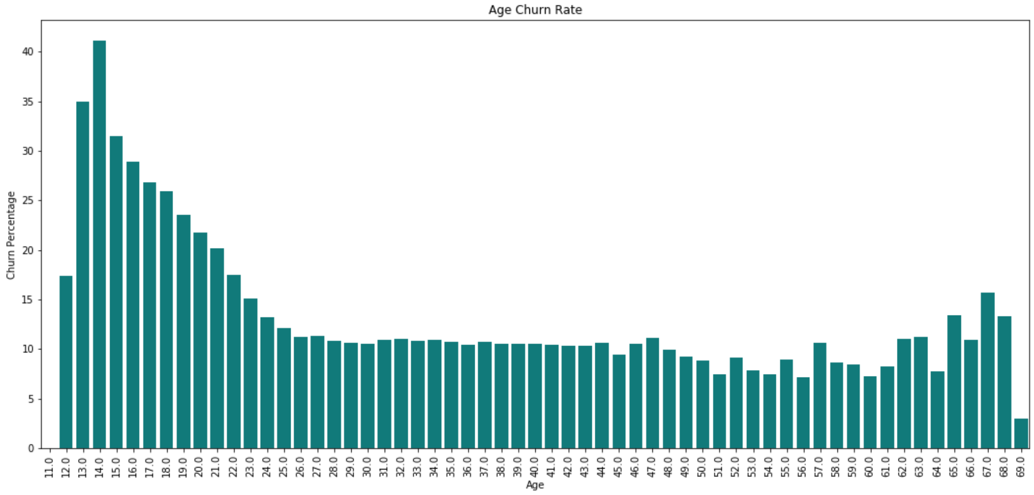




- In the bd (Age) column we observed it has lot of values set to 0 and there are some outliers ranging from ‘-7168’ to ‘2016’.

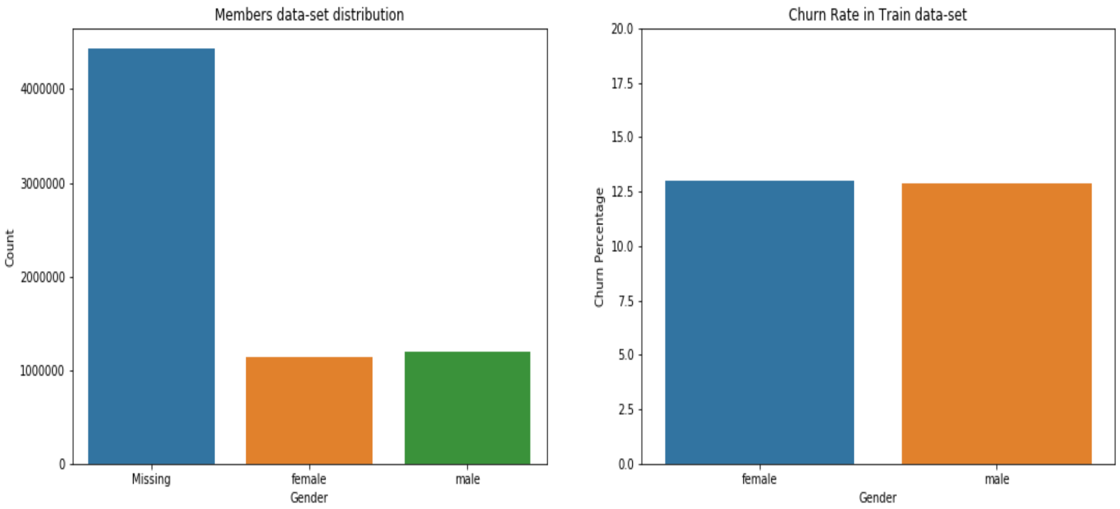


- Later we filtered the bd (Age) column between 10 and 70 and observed that most of the customers are aged between mid-teens to mid 40's.



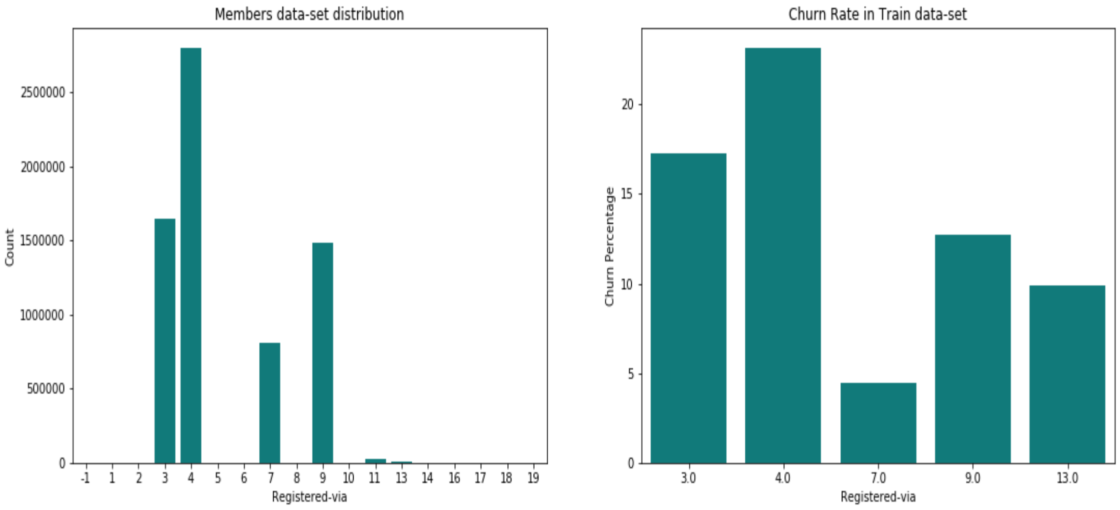
- we find that younger users on average appear to be more likely to churn.

Gender:



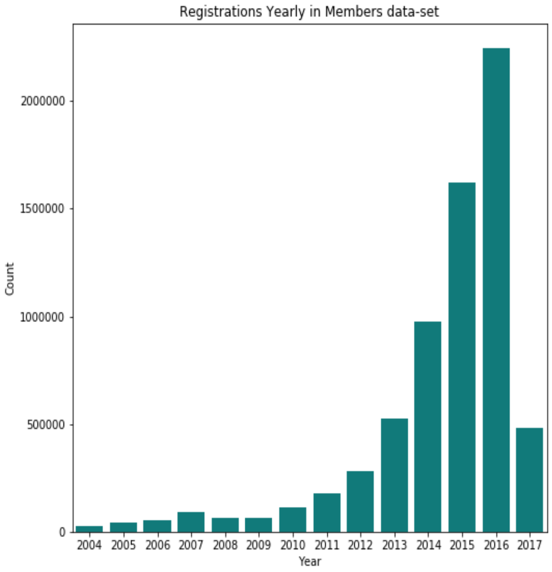
- Around 60% of the data is missing after the merge. With the data we have it seems both male and female are churning quite similar. We have to see how to deal with the missing values in future analysis.

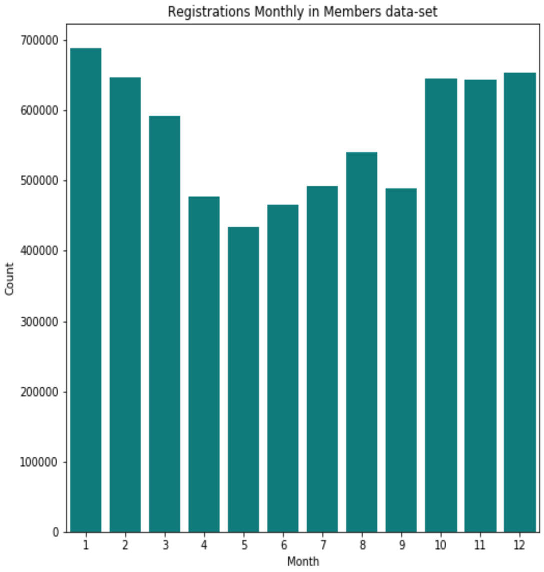
Registered-via:



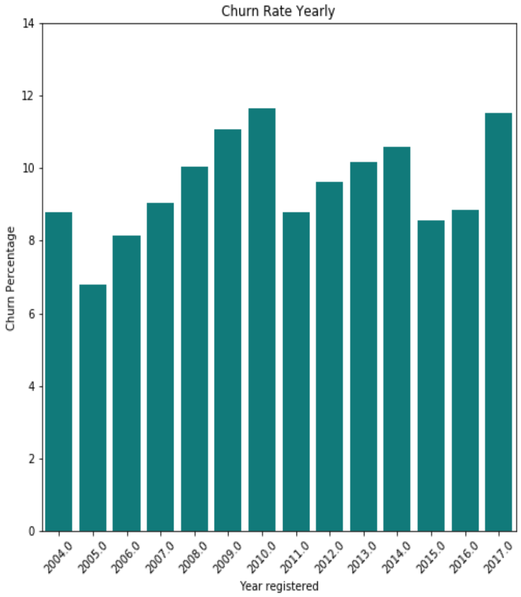
- There are 5 classes ('3', '4', '7', '9', '13') listed as registration method in x1. There are also some additional classes in the members dataset. As we merged the train and members they are missing. Also, there are noticeable differences in terms of registration method. Method ‘7’ appears to correlated with the most loyal users, while method ‘4’ has slightly higher churn rate of all.

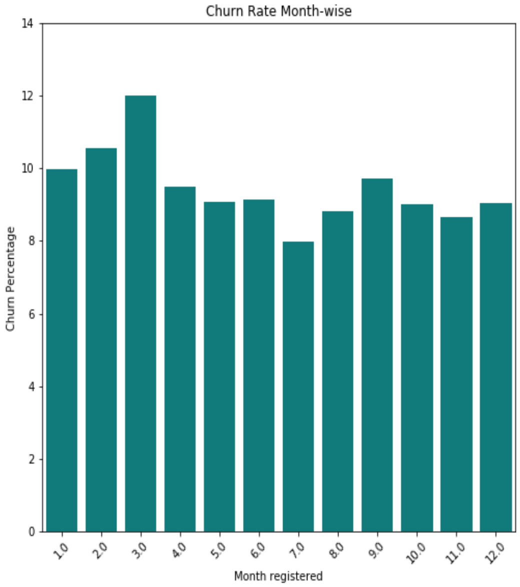
Registration & churning trends yearly & monthly:





- we observed that popularity started rising slowly after 2009 and it started to increase strongly from 2012. Registrations are high during the year end (oct, nov, dec) and year starting months (jan, feb).



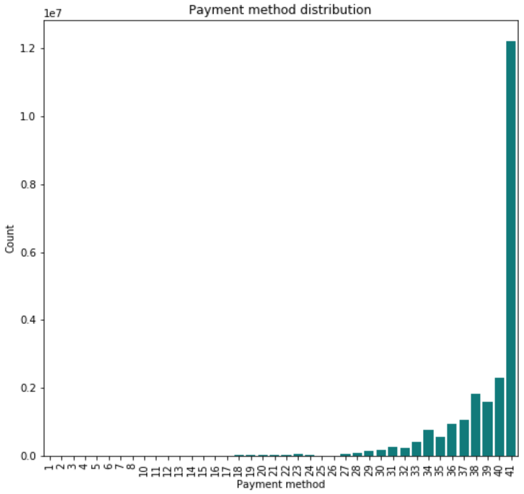


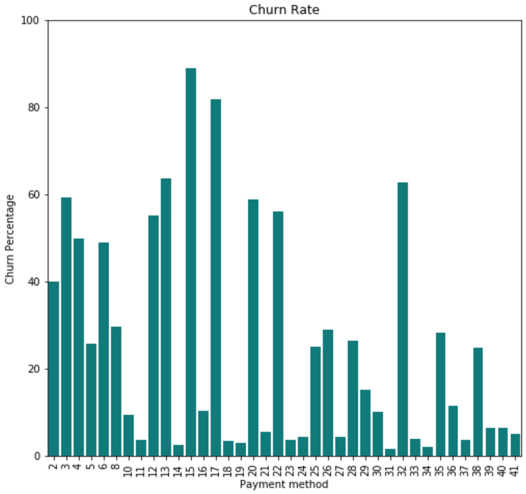
- The churn rate doesn’t seem to follow a trend. It’s been consistent and fluctuating between 8% to 12% regardless of increase in number of registrations every year.

Data Exploration in Transactions & Train

Here we’re merging train & transactions data frames.

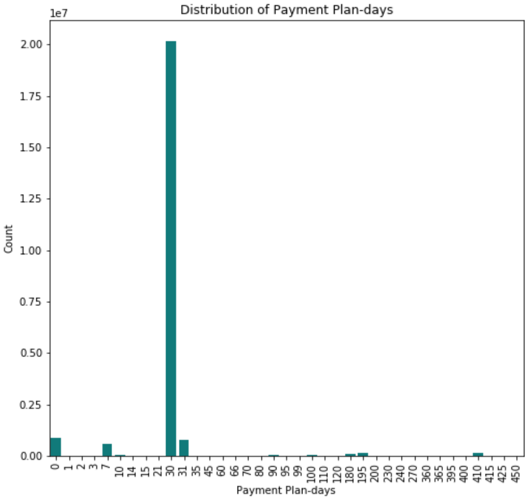
Payment method-id:

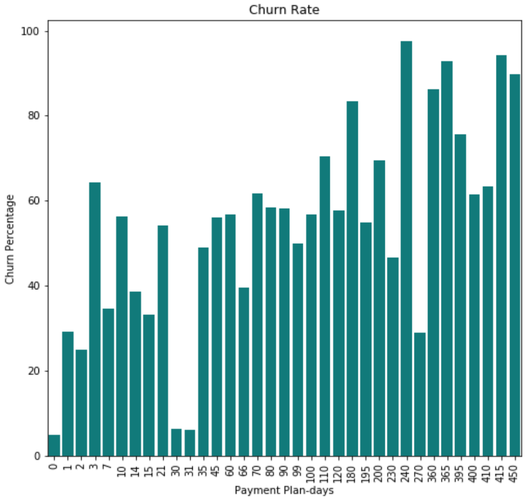




- There are 40 payment methods (method ‘9’ is missing) and the payment - method '41' is by far the most popular one.
- Some payment methods are clearly associated to more loyal users than others. Note that several categories suffer from low-number statistics and the corresponding large churn rate bars. However, the vastly popular payment method “41” is easily in the top 10 of lowest churn.

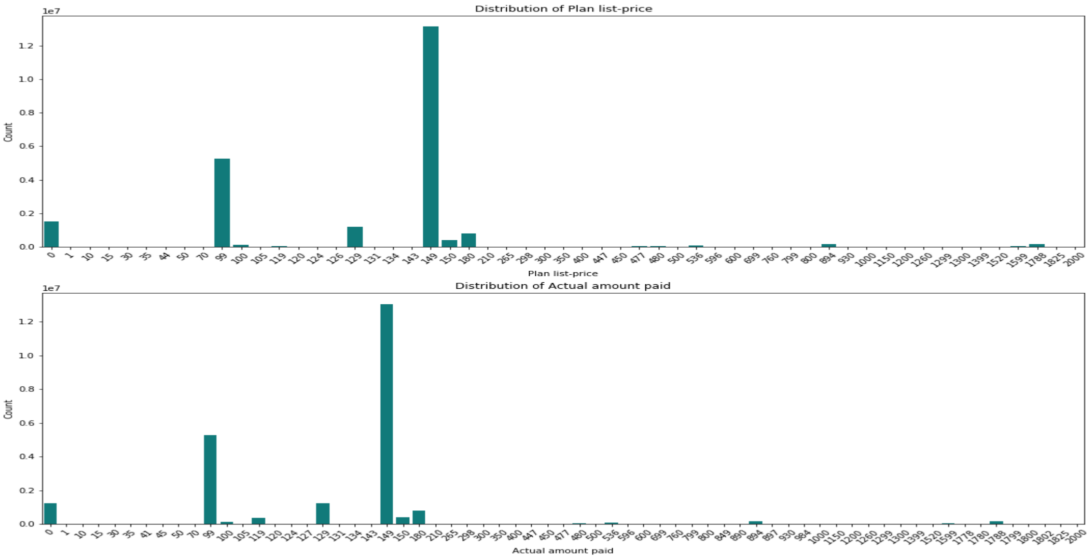
Payment plan days:



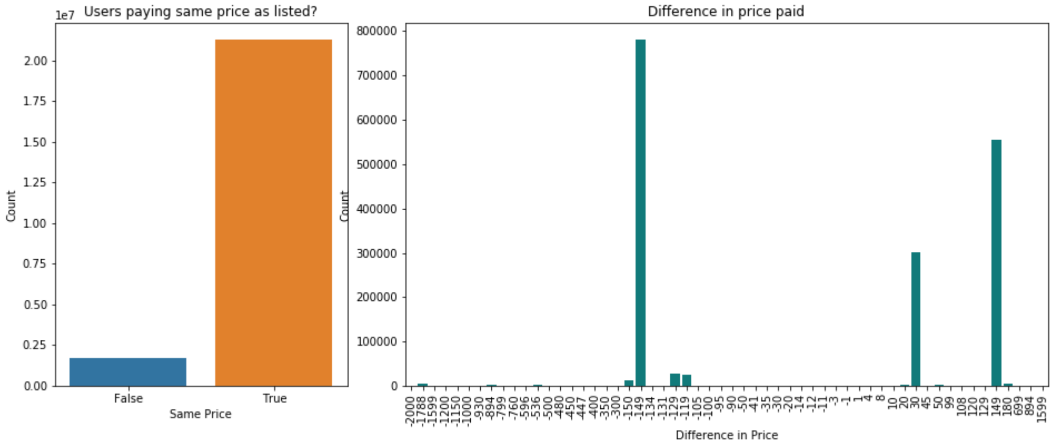


- The payment plan duration categories show strong differences in churn percentage. The lowest churn numbers (around 5%) are associated with the 30-days, 31-days, and the 0-day memberships (surprisingly). The churn percentage for next widely used plans 7-days is 35% & 410-days is 63%. For this feature as well, there are categories with low number statistics and large churn rate bars.

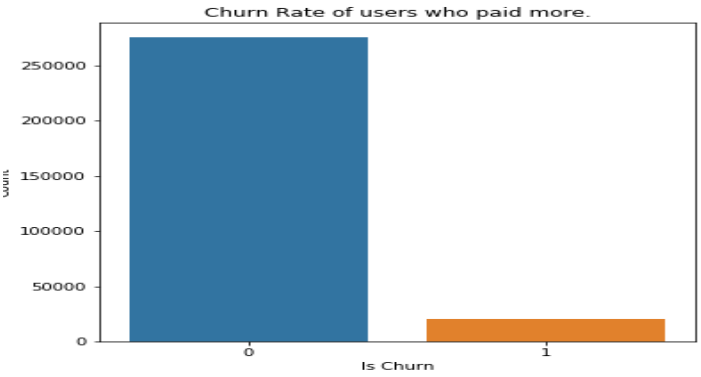
Plan list price & Actual amount paid:



- The overall distributions of planned vs actual payment are very similar, even though differences are slightly visible e.g. for 119 NTD. Since both features have the same discrete payment values we can directly compare their frequency.

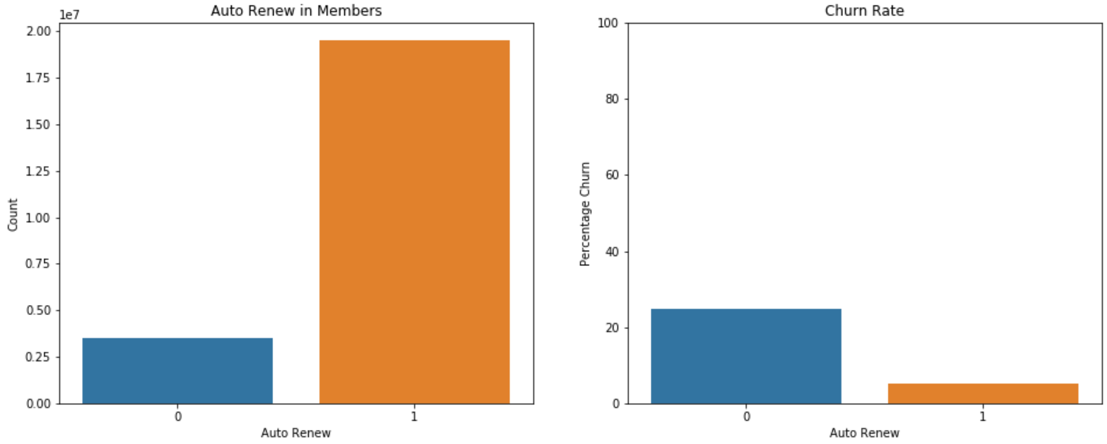


- Interestingly here, in most of the cases the users ended up paying more.



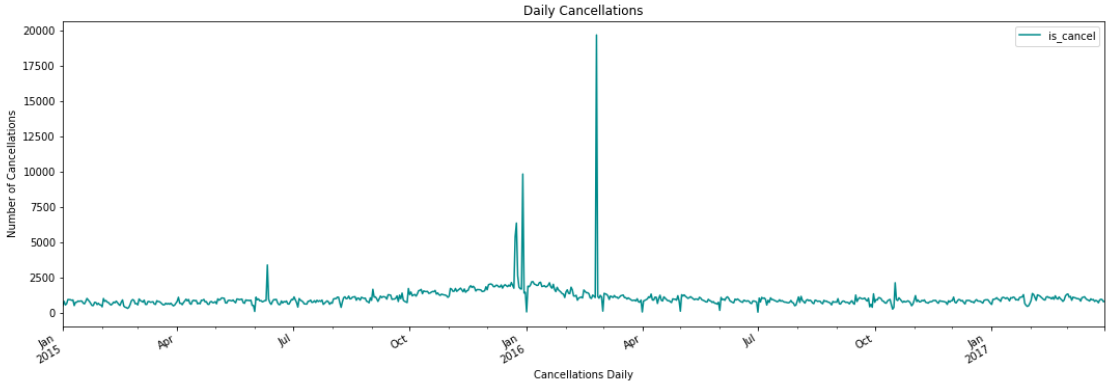
- There isn't any surprising trend in churn rate of users who paid more.

Auto-Renew:

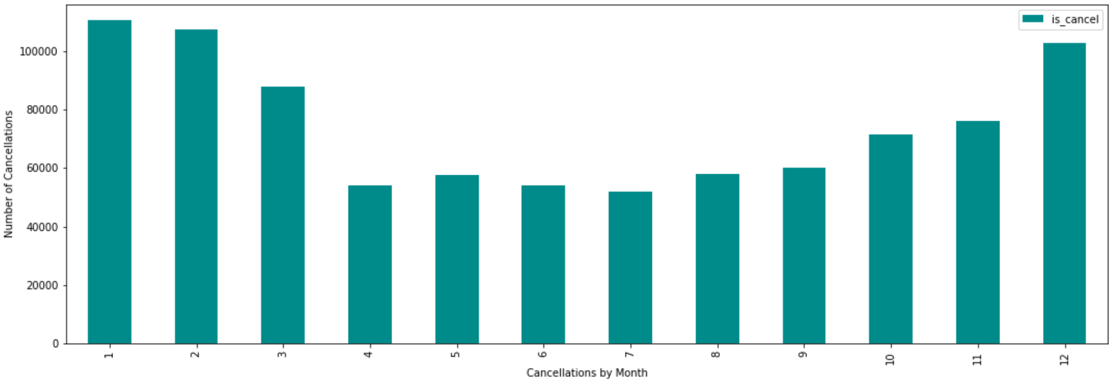


- The vast majority of users have automatic renewal of their subscriptions enabled and users who did not choose to auto renew were clearly more likely to churn.

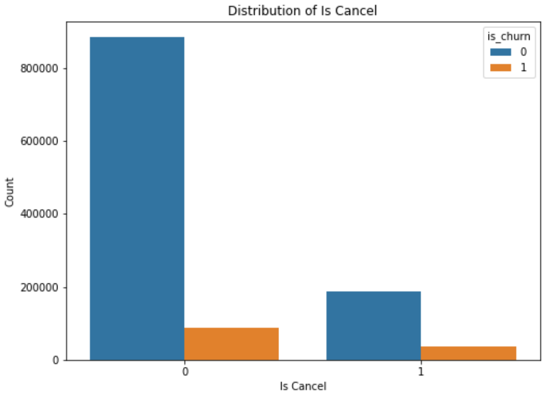
Is-Cancel:

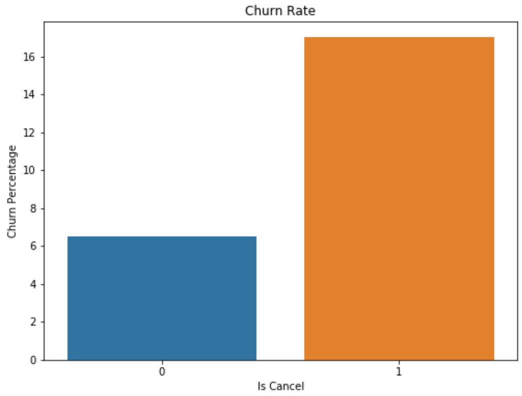


- The cancellation trend seems quite consistent overall with a couple of spikes in jan & mar 2016.



- Cancellations are high in the months of dec, jan, feb & mar. Rest all the months seem very similar.



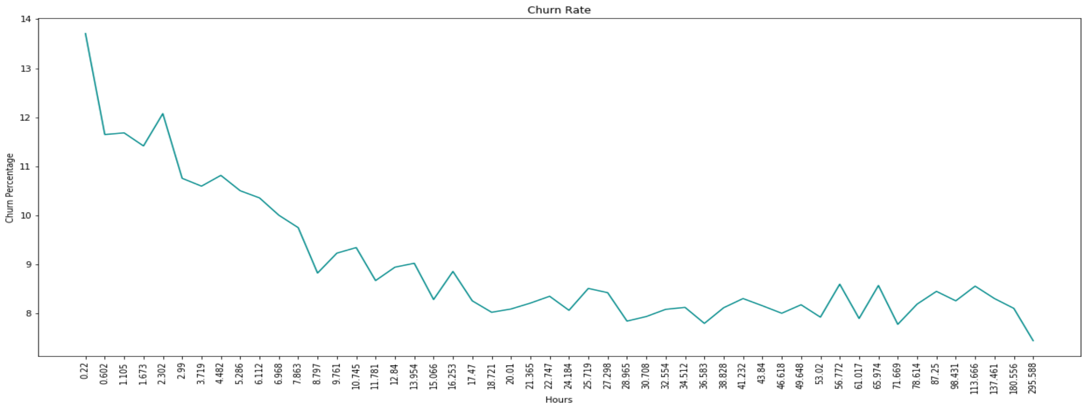


- Not all users who cancelled their subscription are churning. Majority of the cancelled users are re-subscribing within a month.

Data Exploration in User-logs & Train

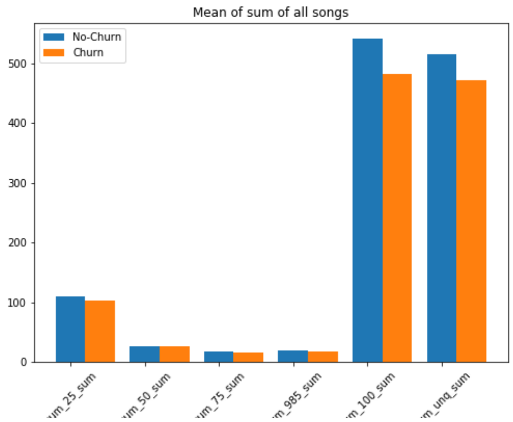
Here we merged the train data frame with user-logs which contains listening behaviors of a user for march.

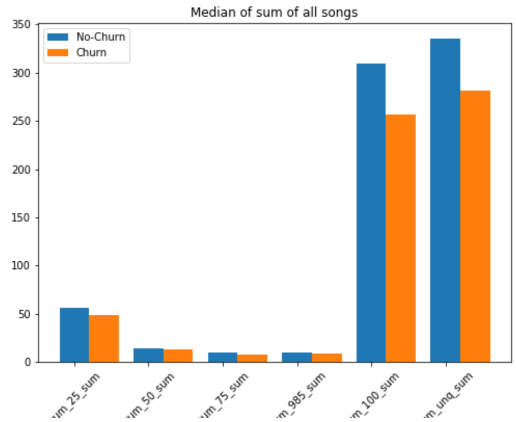
Life span of Churn Users vs No-Churn Users:



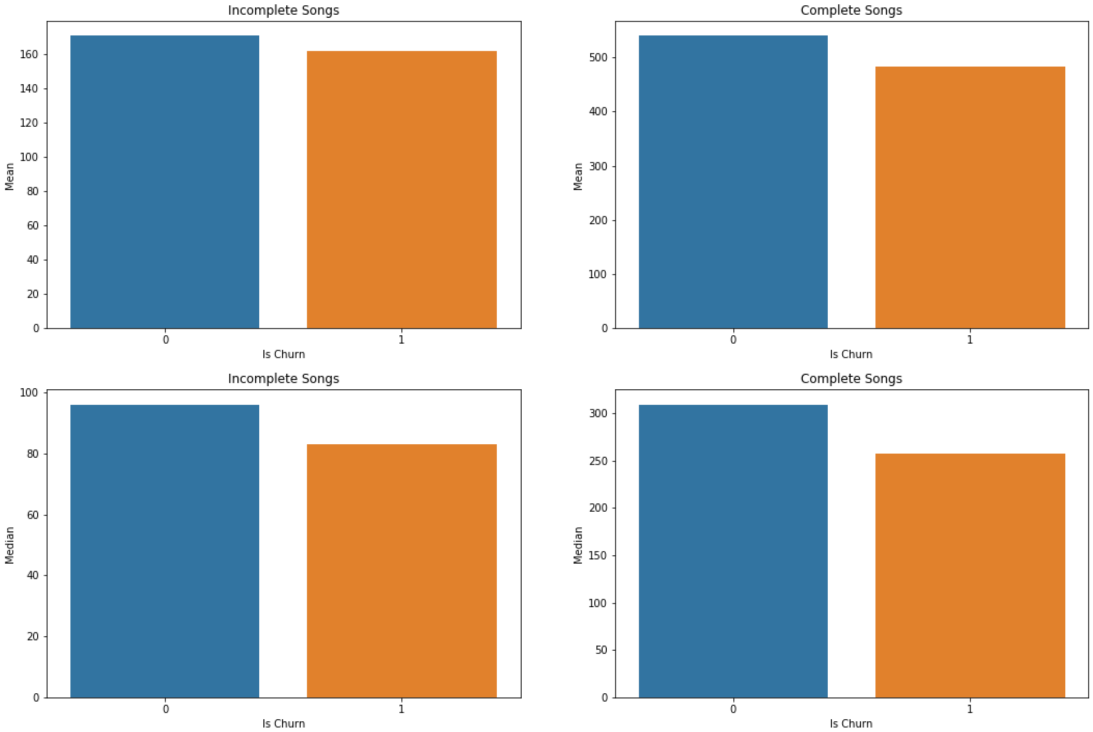
- Here we observed that users with less life span are slightly churning more.

Mean & Median of sum of all songs by Churn Users Vs No-Churn Users:



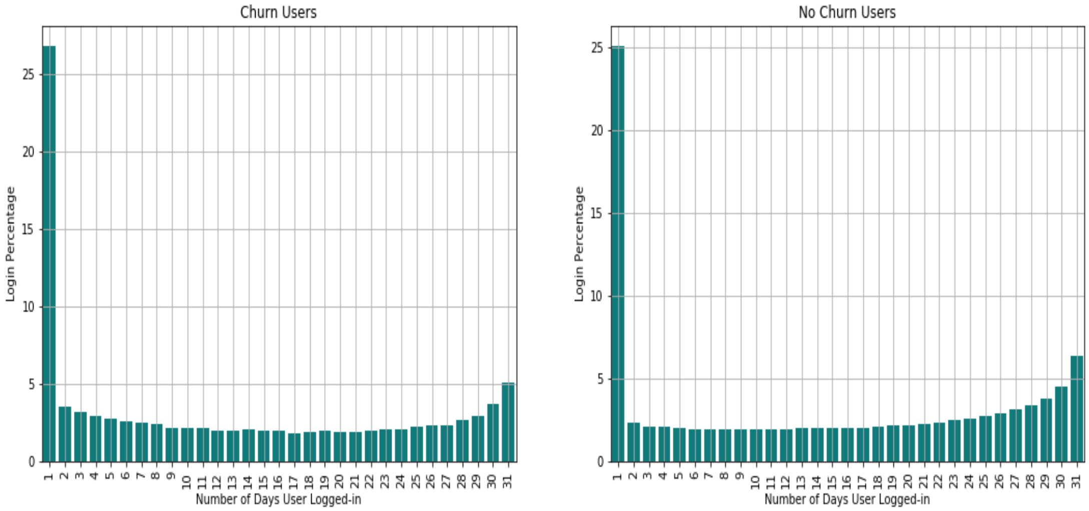


<u>Number of complete songs against incomplete songs</u>:



- Hence, it is clear from the above two plots that Churn users are listening to less number of songs when compared to No-Churn users.

<u>Number of days user has logged in</u>:



- Here, we observe that both the churn users & not churn users log-ins are very similar.

Feature Engineering

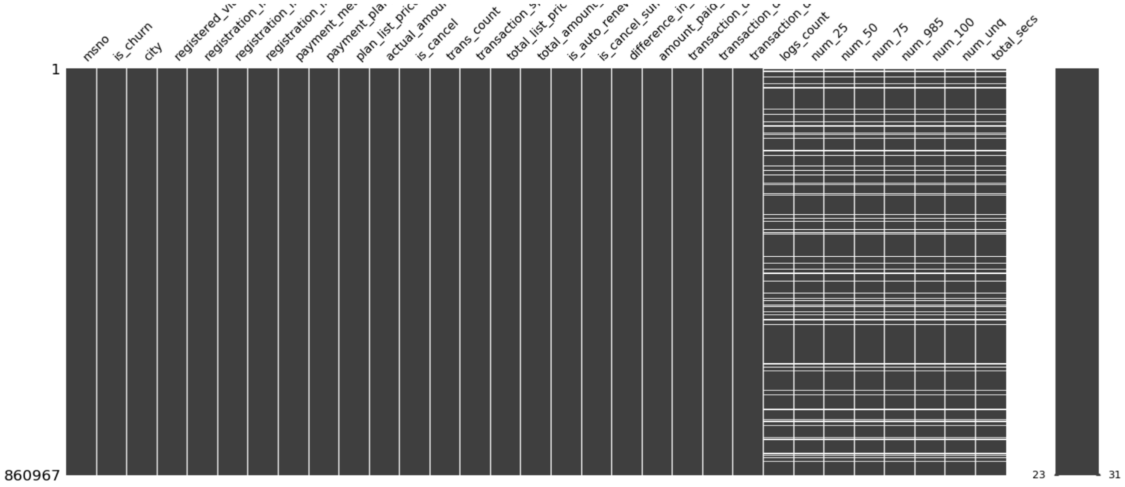
Feature Engineering is the process of creating additional relevant features from the existing raw features in the data, and to increase the predictive power of the learning algorithm. Here are the features that are created.

- **trans_count**: Count of number of transactions for each user.
- **transaction span**: Sum of payment plan days of all the transactions for each user.
- **total_list_price**: Sum of the listed price in all the transactions for each user.
- **total_amount_paid**: Sum of the amount paid in all the transactions by a user.
- **difference_in_price_paid**: Some of the users have paid more or less amount than the list price. So, this feature calculates the difference between the list price and price paid for each user.
- **amount_paid_perday**: This feature is created by dividing the total amount paid by transaction span of each user.
- **logs_count**: Count of number of entries for each user in the user_log dataset.

Also, we have dropped 'bd' and 'gender' features from our final data-frame because of having around 60% null-values.

All the date columns are transformed into 'year', 'month', and 'day' columns respectively.

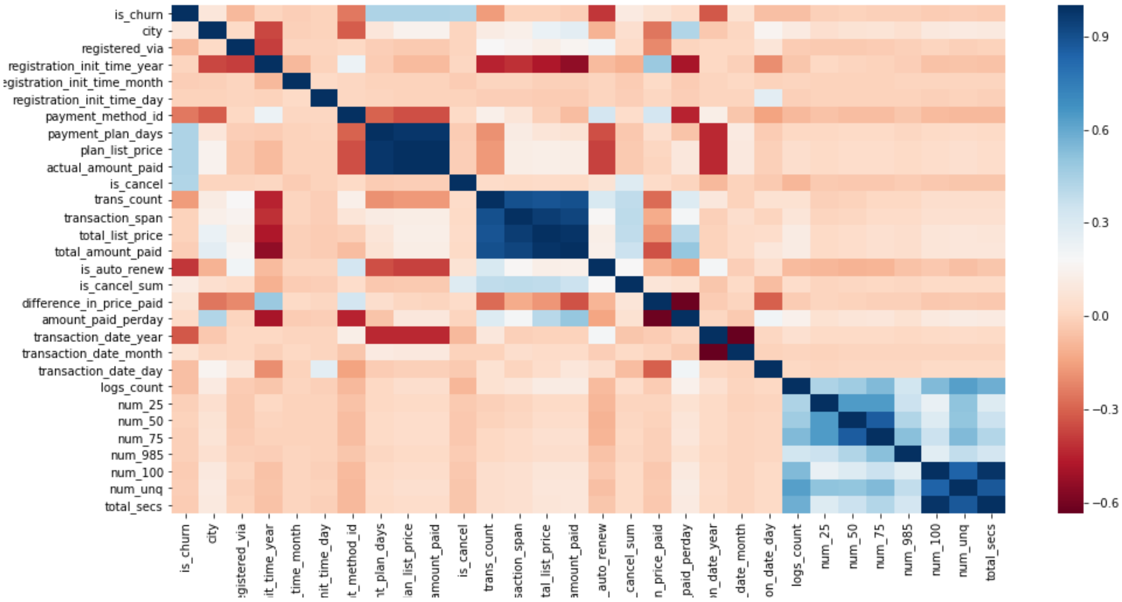
Finally, all the data-frames are joined using left join on 'msno' to the train data-set. Then the final data-frame is visualized for null – values. Here is how it looks.



It is clearly seen in the above image that we have some user entries missing in the user-logs dataset. If it was just in a column or two we could have imputed the missing values with some statistic. Because the entire row has missing values here, we are dropping all the rows with missing rows.

Correlation Heat Map:

The correlation matrix of the final data-frame is visualized below. We observe some strong correlations between some features in transaction and user-log entries.



- There is a strong correlation between *plan_list_price* and *actual_amount_paid* because it is quite obvious that most of the customers paid the listed price.
- Also, we can see that the *payment_plan_days* and both the *plan_list_price, actual_amount_paid* are strongly correlated because the prices are fixed for each and every payment plan.
- Because we have created some new features *(total_list_price, total_ amount_paid)* from the existing ones like *(*plan*_list_price, actual_amount_paid)* which are already strongly correlated we observe high correlation between the new features as well.
- In User-logs dataset we can see that *total_secs* and *num_100* are strongly correlated which totally makes sense that users who played the entire songs will obviously have listened to the songs higher number of seconds.

Data Splitting:

Finally, before building the machine learning models the dataset is split into ‘train’ and ‘test’ sets. Here we have split the train and test sets in 80:20 ratio. We will build our models on the train set and test them on the test set and evaluate our metric which is chosen as ‘log-loss’ in this project.

SMOTE:

Synthetic Minority Oversampling (SMOTE) works by creating synthetic observations based upon the existing minority observations. Since our dataset is highly imbalanced, after splitting the datasets we applied SMOTE on the training set. By oversampling only on the training data, none of the information in the test data is being used to create synthetic observations. So, these results should be generalizable.

At a high level, SMOTE creates synthetic observations of the minority class (Churn-users) by:

1. Finding the k-nearest-neighbors for minority class observations (finding similar observations)
2. Randomly choosing one of the k-nearest-neighbors and using it to create a similar, but randomly tweaked, new observation.

Here are the log-loss values after implementing smote.

Classifier	**Log-loss**
Logistic Regression	0.38806969577590855
Decision Tree Classifier	2.0539204898906522
Random Forest Classifier	0.4124255461464827
Gradient- Boosting Classifier	0.14705783211255458
AdaBoost Classifier	0.6606471482499868
XGBoost Classifier	0.13182430145603982

Because we already have 600K samples in our training set before SMOTE and we will get around 1M samples in our training set after SMOTE we will not be using this method to tune our parameters due to high memory requirement and computationally not efficient.

Log-Loss:

Logarithmic Loss, or simply Log Loss, is a classification loss function often used as an evaluation metric in classification problems. Since success in these projects hinges on effectively minimizing the Log Loss, it makes sense to have some understanding of how this metric is calculated and how it should be interpreted.

Log Loss quantifies the accuracy of a classifier by penalizing false classifications. Minimizing the Log Loss is basically equivalent to maximizing the accuracy of the classifier, but there is a subtle twist which we'll get to in a moment.

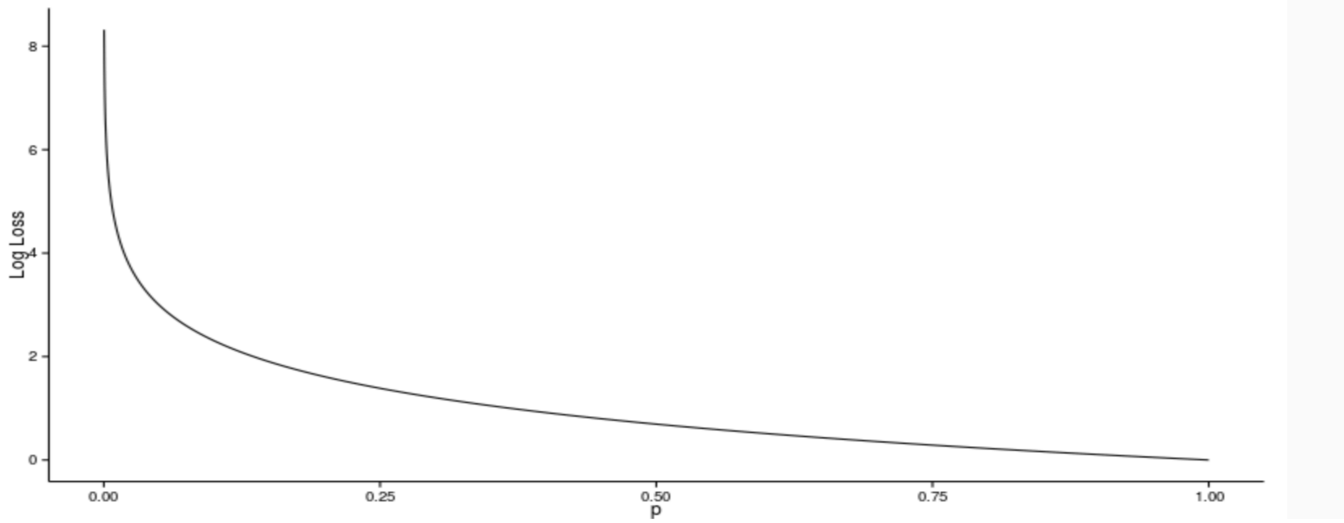
In order to calculate Log Loss, the classifier must assign a probability to each class rather than simply yielding the most likely class. If there are only two classes mathematically Log Loss is defined as

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1-y_i)\log(1-p_i)].$$

where $\boldsymbol{N}$ is the number of samples or instances, $\boldsymbol{y_i}$ is a binary indicator, and p_i is the predicted probability that y_i equals 1.

A perfect classifier would have a Log Loss of precisely zero. Less ideal classifiers have progressively larger values of Log Loss. Note that for each instance only the term for the correct class actually contributes to the sum.

The plot below shows the Log Loss contribution from a single positive instance where the predicted probability ranges from 0 (the completely wrong prediction) to 1 (the correct prediction). It's apparent from the gentle downward slope towards the right that the Log Loss gradually declines as the predicted probability improves. Moving in the opposite direction though, the Log Loss ramps up very rapidly as the predicted probability approaches 0.



Log Loss heavily penalizes classifiers that are confident about an incorrect classification. For example, if for a particular observation, the classifier assigns a very small probability to the correct class then the corresponding contribution to the Log Loss will be very large indeed. Naturally this is going to have a significant impact on the overall Log Loss for the classifier. The bottom line is that it's better to be somewhat wrong than emphatically wrong. Of course, it's always better to be completely right, but that is seldom achievable in practice! There are at least two approaches to dealing with poor classifications:

1. Examine the problematic observations relative to the full data set. Are they simply outliers? In this case, remove them from the data and re-train the classifier.
2. Consider smoothing the predicted probabilities using, for example, Laplace Smoothing. This will result in a less "certain" classifier and might improve the overall Log Loss.

Predictive Modelling

Here we are implementing machine learning algorithms to build a predictive model. Since our problem is a classification problem lets first implement some of the simple classification algorithms & then ensemble methods and evaluate our metric 'log-loss' and see which method results lower 'log-loss' value.

The returned 'log-loss' values by each classifier are as follows:

Classifier	**log-loss value**
Logistic Regression	0.21877964999512514
Decision Tree Classifier	2.0344661505056827
Random Forest Classifier	0.4136045064093898
Gradient-Boosting Classifier	0.11795942846203314
Ada-Boost Classifier	0.653128643077741
XG-Boost Classifier	0.11729961185499269

Because, the Gradient-Boosting Classifier & XG-Boost Classifier are returning lower 'log-loss' values we are using them as baseline models and try to tune their parameters to optimize the loss-function and see if the 'log-loss' value gets any better.

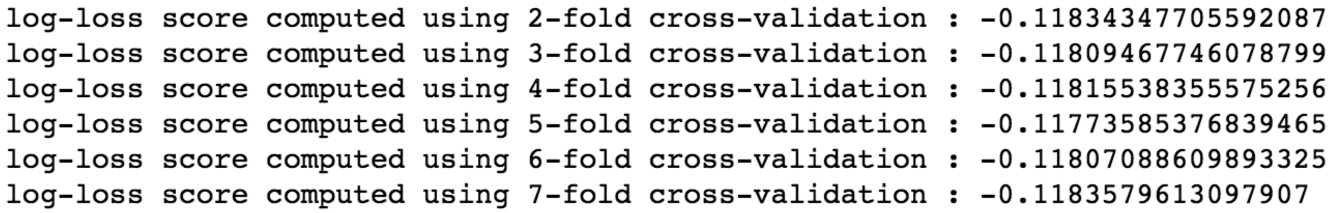
Parameter Tuning in Gradient-Boosting Classifier

There are many parameters that are available which can be tuned. But, we are only tuning some of them. Here are the parameters which we are tuning.

1. **learning_rate:** This determines the impact of each tree on the final outcome. GBM works by starting with an initial estimate which is updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates. Lower values are generally preferred as they make the model robust to the specific characteristics of tree and thus allowing it to generalize well. Lower values would require higher number of trees to model all the relations and will be computationally expensive.
2. **n_estimators:** The number of sequential trees to be modeled.
3. **max_depth:** The maximum depth of a tree.
4. **min_samples_split:** Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting.
5. **min_samples_leaf**: Defines the minimum samples (or observations) required in a terminal node or leaf.
6. **max_features:** The number of features to consider while searching for a best split. These will be randomly selected. As a thumb-rule, square root of the total number of features works great but we should check up to 30-40% of the total number of features.
7. **subsample:** The fraction of observations to be selected for each tree. Selection is done by random sampling.

The observed results by tuning the parameters are as follows:

cross-validation scores:



```
log-loss score computed using 2-fold cross-validation : -0.11834347705592087
log-loss score computed using 3-fold cross-validation : -0.11809467746078799
log-loss score computed using 4-fold cross-validation : -0.11815538355575256
log-loss score computed using 5-fold cross-validation : -0.11773585376839465
log-loss score computed using 6-fold cross-validation : -0.11807088609893325
log-loss score computed using 7-fold cross-validation : -0.1183579613097907
```

CV = 5 returns a slightly better log-loss value. So, we will be considering 5-fold cross validation in Gridsearch for tuning gradient boosting classifier parameters.

Parameter tuning step-1:

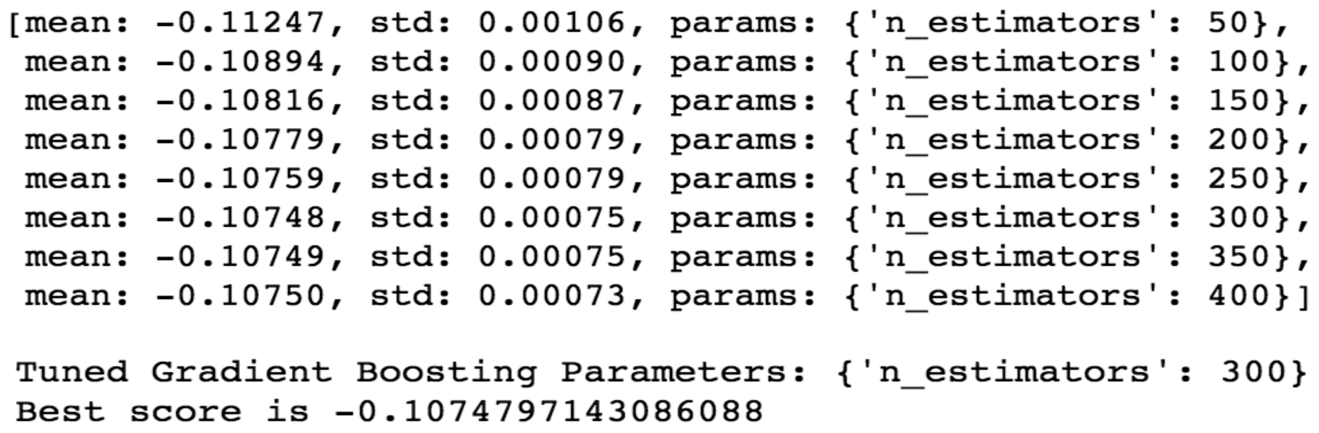
GBM is robust enough to not over fit with increasing trees, but a high number for particular learning rate can lead to overfitting. But as we reduce the learning rate and increase trees, the computation becomes expensive and would take a long time to run on standard personal computers.

Keeping all this in mind, we are taking the following approach:

1. Generally choosing a relatively **high learning rate** works. So, choosing the default value of 0.1.
2. Now to determine the **optimum number of trees for this learning rate** we are considering a range 50-400.

3. **min_samples_split = 500**: This should be ~0.5-1% of total values. Since this is imbalanced class problem, we'll take a small value from the range.
4. **min_samples_leaf = 50**: Can be selected based on intuition. This is just used for preventing overfitting and again a small value because of imbalanced classes.
5. **max_depth = 8**: Should be chosen (5-8) based on the number of observations and predictors. This has 31 columns so let's take 8 here.
6. **max_features = 'sqrt'**: It's a general thumb-rule to start with square root.
7. **subsample = 0.8**: This is a commonly used start value.

Output:



```
[mean: -0.11247, std: 0.00106, params: {'n_estimators': 50},
 mean: -0.10894, std: 0.00090, params: {'n_estimators': 100},
 mean: -0.10816, std: 0.00087, params: {'n_estimators': 150},
 mean: -0.10779, std: 0.00079, params: {'n_estimators': 200},
 mean: -0.10759, std: 0.00079, params: {'n_estimators': 250},
 mean: -0.10748, std: 0.00075, params: {'n_estimators': 300},
 mean: -0.10749, std: 0.00075, params: {'n_estimators': 350},
 mean: -0.10750, std: 0.00073, params: {'n_estimators': 400}]

 Tuned Gradient Boosting Parameters: {'n_estimators': 300}
 Best score is -0.1074797143086088
```

As you can see that here we got 300 as the optimal estimators for 0.1 learning rate. So, we will be using this value for further tuning.

Parameter tuning step-2:

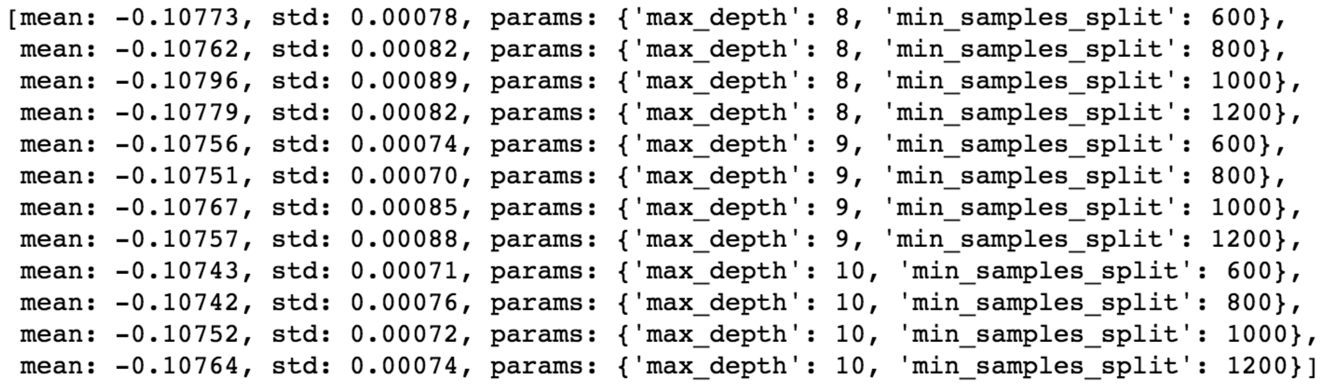
Now let's move onto tuning the tree parameters. I plan to do this in following stages:

1. Tune max_depth and min_samples_split
2. Tune min_samples_leaf
3. Tune max_features

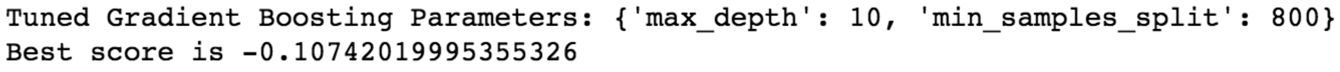
The order of tuning variables should be decided carefully. We should take the variables with a higher impact on outcome first. For instance, max_depth and min_samples_split have a significant impact and we're tuning those first.

To start with, I will test max_depth values of 8 to 11 in steps of 1 and min_samples_split from 600 to 1200 in steps of 200.

Output:



```
[mean: -0.10773, std: 0.00078, params: {'max_depth': 8, 'min_samples_split': 600},
 mean: -0.10762, std: 0.00082, params: {'max_depth': 8, 'min_samples_split': 800},
 mean: -0.10796, std: 0.00089, params: {'max_depth': 8, 'min_samples_split': 1000},
 mean: -0.10779, std: 0.00082, params: {'max_depth': 8, 'min_samples_split': 1200},
 mean: -0.10756, std: 0.00074, params: {'max_depth': 9, 'min_samples_split': 600},
 mean: -0.10751, std: 0.00070, params: {'max_depth': 9, 'min_samples_split': 800},
 mean: -0.10767, std: 0.00085, params: {'max_depth': 9, 'min_samples_split': 1000},
 mean: -0.10757, std: 0.00088, params: {'max_depth': 9, 'min_samples_split': 1200},
 mean: -0.10743, std: 0.00071, params: {'max_depth': 10, 'min_samples_split': 600},
 mean: -0.10742, std: 0.00076, params: {'max_depth': 10, 'min_samples_split': 800},
 mean: -0.10752, std: 0.00072, params: {'max_depth': 10, 'min_samples_split': 1000},
 mean: -0.10764, std: 0.00074, params: {'max_depth': 10, 'min_samples_split': 1200}]
```



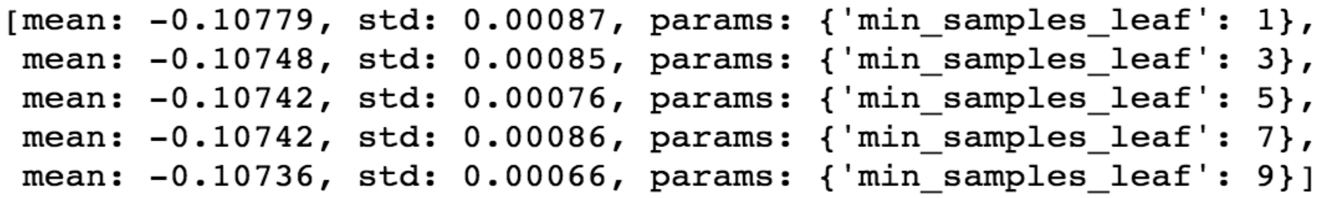
```
Tuned Gradient Boosting Parameters: {'max_depth': 10, 'min_samples_split': 800}
Best score is -0.10742019995355326
```

Here, we have run 12 combinations and the ideal values are 10 for max_depth and 800 for min_samples_split. Note that, 1000 is an extreme value which we tested. We have also checked different ranges for both the parameters but the above displayed results were most ideal. Also, we can see the CV score hasn't changed much. So, considering these values in our further tuning.

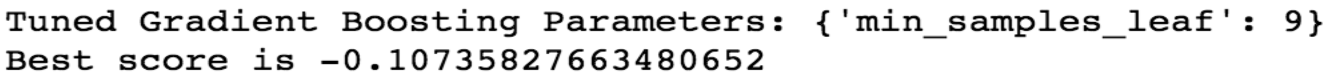
Parameter tuning step-3:

Here, we will take the max_depth of 10 and min_samples_split of 800 as optimum and not try different values for higher min_samples_split. It might not be the best idea always but here if you observe the output closely, max_depth of 10 works better in most of the cases. Also, we can test for 5 values of min_samples_leaf, from 1 to 10 in steps of 2.

Output:



```
[mean: -0.10779, std: 0.00087, params: {'min_samples_leaf': 1},
 mean: -0.10748, std: 0.00085, params: {'min_samples_leaf': 3},
 mean: -0.10742, std: 0.00076, params: {'min_samples_leaf': 5},
 mean: -0.10742, std: 0.00086, params: {'min_samples_leaf': 7},
 mean: -0.10736, std: 0.00066, params: {'min_samples_leaf': 9}]
```



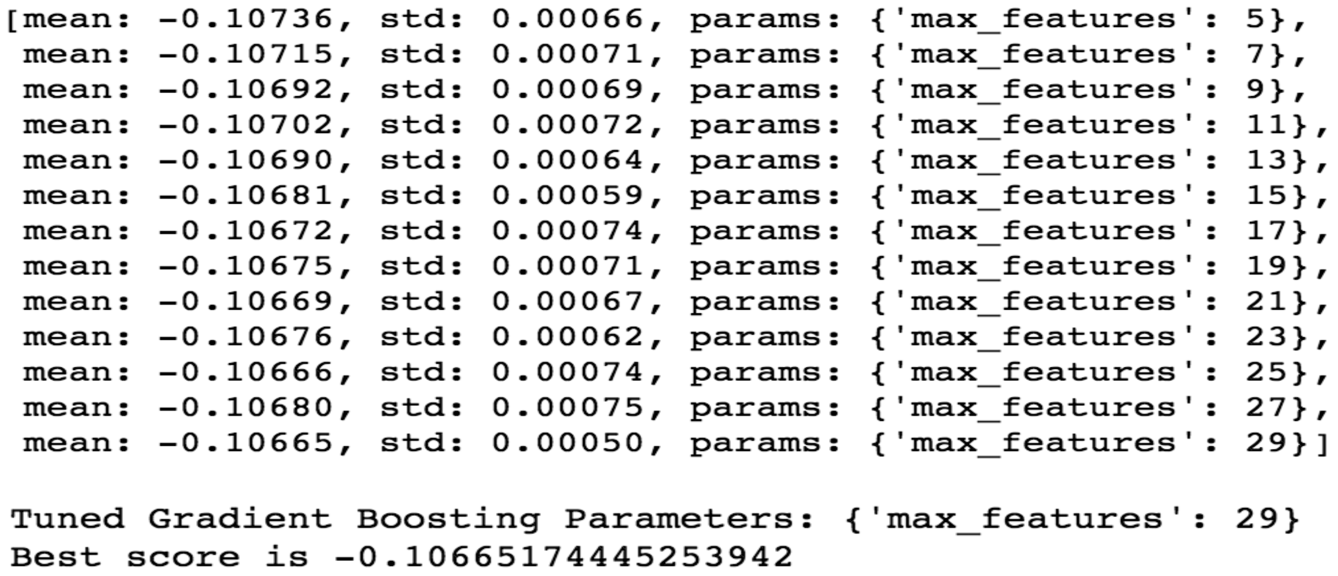
```
Tuned Gradient Boosting Parameters: {'min_samples_leaf': 9}
Best score is -0.10735827663480652
```

Here we get the optimum values as 9 for min_samples_leaf. We have tried different ranges and found out the optimal value of 9. Also, we can see the CV score decreasing to 0.1073 now which is quite an improvement so far. Let's fit the model again on this and have a look at the feature importance.

Parameter tuning step-4:

Now let's tune the last tree-parameters, i.e. max_features by trying 13 values from 5 to 29 in steps of 2.

Output:



```
[mean: -0.10736, std: 0.00066, params: {'max_features': 5},
 mean: -0.10715, std: 0.00071, params: {'max_features': 7},
 mean: -0.10692, std: 0.00069, params: {'max_features': 9},
 mean: -0.10702, std: 0.00072, params: {'max_features': 11},
 mean: -0.10690, std: 0.00064, params: {'max_features': 13},
 mean: -0.10681, std: 0.00059, params: {'max_features': 15},
 mean: -0.10672, std: 0.00074, params: {'max_features': 17},
 mean: -0.10675, std: 0.00071, params: {'max_features': 19},
 mean: -0.10669, std: 0.00067, params: {'max_features': 21},
 mean: -0.10676, std: 0.00062, params: {'max_features': 23},
 mean: -0.10666, std: 0.00074, params: {'max_features': 25},
 mean: -0.10680, std: 0.00075, params: {'max_features': 27},
 mean: -0.10665, std: 0.00050, params: {'max_features': 29}]

Tuned Gradient Boosting Parameters: {'max_features': 29}
Best score is -0.10665174445253942
```

Interestingly our model returned a lower log-loss value for 29 features which are maximum number of features. Though the log-loss value went down as the features increased it looked almost the same for 21, 25, 29 features. Could have tried different numbers for max_features but we are considering 29 for now.

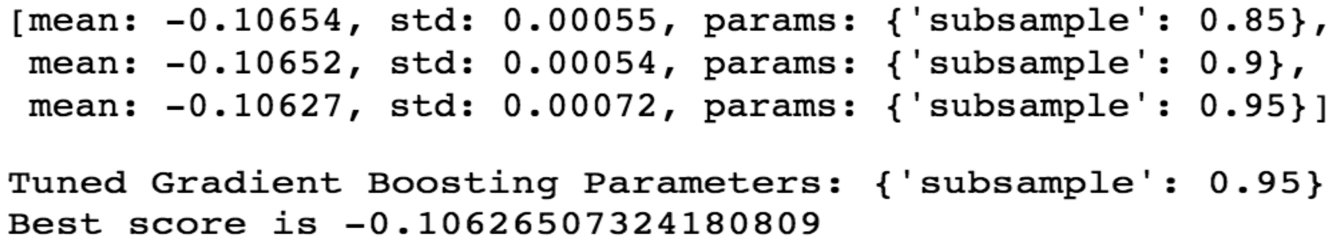
With this we have the final tree-parameters as:

- min_samples_split: 800
- min_samples_leaf: 9
- max_depth: 10
- max_features: 29

Parameter tuning step-5:

The next step would be try different subsample values. Let's take values 0.85, 0.9, 0.95. Initially we chose the values from 0.5 to 0.95.

Output:



```
[mean: -0.10654, std: 0.00055, params: {'subsample': 0.85},
 mean: -0.10652, std: 0.00054, params: {'subsample': 0.9},
 mean: -0.10627, std: 0.00072, params: {'subsample': 0.95}]

Tuned Gradient Boosting Parameters: {'subsample': 0.95}
Best score is -0.1062650732418080 9
```

Here, we found 0.95 as the optimum value. Finally, we have all the parameters needed. Now, we need to lower the learning rate and increase the number of estimators proportionally. Note that these trees might not be the most optimum values but a good benchmark. As trees increase, it will become increasingly computationally expensive to perform CV and find the optimum values.

Final step:

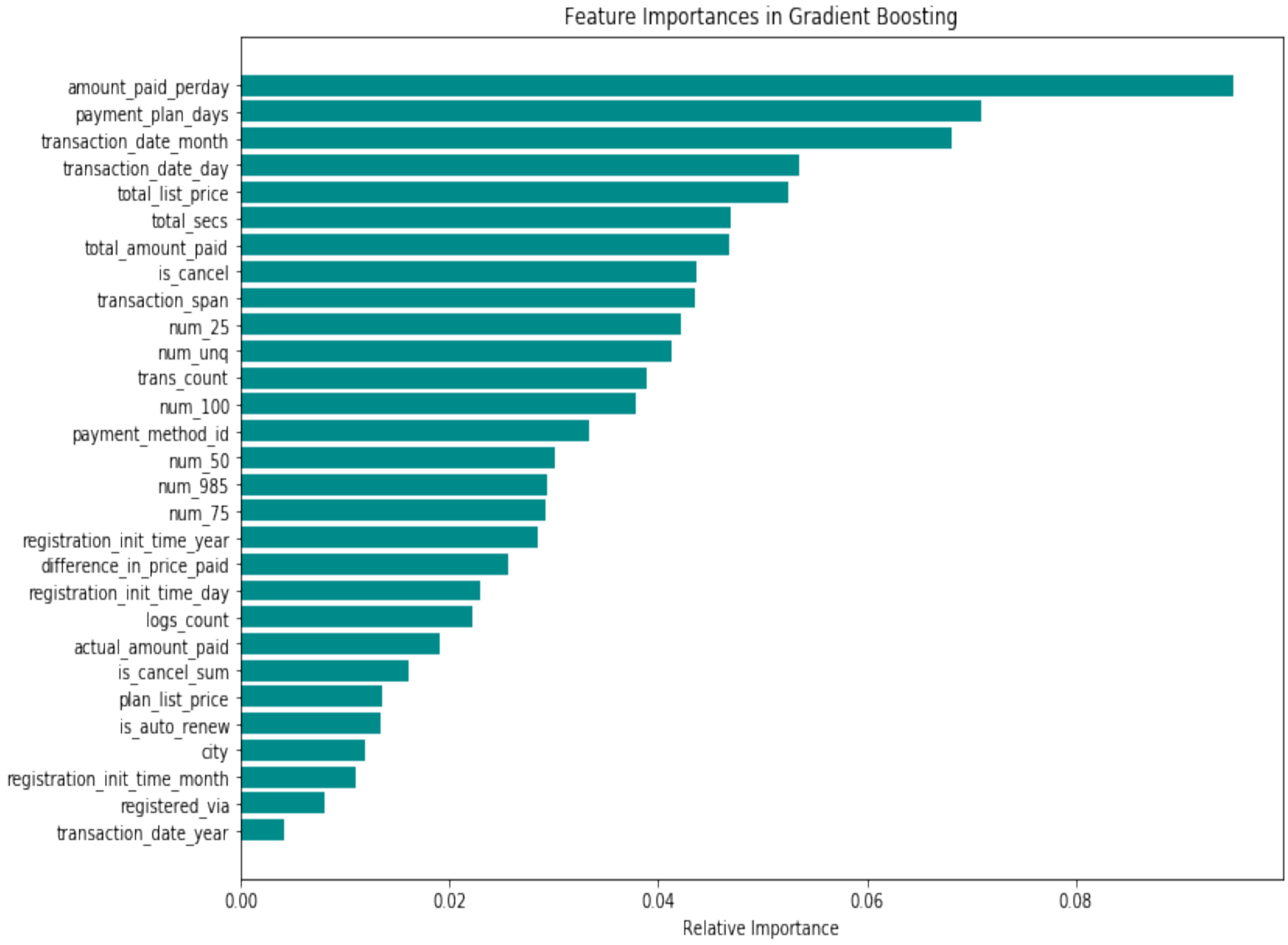
Now let’s reduce to one-twentieth of the original value, i.e. 0.005 for 6000 trees.

Output:

log-loss:	**0.1026813547203256**

Though it gets computationally very expensive finally now we can clearly see that this is a very important step as log-loss scored improved from ~0.1062 to ~0.1026 which is a significant jump.

Gradient-Boosting Feature Importance plot:



Conclusion:

Finally, we can say that the ‘amount paid per day by each user’, ‘payment plan days’, ‘transaction dates’ are top contributors for user churn.

We can even try tuning XG-Boost classifier parameters and also include ‘user-log’ file which we couldn’t use in this analysis due to large file size (32 GB) to improve ‘log-loss’ score.

Business Impact:

Since the music streaming service providers are becoming more competitive day by day one of the major problem these companies are facing is customer retention. Because 10 – 20 percent of customers leave when their subscriptions expire, and it is getting increasingly difficult to acquire new customers. Since the market is now saturated, the huge growth in the market has tapered off. These companies are now engaged in battles to attract each other’s customers while retaining their own.

Attracting new customers is much more expensive than retaining existing ones, so a good deal of marketing budget is allocated to prevent churn. Hence this predictive model helps the business by predicting the customers who are going to churn which helps the business to take certain measures to retain those customers beforehand.