# Music Box Project Report

Ziwei Wang

# Project overview

X music box is a well-known music streaming service and interested in Churn Prediction and Recommendation. To help explore this question, they have provided log data containing billions of song play, search, and download records generated by 600K users.

**Github address:** https://github.com/will-zw-wang/Music_box-Churn_Prediction_and_Recommender_System

# Dataset description

**Play log data:**
'uid', 'device', 'song_id', 'song_type', 'song_name', 'singer', 'play_time', 'song_length', 'paid_flag', 'date'

**Download log data:**
'uid', 'device', 'song_id', 'song_name', 'singer', 'paid_flag', 'date'

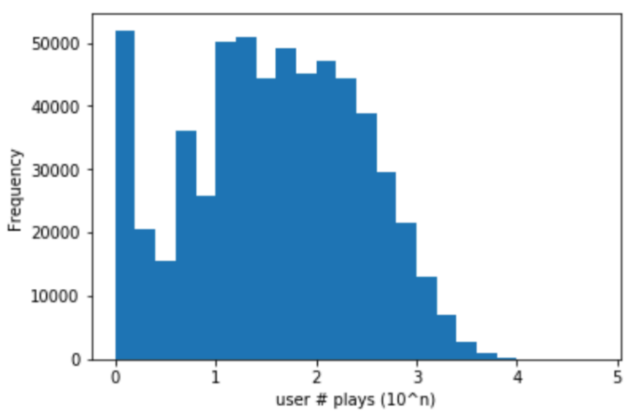
**Search log data:**
'uid', 'device', 'time_stamp', 'search_query', 'date'

# A. Data Processing and Feature Generation

## 1. Load data into Spark DataFrame and perform brief data exploration

### 1.1 Remove bots and outliers:

We counted the number of records per 'uid' to figure out bots and outlier which have unreasonable huge amount of records. We plotted the log(count) with histogram as below:




We noticed there were a few 'uid' have huge amount of records up to 10^5, we removed these bots and outliers extremely large 'count' larger than 99.9 percentile of all 'count' value.

**1.2 Apply downsampling on uid level:**

As we had nearly 1.5 billion records generated by 600K users after moving the bots and outliers, we performed downsampling on 'uid' level to reduce computing pressure.

Down_sample_ratio was 0.1 and finally we had records generated by 60K users.
Then, we had Spark DataFrame as below:

```
+---------+-----+------+-------+----------+---------+-----------+
|      uid|event|device|song_id|      date|play_time|song_length|
+---------+-----+------+-------+----------+---------+-----------+
|168540455|    P|    ar| 298250|2017-03-30|      189|        190|
|168535490|    P|    ar|6616004|2017-03-30|      283|        283|
|168530895|    P|    ar|      0|2017-03-30|      264|        265|
|168551548|    P|    ar|1474915|2017-03-30|        5|        243|
|168551509|    P|    ar|6329735|2017-03-30|      289|        289|
+---------+-----+------+-------+----------+---------+-----------+
only showing top 5 rows
```

**1.3 Briefly data exploration**
total number of entries: 12445200

total number of users: 59754

We count the event type, 'D' for 'download', 'S' for 'search' and 'P' for 'play', we noticed 'play' records accounted for a large proportion.

```
+-----+--------+
|event|   count|
+-----+--------+
|    D|  646004|
|    S|  779581|
|    P|11019615|
+-----+--------+
```

The most popular songs as below

```
+--------+------+
| song_id| count|
+--------+------+
|       0|914738|
|    null|785038|
| 9950164| 87436|
|15249349| 55854|
| 5237384| 41825|
+--------+------+
```

The most active user_id as below

```
+---------+-----+
|      uid|count|
+---------+-----+
|168954949| 7131|
|167925318| 6829|
|167979374| 6343|
|168416042| 5564|
|168442087| 5447|
+---------+-----+
only showing top 5 rows
```

## 2. Label definition

We defined **churn** and **time windows** as below:
**1 for churn:** user had P/D/S entries in 'Feature window' while had no P/D/S entries in 'Label window'
**0 for not churn:** user had P/D/S entries in both two windows

**Label window:** 2017-04-29 ~ 2017-05-12 days: 14
**Feature window:** 2017-03-30 ~ 2017-04-28 days: 30

**Note:** here we ignored user who had P/D/S entries in 'Label window' while had no P/D/S entries in 'Feature window'

We counted the label and noticed that we have 36,272 churn label and 22,160 active label, two classes were comparable.

| label | count |
|---|---|
| 1 | 36272 |
| 0 | 22160 |

## 3. Data cleaning

As we mentioned above, that 'play' records accounted for a large proportion of the total records, we summarized the **'play' records in feature window** as below**:**

| summary | uid | event | device | song_id | play_time | song_length |
|---|---|---|---|---|---|---|
| count | 8395152 | 8395152 | 8395152 | 8391657 | 8391865 | 8393403 |
| mean | 1.6628846441030693E8 | null | null | 1.656281819333759... | 28226.20623478178 | -1231.9704732739265 |
| stddev | 1.500893981882352E7 | null | null | 4.629741133728094... | 7.927618484536014E7 | 1820892.4337731672 |
| min | 100077577 | P | ar | -1 | -0.011976957 | -1 |
| max | 99850419 | P | ip | 9999722 | nan | 999 |

We noticed some issue with column **'song_length'** and **'play_time'** as below:
**1.** We need to check whether there are songs with zero or negative 'song_length', if so, we need to manipulate these values.
**2.** We noticed that the mean value of 'song_length' was negative, there may be some outliers, we need to exclude them.
**3.** We noticed that there are 'nan' and negative value in 'play_time', we need to manipulate these values.
**4.** We noticed that the mean value of 'play_time' is incredibly large, there may be some outliers, we need to exclude them.

### 3.1 Data cleaning for "song_length"

**Problem:**
**1.** We need to check whether there are songs with zero or negative 'song_length', if so, we need to manipulate these values.
**2.** We noticed that the mean value of 'song_length' was negative, there may be some outliers, we need to exclude them.

**Removed songs with missing 'song_lenght':**
**count 'nan' song_length**: There are 1749 songs with NULL song_lenght
We noticed all the songs records with missing 'song_lenght' also missing 'song_id' and missing 'play_time', and these odd songs were played by only 17 'uid'.
We checked the other records generated by these 17 users and did not find any abnormal, so we did not need to remove all the records generated by them.
Thus, we regarded these song records as outliers and remove these records directly.
**Checked pattern of 'song_length == 0':**

To check if there are songs with “song_length == 0” while “play_time ≠ 0”, if so, we need to deal with “song_length == 0”. There are 287252 songs with “song_length == 0” while “play_time ≠ 0”.
We need to deal with the the songs with “song_length == 0”, while before that, I need to check whether the churn rate of songs with “song_length == 0” and that of songs with “song_length !=0” are significant different.

| song_length_unnormal_churn_rate | song_length_normal_churn_rate |
|---|---|
| 0.32 | 0.26 |

We ran a **t-test** and proved that ‘song_length == 0’ appeared randomly without pattern, and as we would remove outliers later, so we could **replace them with mean value later**.

Replace **negative** and **zero** song_length with 'song_length_mean', and **removed outliers** with extremely large ‘song_length’ **larger than 99.9 percentile of all ‘song_length’ value**:
99th of song_length as top_song_length_threshold to remove outliers is 1333
mean value of larger or equal to zero "song_length" records is 223.0

**Here we solved the issue 1 and 2 mentioned above.**

### 3.2 Data cleaning for "play_time"

**Problem:**

**3.** We noticed that there are 'nan' and negative value in ‘play_time’, we need to manipulate these values.
**4.** We noticed that the mean value of 'play_time' is incredibly large, there may be some outliers, we need to exclude them.

**Checked pattern of NULL and Negative play_time:**
There are 1183 songs with NULL play_time. Before I deal with the songs with NULL play_time, I need to check whether the churn rate of songs with NULL play_time and that of songs with “play_time >=0” are significant different?

| play_time_unnormal_churn_rate | play_time_normal_churn_rate |
|---|---|
| 0.28 | 0.27 |

We ran **a t-test** and proved that missing ‘play_time’ appeared randomly without pattern, and as we would remove outliers later, so we could replace them with **mean value** later.
As there are only 6 **songs with negative play_time**, we will manipulate them together with NULL play_time records, replace them with **mean value** of “play_time >=0” records

**Removed outliers** with extremely large ‘play_time’ larger than 99.9 percentile of all ‘play_time’ value.
As we found only several negative ‘play_time’ records, we **replaced negative and zero** ‘play_time’ with mean value.
99th of play_time as top_play_time_threshold to remove outliers 11830.
mean value of larger or equal to zero "play_time" records is 144.0.

**Here we solved the issue 3 and 4 mentioned above.**

We **finished data cleaning** and summarized the **‘play’ records in feature window** as below:

| summary | uid | event | device | song_id | play_time | song_length | label |
|---|---|---|---|---|---|---|---|
| count | 8182785 | 8182785 | 8182785 | 8181042 | 8182785 | 8182785 | 8182785 |
| mean | 1.663321958387472E8 | null | null | 1.682889676009665... | 144.4299803795407 | 240.9282491230064 | 0.26507148849688705 |
| stddev | 1.4871156627324002E7 | null | null | 4.679619339481816... | 272.75805647582456 | 95.71229897828493 | 0.4413712930063083 |
| min | 100077577 | P | ar | -1 | 0.0 | 1.0 | 0 |
| max | 99850419 | P | ip | 9999722 | 11830.0 | 1333.0 | 1 |

## 4. Feature generation

After data cleaning, we started to generate features, here we generated five kind of features.

### 4.1 Generate Features A: Play time percentage related features

#### 4.1.1 Generate Features A1: Proportion features of different level of play time percentage

Generate **play time percentage proportion feature** as:
'time_percentage_0_to_20',
'time_percentage_20_to_40',
'time_percentage_40_to_60',
'time_percentage_60_to_80',
and 'time_percentage_larger_than_80'
**Note: for each 'uid', the sum of five features above is 1.**

**For example:** 'time_percentage_0_to_20' means 0<=percentage<20, which is ratio of (number of played songs with play time percentage less than 20%) to (total number of played songs) per 'uid'

Features as below:

| uid | time_percentage_0_to_20 | time_percentage_20_to_40 | time_percentage_40_to_60 | time_percentage_60_to_80 | time_percentage_larger_than_80 |
|---|---|---|---|---|---|
| 104777734 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 11596711 | 0.47 | 0.11 | 0.02 | 0.06 | 0.34 |
| 118301183 | 0.7 | 0.1 | 0.0 | 0.0 | 0.2 |
| 151294213 | 0.81 | 0.0 | 0.0 | 0.13 | 0.06 |
| 166601616 | 0.29 | 0.12 | 0.08 | 0.09 | 0.42 |

only showing top 5 rows

#### 4.1.2 Generate Features A2: Acceleration features of different level of play time percentage

Ratio of count of songs played with >=80 percentage of nearest 7 days to that of nearest 30 days.

Feature as below:

| uid | time_percentage_larger_than_80_7d_over_30d |
|---|---|
| 11596711 | 0.45 |
| 118301183 | 0.0 |
| 151294213 | 0.0 |
| 166601616 | 0.45 |
| 167570658 | 0.1 |

only showing top 5 rows

### 4.2 Generate Feature B: Play_time related feature

#### 4.2.1 Generate Features B1: Total play_time

Total play time per 'uid'.

Feature as below:

| uid | total_play_time |
|---|---|
| 104777734 | 37 |
| 11596711 | 11476 |
| 118301183 | 611 |
| 151294213 | 737 |
| 166601616 | 7238 |

only showing top 5 rows

#### 4.2.2 Generate Features B2: Acceleration features of total play_time

**Ratio of total play time of nearest 7 days to that of nearest 30 days.**
if the total play time acceleration ratio is less than 25%, means user had played less time in the last 7 days than average level of the last 30 days.

Feature as below:

| uid | total_play_time_7d_over_30d |
|---|---|
| 104777734 | 0.0 |
| 11596711 | 0.47 |
| 118301183 | 0.0 |
| 151294213 | 0.0 |
| 166601616 | 0.57 |

#### 4.2.3 Generate Features B3: Average play_time of played songs

Average play time of songs per 'uid'.

Feature as below:

| uid | average_play_time |
|---|---|
| 104777734 | 12.33 |
| 11596711 | 88.96 |
| 118301183 | 61.1 |
| 151294213 | 46.06 |
| 166601616 | 92.79 |

only showing top 5 rows

### 4.3 Generate Features C: Event related features

#### 4.3.1 Generate Event features C1: events frequency in given windows

Count of ‘P’, ‘D’ and ‘S’ in given windows per ‘uid’.

Features as below:

```
+---------+-------------+-------------+-------------+--------------+--------------+
|      uid|freq_P_last_1|freq_P_last_3|freq_P_last_7|freq_P_last_14|freq_P_last_30|
+---------+-------------+-------------+-------------+--------------+--------------+
| 81114900|            0|            0|            0|             0|            14|
|168555344|            0|            0|           35|           111|           252|
|168572740|            0|            0|            0|             0|             5|
|168580671|           24|           97|          126|           276|          1524|
|168610161|            0|            0|            0|             2|            49|
+---------+-------------+-------------+-------------+--------------+--------------+
only showing top 5 rows
```

#### 4.3.2 Generate Event features C2:  Acceleration features of events

Ratio of event frequency of nearest 7 days to that of nearest 30 days. (for ‘P’, ‘D’ and ‘S’)
if the total play time acceleration ratio is less than 25%, means user had had operated less frequently in the last 7 days than average level of the last 30 days.

Feature as below:

```
+---------+---------------+
|      uid|P_7d_over_P_30d|
+---------+---------------+
| 81114900|            0.0|
|168555344|           0.14|
|168572740|            0.0|
|168580671|           0.08|
|168610161|            0.0|
+---------+---------------+
```

### 4.4 Generate Features D: Recency related features

#### 4.4.1 Generate Recency features D1: Last Event Time from feature_window_end_date

Measure the time gap between ‘the last active day’ and ‘feature_window_end_date’. (for ‘P’, ‘D’ and ‘S’)

Feature as below:

```
+---------+---------------------------+
|      uid|last_P_time_from_2017-04-28|
+---------+---------------------------+
| 81114900|                         29|
|168555344|                          4|
|168572740|                         29|
|168580671|                          0|
|168610161|                          9|
+---------+---------------------------+
```

#### 4.5 Generate Features E: Profile related features

#### 4.5.1 Generate Profile features E1: device_feature

Device type per ‘uid’.

We found 20 users use multiple devices, we assigned the device label as device with more entries by correspond user, if user has the same entries number of 'ar' and 'ip', we assign its device label as 'ip'.

Feature as below: Here we have device_feature with value '0' for 'ip' and '1' for 'ar'.

```
+---------+------+
|      uid|device|
+---------+------+
| 81114900|     1|
|168555344|     1|
|168572740|     1|
|168580671|     1|
|168610161|     1|
+---------+------+
only showing top 5 rows
```

## 5. Form training data for prediction models

**All faeutres as below:**
A1: df_percentage_proportion_feature
A2: df_percentage_acceleration_feature
B1: total_play_time_feature
B2: df_total_play_time_acceleration_feature
B3: average_play_time_feature
C1: df_event_frequrency_feature_list
C2: df_event_accerlaration_feature_list
D1: df_recency_feature_list
E1: df_device_feature

# B. Churn Prediction Models and Insights

## 0. Define Features and Target, Split train-test data, and define model realted functions

**X.shape:** (58432, 32)
**y.value_counts(): 1 - 36272** & **0 - 22160**
Train-test split the data: 20% & 80%

**Features**
['time_percentage_0_to_20', 'time_percentage_20_to_40', 'time_percentage_40_to_60', 'time_percentage_60_to_80', 'time_percentage_larger_than_80', 'time_percentage_larger_than_80_7d_over_30d', 'total_play_time', 'total_play_time_7d_over_30d', 'average_play_time', 'freq_P_last_1', 'freq_P_last_3', 'freq_P_last_7', 'freq_P_last_14', 'freq_P_last_30', 'freq_D_last_1', 'freq_D_last_3', 'freq_D_last_7', 'freq_D_last_14', 'freq_D_last_30', 'freq_S_last_1', 'freq_S_last_3', 'freq_S_last_7', 'freq_S_last_14', 'freq_S_last_30', 'P_7d_over_P_30d', 'D_7d_over_D_30d', 'S_7d_over_S_30d', 'last_P_time_from_2017-04-28', 'last_D_time_from_2017-04-28', 'last_S_time_from_2017-04-28', 'device_type_0', 'device_type_1']

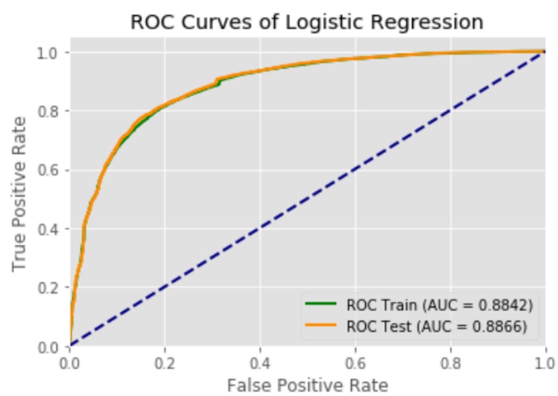
## 1. Models comparison and reasoning

### 1.1 Logistic Regression

As we are taking a classification task, we first try Logistic Regression.
**clf = LogisticRegression(C=1.0, penalty='l2')**

The performance of our model as below:

| metrics | train | test |
|---|---|---|
| AUC | 0.884195 | 0.886565 |
| Accuracy | 0.812001 | 0.814666 |
| Precision | 0.828210 | 0.827604 |
| Recall | 0.880340 | 0.883023 |
| f1-score | 0.853480 | 0.854416 |



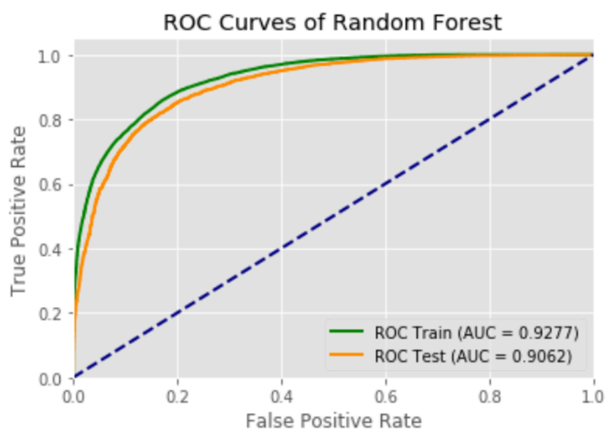

**AUC** of test data is **0.8866** with **Logistic Regression**, we will try to improve model performance with **Random Forest.**

### 1.2 Random Forest

```
# Choose some parameter combinations to try
parameters = {'n_estimators': 50,
              'max_features': 'auto',
              'criterion': 'gini',
              'max_depth': 20,
              'min_samples_split': 2,
              'min_samples_leaf': 20,
              'random_state': 0,
              'n_jobs': -1
              }
```

The performance of our model as below:

| metrics | train | test |
|---|---|---|
| AUC | 0.927675 | 0.906158 |
| Accuracy | 0.852348 | 0.832891 |
| Precision | 0.862525 | 0.845839 |
| Recall | 0.907202 | 0.891081 |
| f1-score | 0.884299 | 0.867871 |



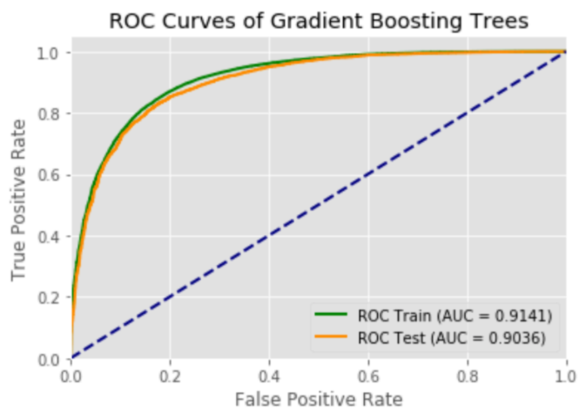

**AUC** of test data is **0.9061** with **Random Forest**, better than that of **Logistic Regression with 0.8866**, because there are feature interaction and non-linearity relationship between features and target in our data set, **trees** algorithms can deal with these problems while **logistic regression** cannot.

Then we tried to further improve model performance with **Gradient Boosting Trees**, because in general, **Gradient Boosting Trees** can perform better than **Random Forest**, because it additionally tries to find optimal linear combination of trees (assume final model is the weighted sum of predictions of individual trees) in relation to given train data. This extra tuning may lead to more predictive power.

### 1.3 Gradient Boosting Trees

```
parameters = {
    'n_estimators': 200,
    'max_depth': 3,
    'learning_rate': 0.2,
    'subsample': 0.8,
    'max_features':0.7,
    'random_state': 42
}
```

```
                train      test
metrics
AUC          0.914053  0.903622
Accuracy     0.846572  0.830838
Precision    0.858485  0.843442
Recall       0.902009  0.890664
f1-score     0.879709  0.866410
```




**AUC** of test data is **0.9036** with **Gradient Boosting Trees**, is close to that of **Random Forest** with **0.9061**, means our Random forest has already performed greatly in this dataset and hard for Gradient Boosting Trees to perform better.

**Thus, we choose Random Forest here.**

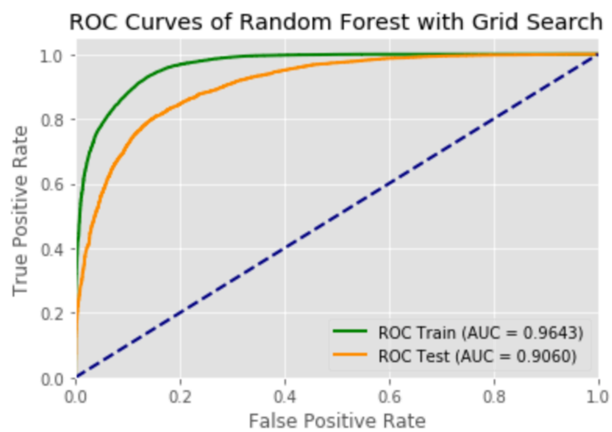
**Next, we will try HyperParameter Tuning with Grid Search for Random Forest, to figure out whether we can do better.**

## 2. HyperParameter Tuning with Grid Search

### 2.1 Random Forest HyperParameter Tuning with Grid Search

```
# Choose some parameter combinations to try
param_grid = {'n_estimators': [100,200],
              'max_features': ['auto'],
              'criterion': ['gini'],
              'max_depth': [15,20,25],
              'min_samples_split': [2],
              'min_samples_leaf': [2,10,20],
              'n_jobs':[-1]
             }
```

| | train | test |
|---|---|---|
| metrics | | |
| AUC | 0.964287 | 0.906041 |
| Accuracy | 0.906771 | 0.829469 |
| Precision | 0.901991 | 0.843383 |
| Recall | 0.953739 | 0.888024 |
| f1-score | 0.927143 | 0.865128 |



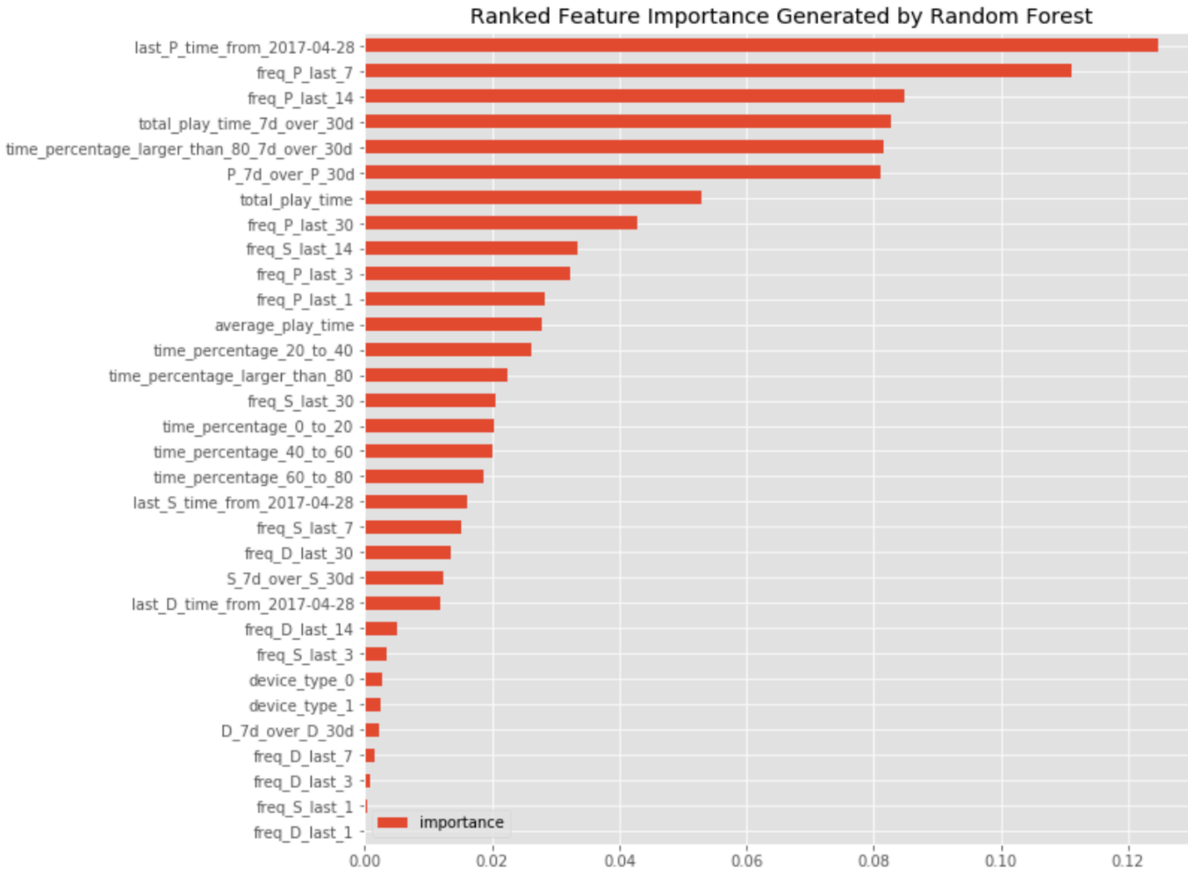

**AUC** of test data is **0.9060** with **Random Forest HyperParameter Tuning with Grid Search**, is close to that of previous **Random Forest** with **0.9061.**

**We select this model to explore the features importance to get some insights**

## 3. Explore features importance to get insights

### 3.1 Top 10 features analysis

The plot below shows the **ranked feature importance** generated by Random Forest:




As we can see, the **top 10 features** are:

**1. 'last_P_time_from_2017-04-28':** the larger the time gap between 'the last active day' and 'feature_window_end_date' is, the more likely the user will churn.

**2. 'freq_P_last_7':** the smaller play frequency in the last 7 days, the more likely the user will churn. And its feature importance is larger than those of 'freq_P_last_14', 'freq_P_last_30', which shows user recent behavior pattern is more informative.

**3. 'freq_P_last_14':** the same idea with **'freq_P_last_7'**

**4. 'total_play_time_7d_over_30d':** if the total play time acceleration ratio is less than 25%, means user had played less time in the last 7 days than average level of the last 30 days, the smaller the ratio is, the more likely the user will churn.

**5. 'time_percentage_larger_than_80_7d_over_30d':** if the time percentage acceleration ratio is less than 25%, means user had played less songs with high percentage in the last 7 days than average level of the last 30 days, the smaller the ratio is, the more likely the user will churn.

**6. 'P_7d_over_30d':** if the play frequency acceleration ratio is less than 25%, means user had played less frequently in the last 7 days than average level of the last 30 days, the smaller the ratio is, the more likely the user will churn.

**7. 'total_play_time':** the smaller total play time is, the more likely the user will churn.

**8. 'freq_P_last_30':** the same idea with **'freq_P_last_7'**

**9. 'freq_S_last_14':** the smaller search frequency in the last 14 days, the more likely the user will churn. Its feature importance is large perhaps because 14 days fits our label window which last 14 days better.

**10. 'freq_P_last_3':** the same idea with **'freq_P_last_7',** while its time window is too short to be as informative as 'freq_P_last_7'

### 3.2 Business model and stage analysis

As we are not told what business the music box is running, we will analyze different scenarios to figure out what we can do to reactive users with high churn probability:

**1．If it's running a freemium model:**
For free users with high churn probability, we can offer them one month paid version free trial to reactive them;
For paid users with high churn probability, we can offer them discount or even a freemonth;
**2． If it's running a paid model:**
For paid users with high churn probability, we can offer them discount or even a freemonth;

**Of course, we need to consider two factors before offering free trial, discount or freemonth:**
**1.** We should weigh the cost of doing so against the cost of acquiring another customer.
**2.** We should make sure whether our music box is in the 'Stickiness' stage to improve user engagement, or in the 'Revenue' stage to improve profits.

**Scenarios one:**
If music box is in the 'Stickiness' stage, it's reasonable to offer trial, discount and freemonth even when the cost of doing so overweighs the cost of acquiring another customer to improve user engagement.

**Scenarios two:**
If music box is in 'Revenue' stage and the cost of offering trial, discount or freemonth overweighs the cost of acquiring another customer, it's better to just sending e-mail and allocate more budgets to campaigns to attract new users and improve profits.

**We calculated the churn rate of our music box with 0.62.** As the churn rate is very high, we concluded it's in the 'Stickiness' stage, thus offering free trial, discount or freemonth is reasonable, and sending e-mail which is applicable to every business model is also good choice.

### 3.3 Insights

**By analyzing the top 10 features and business model above, we have some insights:**

**1.** Users' recent behavior pattern is more informative, we should pay more attention to last 7 days and last 14 days related matric, especially frequency features and acceleration features, like 'freq_P_last_14', 'freq_P_last_7', 'P_7d_over_30d', 'total_play_time_7d_over_30d' and 'time_percentage_larger_than_80_7d_over_30d'.

**2.** Once the time gap between 'the last active day' and 'feature_window_end_date' comes to 7 days, we should pay more attention to these users; once the time gap comes to 14 days, we should take some action to reactive them, like sending e-mail to recommend songs, offering paid version free trial, discount or freemonth.

**3.** We can also generate the churn probability of every user and rank to figure out the users most likely to churn, then send them e-mail to recommend songs, or offer them free trial, discount or freemonth to reactive them.

**4.** We can figure out our target users who are most unlikely to churn with our model, try to figure out what's common to this subsection of users, refocus on their needs, and grow from there.
To be specific, develop features which target users care, allocate campaigns budget to markets where our target users in.

**Generate the churn probability of every user and rank:**

| | uid | churn_probality |
|---|---|---|
| **23403** | 167579404 | 0.992385 |
| **34738** | 168074891 | 0.992385 |
| **9010** | 168060616 | 0.992385 |
| **26701** | 168063033 | 0.992297 |
| **5** | 167570658 | 0.992033 |

**3.4 Next step**

**Besides the insights mentioned above, I think there are aspects we can further dive deep, like:**

**1.** If we have a hypothesis that users churn because they don't like the songs we provide, we can analyze the songs played by churn users before they churned in a suitable time windows, try to figure out what's common to the songs driven users churn, and avoid providing these kinds.

**2.** It's also important to track performance over time. If we have more data, we can see whether we're improving or not, perform **cohort analysis** by comparing churn rate for each month.

**3.** Develop '**like**' feature to build a lock-in users experience, user can 'like' the music and keep it in their personal playlist, the more songs users keep, the stickier they will be, because there's a lot of data in place, so churn probability may be lower.

# C. Recommendation Models and Insights

## 0. Form rating data for recommendation system

We define **'rating'** as **max{'play_score', 'download_score'}:**

**'play_score'** are generate by 'play_time_percentage_of_song_length'.
The idea is that the larger played percentage is, the more likely the user like the song, rules as below:
0.8 <= 'play_time_percentage_of_song_length', assign 'play_score' 5
0.6 <= 'play_time_percentage_of_song_length' < 0.8, assign 'play_score' 4
0.4 <= 'play_time_percentage_of_song_length' < 0.6, assign 'play_score' 3
0.2 <= 'play_time_percentage_of_song_length' < 0.4, assign 'play_score' 2
0 <= 'play_time_percentage_of_song_length' < 0.2, assign 'play_score' 1
Note: If per uid per song_id has multiple ratings, we take average.

**'download_score'** are generate by whether user has download entry in feature window: 2017-03-30 ~ 2017-04-28.
The idea is that if a user downloads a song, he has great probability to like the song, rules as below:
If have download entry, assign 'download_score' 5
If no download entry, assign 'download_score' 0

## 1. Clean data and create utility matrix

We have rating data as below:

| | uid | song_id | rating |
|---|---|---|---|
| **0** | 10199495 | 22820742 | 2.0 |
| **1** | 104213634 | 6096002 | 1.0 |
| **2** | 104992781 | 127709 | 4.0 |
| **3** | 104992781 | 708290 | 4.0 |
| **4** | 10607827 | 22872742 | 4.0 |

**1.1 Clean data and create utility matrix**
df_recommender = df[['song_id', 'uid', 'rating']]
df_recommender.shape = (2539015, 3)

**1.2 downsample on user_id level**
Total number of users: 56694, **down_sample_ratio** = 0.05
total number of users after down sample: 2845
df_recommender_down_sample = (130791, 3)

There are many users that haven't play many songs, we build utility matrix with **only users played more than five songs**.
For the removed or new users, we can recommend **popular songs** at first.

**1.3 Create utility matrix from records**
2125x53728 sparse matrix, with 129407 stored elements in LInked List format

## 2. Build Recommenders and insights

**2.1 Popularity-based recommender**

For every new user or user played less than five songs, we build a Popularity-based recommender to recommend most popular songs at first.

We defined **'popular'** as songs with most played records.

Our **Popularity-based recommender** recommended top 10 songs: [2088, 5163, 6116, 5872, 6125, 15662, 5785, 6059, 5550, 1454].

**2.2 Neighborhood-based Approach Collaborative Filtering Recommender**

**2.2.1 Item-Item similarity recommender**

For **user played more than five songs**, we tried **Neighborhood-based approach** to build an **Item-Item similarity recommender** here.
Given a user_id and recommend 10 songs with **largest predictive rating** based on:
a. the similarity between items calculated using people's ratings of those items and
b. ratings user_id previous gave to the items.

## Item-Item based CF: How to make predictions

Say user *u* hasn't rated item *i*. We want to predict the rating that this user *would* give this item.

$$\text{rating}(u, i) = \frac{\sum_{j \in I_u} \text{similarity}(i, j) * r_{u,j}}{\sum_{j \in I_u} \text{similarity}(i, j)}$$

$$I_u = \text{set of items rated by user } u$$

$$r_{u,j} = \text{user } u\text{'s rating of item } j$$

We order by descending predicted rating for a single user, and recommend the top *k* items to the user.

We tried to get final recommendations for a user: user_number = 100, and our **Item-Item similarity recommender** recommended top 10 songs: [47185, 41924, 36516, 36517, 36518, 46249, 46248, 46247, 46246, 46245], with an average absolute error of **0. 9656**.

Then we tried to improve performance with **Matrix Factorization** approach to build recommender, because **matrix factorization models** always perform better than **neighborhood models in collaborative filtering.**

**The reason** is when we factorize a 'm*n' matrix into two 'm*k' and 'k*n' matrices we are reducing our "n"items to "k"factors, which means that instead than having our 50000 songs, we now have 500 factors where each factor is a linear combination of songs.
The key is that recommending based on factors is more robust than just using song similarities, maybe a user hasn't played the song 'stay' but the user might have player other songs that are related to 'stay' via some latent factors and those can be used.
The factors are called latent because they are there in our data but are not really discovered until we run the reduced rank matrix factorization, then the factors emerge and hence the "latency".

### 2.3 Matrix Factorization Approach Collaborative Filtering Recommender

#### 2.3.1 NMF

The **RMSE** of **NMF** is **1.3494**, and the **average absolute error** is **0.6121**, the performance is acceptable.

We tried to get final recommendations for a user: user_number = 100, and our **NMF recommender** recommended top 10 songs: [51347, 51348, 41708, 1170, 11460, 1666, 7873, 7144, 10250, 6837], with an **average absolute error** of **0.0137.**

The same as what we discussed above, the **average absolute error** of **NMF** for this specific user is better than **0.9656** of **Item-Item similarity recommender**.

Then we tried **UVD** to verify whether it performs better than **NMF**.

#### 2.3.2 UVD

The **RMSE** of **UVD** is **1.1729** and the **average absolute error** is **0.5569**, which are better than scores of NMF(**1.3494** and **0.6121**).

**UVD** performs better because it has **larger degree of freedom** than NMF, to be specific, NMF is a **specialization** of **UVD**, all values of V, W, and H in NMF must be **non-negative**.

Then we tried to get final recommendations for a user: user_number = 100, and our **UVD recommender** recommended top 10 songs: [10718, 6837, 4440, 10605, 21281, 1170, 21562, 51347, 51348, 6422], with an **average absolute error** of **0.0391**, which is very close to **0.0137** of **NMF.**

### 2.4 Specific recommendation results comparison and insights

Then we compared the recommendation results for 'user with user_number = 100' generated by the four models above to get insights.

We generated the overlap tables of the top_10 and top_100 results given by the four models for 'user with user_number = 100' as below:

```
The overlap of the top 10 recommendation generated by these four models
```

| | number_of_same_recommended_songs_in_top_10 |
|---|---|
| Popularity-based_with_Item-Item | 0 |
| Popularity-based_with_NMF | 0 |
| Popularity-based_with_UVD | 0 |
| Item-Item_with_NMF | 0 |
| Item-Item_with_UVD | 0 |
| NMF_with_UVD | 4 |
| Popularity-based_with_Item-Item_with_NMF_with_UVD | 0 |

```
The overlap of the top 100 recommendation generated by these four models
```

| | number_of_same_recommended_songs_in_top_100 |
|---|---|
| Popularity-based_with_Item-Item | 0 |
| Popularity-based_with_NMF | 3 |
| Popularity-based_with_UVD | 1 |
| Item-Item_with_NMF | 0 |
| Item-Item_with_UVD | 0 |
| NMF_with_UVD | 66 |
| Popularity-based_with_Item-Item_with_NMF_with_UVD | 0 |

**From the overlap tables above, we notice that:**

**1.** The recommended songs given by **Popularity-based**, **Neighborhood-based** approach and **Matrix Factorization** approach models are very different from each other, have no overlap in top 10 and only 3 overlaps in top 100 recommended songs.

**2.** While the recommended songs given by **NMF** and **UVD** have 4 overlaps in top 10 and 66 overlaps in top 100 recommended songs.

**Conclusion:**

**1.** For new user or user played less than five songs, we can recommend most popular songs at first generated by our **Popularity-based recommender**.

**2.** For user played more than five songs:
**2.1** Given the performances of **NMF** and **UVD** are comparable, we can have the overlap of commendation results generated by these two models as the final recommendation.

**2.2** As the results generated by **Popularity-based**, **Neighborhood-based** Approach and **Matrix Factorization** Approach models are totally different, we can allocate different weights to these models to construct the final recommendation.
Like 0.2 for **Popularity-based**, 0.2 for **Neighborhood-based**, 0.6 for overlap of **NMF** and **UVD**.

**2.5 Next step**

**Besides the insights mentioned above, I think there are aspects we can further dive deep, like:**

**1.** As we have huge amounts of users, we can try to perform clustering to all users, cluster users with high similarities into the same cluster, which allows us to perform different recommendation algorithm to different clusters, more efficient and more targeted.

**2.** Develop **'dislike'** feature which allow users to flag songs they don't like, so that our recommender will not recommend the disliked songs again, on the other hand, our recommender can avoid recommending songs with high similarities with disliked songs.

**3.** Our recommendation system should also consider the style of the song, such as recommending rock or pop music in working hours, recommending light music or antiques in evening time, etc.