KKBOX's Music Recommendation Challenge Solution with Feature Engineering

Jianyu Zhang Françoise Fogelman-Soulié
School of Computer Software
Tianjin University
(edzhang, soulie) @tju.edu.cn

ABSTRACT

Recommendation is widely used in our daily life. Especially in the e-commerce area, a good recommendation system can help users a lot. In this paper, we introduce our approach for the KKBOX's Music Recommendation Challenge. In this challenge, we were asked to build a recommendation system that can predict whether a user will listen again to a song within one month after the user's very first observable listening event in KKBOX. Our solution was mostly based upon systematic and extensive feature engineering and an ensemble of simple boosting tree classification algorithms, both of which could easily be used in industry. However, we did not use timestamp of user-song interactions here, since this was hidden by Kaggle to avoid leakage.

KEYWORDS

Feature engineering, recommender systems, gradient boosting tree, SVD.

1 INTRODUCTION

In the WSDM Cup KKBOX's Music Recommendation Challenge [6], we had to build a recommendation system that can predict whether a user will listen to a song again within one month after the user's very first observable listening event in KKBOX. If the user did not listen to the song again within one month, the target variable will be 0, and 1 otherwise. The training and test sets consist in unique user/song pairs selected from users' listening history in a certain period, split by time. The test set is split 50/50 between public and private leaderboards; obviously, targets are unknown in the leaderboards test set, but the data distribution is, and we can thus use it as we will show later. At the time of the competition, submissions were evaluated on the public leaderboard, while final results were obtained on the private leaderboard. As usual in such challenges, one must take particular care to avoid learning the public leaderboard too well at the risk of *overfitting* and obtaining degraded results on the private leaderboard (this actually happened to the top four winners on the public leaderboard, the fourth, ekffar, falling to fifth rank in the private leaderboard).

In this competition, the organizer also provides three attributes of the user-song interaction context, as well as attributes of each song and user. The use of *public data* was encouraged by the organizers; however, we did not find public datasets we could efficiently use within the challenge time-period.

There is no explicit *timestamp* information for the first listening event of a user-song pair. However, the order of examples in the training and test datasets itself is time-ordered (apparently, the Kaggle organizers did not shuffle the datasets). Using this time-ordered information and new registration information, we could approximately extract the timestamp of the first listening event of a user-song pair. However, the organizers said they removed the timestamp feature to avoid leakage. Therefore, we did not use anywhere this timestamp information in our approach.

Feature engineering is a critical step in the data science process, which comes right before the modeling stage. It is one of the most important and time-consuming tasks in predictive analytics projects. Its purpose is to design, from the raw data, features that will make the model easier and faster to train, and increase its performances. In practice, almost all the winners in recent Kaggle competitions have extensively used feature engineering, and put a lot of time and energy in designing these features. For example, in an extreme case, winners in the *Grupo Bimbo Inventory Prediction*[1] reported that they spent 95% of their time on feature engineering and only 5% on modeling (details can be found in the challenge blog [2]). In the *Outbrain Click Prediction*[3] kaggle challenge, it was possible to get 19th position by the help of feature engineering [2]. Feature engineering involves deep data exploration to understand their specificities, as

WSDM 2018, February 5-9 2018, Los Angeles, California, USA
© 2018 ACM. ISBN xxxx. . . $15.00
DOI: http://dx.doi.org/xxxx

[1] https://www.kaggle.com/ben519/grupo-bimbo-inventory-demand/visualize-predictions
[2] http://blog.kaggle.com/2016/09/27/grupo-bimbo-inventory-demand-winners-interviewclustifier-alex-andrey/
[3] https://www.kaggle.com/c/outbrain-click-prediction

well as domain knowledge to create meaningful and helpful features. Until now, feature engineering is still more art than science. However, some authors [3], [4] have proposed methods to automate the process of feature engineering, which would certainly be a huge advantage for Kaggle challenges, and more generally, to any predictive analytics project. Our approach for the WSDM Cup aligns with these objectives: we tried to generate as automatically as possible a large number of features. Careful evaluation of generated features allowed us to progressively increase performances. We used very simple vanilla models.

So feature engineering is a really time-consuming part of data mining, requiring strong data mining and statistics backgrounds as well as domain knowledge, but can improve performances significantly. Automatic feature engineering will certainly happen in the future.

This paper describes the approach of the team EdZhang in the WSDM Cup in section 2 and the results obtained in section 3. The team obtained rank 6 in the challenge.

2 APPROACH

2.1 Dataset description

In this music recommendation competition, there are five csv files. The structure of the dataset is really simple and is shown in Figure 1. The train.csv and test.csv contain user ID, songs ID and three context attributes. The other three csv files include attributes of each song and user. We can merge all these attributes into train.csv and test.csv to generate a union table.

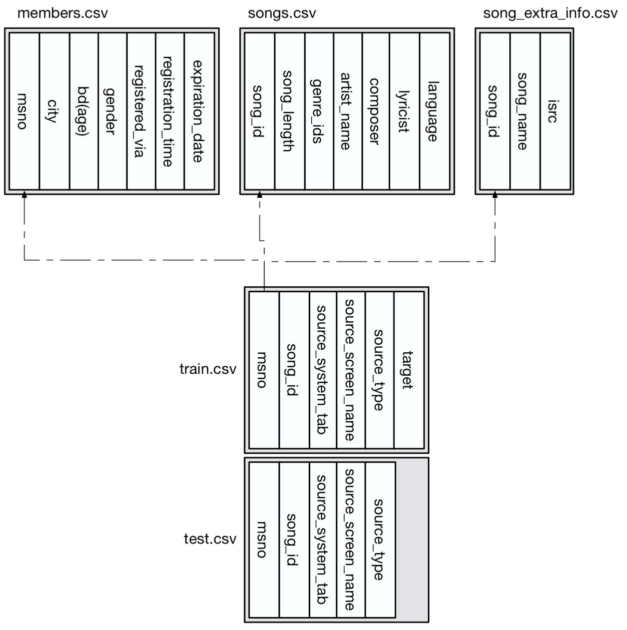

Figure 1: Structure of the dataset in KKBOX's Music Recommendation Challenge. Light gray square indicates csv table in the dataset, white square indicates one attribute in the csv table and the link between two tables indicates that two tables can be merged by that attribute.

The number of features in each csv file is also very small. Table 1 shows the details of the count of features in each csv file, where user ID and song ID are included in train.csv and test.csv.

Table 1: Description of csv file in the dataset

Name of file	Count of Features
train.csv	6
test.csv	5
songs.csv	6
members.csv	6
song_extra_info.csv	2

In the following sections, we will use the following notations, described in Table 2.

Table 2: Notations

Notation	Description
m, n	Number of users, songs
M^e	Co-occurrence matrix of user and e, where e can be $entity$ song, or artist
$M^e_{i,j}$	The value in i^{th} row and j^{th} column of matrix M^e, e.g. number of times user i listened to song j
$M^e_{i,-}$	The i^{th} row of matrix M^e.
$M^e_{-,j}$	The j^{th} column of matrix M^e.
F(M^e)	A function applied on matrix M^e.
L	The size of latent space of SVD.
N_a	Neighborhood of user a.
$F_{name\_feature}$	A new feature or a new groups of features

2.2 Exploration

Data exploration is the first step in a data mining project and heavily depends on the dataset. It is important to give an idea of how to deal with the task. In Figure 1, we can see that the structure of our dataset is not very complex. Therefore, we merged all the tables into one union table before doing the data exploration described below.

2.2.1 Count of values in each feature

Because most of the features in the dataset are categorical features, we calculate the number of different values of each categorical feature as shown in Table 3. The counts of user, song, artist_name, composer, lyricist are very large, so one-hot encoding will not be a suitable preprocessing method because of the curse of dimensionality and the cost of memory and computation. In this situation, choosing other encoding methods or finding some representations of these features will help the model.

2.2.2 Missing values

Understanding and handling missing values can also heavily influence the performance of a predictive model. We make a brief summary of the observed missing value rates in Table 3. Composer, lyricist and gender have high missing value rates compared with the other features. Because in this challenge, competitors could use external data, these features would be good candidates to search external data. Although we did not use any external data because of lack of time, it would be useful to enrich, through external data, the features with high missing value rates. Furthermore, the difference of missing value rates among different features also leads us to generate more features for artist_name than for composer and lyricist.

Table 3: Count of values and missing value rate of different features

Name	Type	Count of values	Missing value rate (%)
msno	categorical	34,403	0
song_id	categorical	419,839	0
source_system_tab	categorical	9	0.236
source_screen_name	categorical	22	5.815
source_type	categorical	12	0.290
song_length	numeric	-	0.001
genre_ids	categorical	608	1.616
artist_name	categorical	46,372	0.001
composer	categorical	86,438	23.102
lyricist	categorical	37,876	44.327
language	categorical	10	0.002
city	categorical	21	0
bd (age)	numeric	-	0
gender	categorical	2	40.403
registered_via	categorical	6	0
registration_init_time	timestamp	-	0
expiration_date	timestamp	-	0
song name	text	-	0.023
isrc	categorical	315,966	7.796

2.2.3 Training and test sets

We also want to see the difference between training and test sets, because most data mining algorithms have a basic assumption that the training and test sets must have the same distribution (identically distributed). There are 7,377,418 examples in the training set and 2,556,790 examples in the test set, which is a ratio of almost 2.88 to 1. For the target in the training set, the ratio of the number of positive examples to the number of negative examples is about 1, so we have a balanced binary classification task. We check the distributions of the training and test sets on the various attributes. For example, we compare the count of each value in the context features (source_screen_name, source_system_tab, source_type) between training and test sets. In Figure 2, we can see that the distributions of the training and test sets are really different; for example, the difference between the percentage of the value "my library" of source_system_tab in the training set is 10% larger than in the test set. This situation also occurs for the other context features source_sceen_name and source_type. Because of such large differences, it is in theory not allowed to apply on test set a model trained on the training set. This is why we did feature engineering on the training and test sets together, as will be shown later, which is certainly not a good choice in the normal i.d. situation.

There are 35,996 different songs in the training set, 224,753 in the test set and 59,873 different songs only exist in the test set. This means that the test set has 26.64% of new songs, usually called *cold-start*. These songs are present in 12.52% of the test examples. This is a big challenge in this task. The cold-start problem also exists for users. There are 30,755 different users in the training set and 25,131 different users in the test set where 3,648 only exist in the test set (14.52% cold-start users, appearing in 7.20% of test examples), which is shown in Figure 3. The cold-start problem in this task is so significant that we need to put even more care on getting a robust model to avoid overfitting.

Moreover, some user-song pairs exist in both training and test sets. Because if a user listens the same song after a long time (e.g. two months), the system will count that user-song pair again. In total, the count of this kind of user-song pairs is 320,446, which is 12.53% of the test set. These user-song pairs are the same in the training and test sets in the attributes level, but their meaning in training and test are totally different because there is a hidden time-order between training and test sets. This kind of situation also challenges us to make a robust model so as to avoid overfitting.

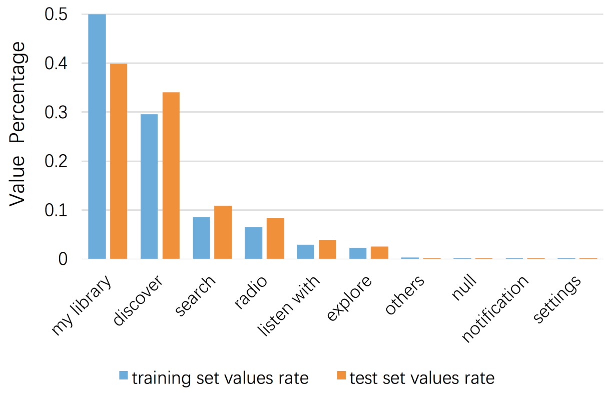

Figure 2: Count of each value in source_system_table in the training (blue) and test (orange) sets. X axis shows different values in source_system_tab, Y axis shows the percentage of examples in training / test sets having this value.

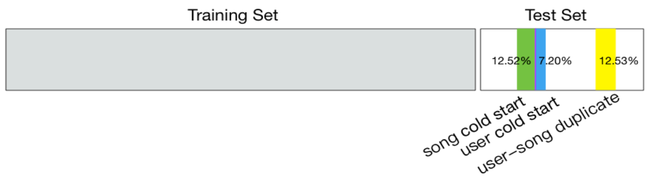

Figure 3: Structure of training and test sets. The light grey indicates the training set and the white part indicates the test set. "user-song duplicate" indicates the user-song pairs that exist both in the training set and test set. "user cold start" and "song cold start" indicate the user or song that never occur in the training set.

2.3 Feature engineering

2.3.1 Attributes of song

There are two csv files related with the raw features of song (song.csv and song_extra_info.csv). We extract several features from song_extra_info.csv.

Language from song name: we use the langid package https://pypi.python.org/pypi/langid to detect the language associated to the song name and, for each song s, derive feature :

$$F_{Language\_Song}(s) \tag{1}$$

Year and country of song from ISRC: from Wikipedia, we know that the first two characters of The International Standard Recording Code (ISRC) is a two-character country code and the last two characters of ISRC represent the last two digits of the reference year. We extract the year of song and the country code of song from ISRC, for each song s:

$$F_{Year\_Song}(s) \quad F_{Country\_Song}(s) \tag{2}$$

Statistic features of genre, artist, composer, lyricist: Because some songs could contain more than one genre, artist, composer and lyricist. We calculate the count of genre, artist, composer and lyricist in each song. In some songs, the Artist, composer and lyricist could be same. We generate two features that indicates if the artist and composer in a song are the same one, if the artist, composer and lyricist in a song are the same one.

$$\begin{gathered} F_{genre_{len}}(s), F_{artist_{len}}(s), \\ F_{composer\_len}(s) \\ F_{lyricist\_len}(s) \\ F_{artist\_composer\_same}(s) \\ F_{artist\_composer\_lyricist\_same}(s) \end{gathered} \tag{3}$$

2.3.2 Attributes of user

Age of users: in data exploration, we can see outliers in the age attribute, including small age (age=0), negative age (age<0) and large age (age >130). In this approach, we set to 0 all ages less than or equal to 0, and set to 76 all ages larger than 75. For each user a we define feature:

$$F_{Age\_User}(a) \tag{4}$$

Registration and expiration date of users: from the description of the competition, we know that the time block of the dataset is before the time block of KKBOX's Churn Prediction Challenge [7] where the log history goes from 2015-01-01 to 2017-03-31. We thus generate two features computing how many days there are between registration date and 2017-03-31, and between expiration date and 2017-03-31. In addition, we generate the year, month of year and day of month for the registration and expiration dates. For each user a we define features: Registration and expiration date (user)

$$\begin{gathered} F_{Registration\_User}(a),\ F_{Registration\_User\_Y}(a), \\ F_{Registration\_User\_MY}(a), F_{Registration\_User\_DM}(a), \\ F_{Expiration\_User}(a), F_{Expiration\_User\_Y}(a), \\ F_{Expiration\_User\_MY}(a), F_{Expiration\_User\_DM}(a) \end{gathered} \tag{5}$$

Age gap between user and song: this feature represents the gap between the age of users and the year of a song:

$$F_{AgeGap\_UserSong}(a, s) \tag{6}$$

In real life, young people usually prefer newer songs, while older are more likely to listen to older songs.

Count of songs each user listened: this feature indicates how many different songs each user listened.

$$F_{Count\_UserSong}(a, s) \tag{7}$$

2.3.3 Representation features

Because of the strong interaction between user and song, we should certainly derive interaction-based features. However, since the dimension of user and song is large, we cannot encode the behavior of user or song by One-Hot encoding. This requires us to find adequate representations for user, song and user-song pairs. In the recommendation area, neighborhood based Top-N collaborative filtering (CF) and model based matrix factorization (MF) are two strong and powerful families (see [1]) which we could use to generate features. The item-based CF will compute the Top-N most similar items for each item based on some similarity measure (e.g. cosine similarity, Jaccard similarity, etc.) as the neighbors of that item. Then it will compute a score of each user-item pair based on what the user had purchased and neighbors of the item. Similarly, the user-based CF will construct the Top-N most similar users for each user as the neighbors of that user. On the other hand, matrix factorization method projects both user and item into a latent space of low dimension. In that space, a user will be close to the items that he had purchased. The most efficient MF method is the Singular Value Decomposition (SVD) ([5]).

SVD representation of user and songs. To do SVD matrix factorization we used sparsesvd (a Python package[4]), which is a wrapper around the SVDLIBC[5] library by Doug Rohde. The sparsesvd package can handle SciPy's sparse Compressed Sparse Column (CSC) matrix format, so it is memory-efficient.

$$\mathrm{M} \approx \mathrm{U}.V^T \tag{8}$$

Where M is an $m{\times}n$ input matrix, U an $m{\times}k$ matrix, V an $n{\times}k$ matrix, k is the dimension of the SVD latent space.

For the input part, we use all the user-song interactions in the train set and test except the duplicate interactions to construct the user-song matrix $M = M^{song}$. As discussed in section 2.2.2, the distributions of training and test sets are very different. To mitigate this effect, we chose to do feature engineering on the combination of both training and test sets. This means that in the user-song matrix M^{song}, 0 shows that the user did not listen to the song, 1 indicates that he did at least once. Because we introduce

[4] https://pypi.python.org/pypi/sparsesvd/
[5] http://tedlab.mit.edu/~dr/SVDLIBC/

some leakage from the test set into the training data, we need to be extra careful about robustness and overfitting. We will see how below.

The dimension k of the SVD latent space is a hyper-parameter we evaluated by cross-validation: in our final results, we choose $k = 30$. Using the latent space representations computed in equation (1), we generate one 30-dimension user representation and one 30-dimension song representation. We also compute the dot product of these two representations as a representation of the user-song pair.

Moreover, we multiply each dimension of user representation and song representation to generate another 30-dimension user-song pairs representation. These features can indicate how similar of user and song in each dimension of the latent space.
So we generate totally 30+30+30+1=91 new features here:

$$F_{SVD\_UserSong}(a,s) = (\hat{a}, \hat{s}, \hat{a}.\hat{s}, \hat{a} \overline{\overline{\times}}\, \hat{s}) \tag{9}$$

Where $\hat{a} = U_{a-}$, $\hat{s} = V_{s-}$, $\hat{a}.\hat{s}$ is the dot product of projections $\hat{a}$ and $\hat{s}$, and $\hat{a} \overline{\overline{\times}}\, \hat{s} = (\hat{a}_1 \times \hat{s}_1, \dots, \hat{a}_{30} \times \hat{s}_{30})$.

CF score of each user-song pairs: Top-N CF is another main family of recommendation methods system. We use user-based CF rather than item-based CF, because there are less users than songs in the dataset as shown in Table 3. We compute cosine similarity between all different user pairs based on user-song matrix M^{song_id}:

$$cosine(a,b) = \frac{\alpha \cdot \beta}{||\alpha|| \cdot ||\beta||} \tag{10}$$

Where α and β are the two vectors of users a and b in matrix M^{song_id}, $\alpha = M^{song\_id}_{a,-}$, $\beta = M^{song\_id}_{b,-}$. As before, we use all the user-song interactions in the training and test sets except the duplicate interactions to construct user-song matrix M^{song_id}.

For each user a, we select as neighborhood N_a of a the top 100 users with highest cosine similarity with user a. For each user-song pair (a,s), we then generate the Top-N CF score as :

$$F_{Top-N\_CF\_score}(a,s) = \frac{1}{m_a} \sum_{u \in N_a} M^{song\_id}_{u,s} \, . \, cosine(a,u) \tag{11}$$

Where m_a is the number of users in N_a.

SVD representation of user-artist pairs: It is similar to the SVD representation on user-song pairs. The difference between user-song pair and user-artist pair representations is that for each song there might be more than one artist.

1. We Split the artist attributes by separators (, | \\ & \ / + ; ， and feat. Features featuring with X) (different separators are separated by space). One song can have more than one artist, which are separated mostly by these separators. That is the reason we choose these separators here.
2. We compute a user-artist matrix M^{artist} where M^{artist}_{aj} indicates the number of songs that contain artist j which user a has listened to. We also eliminate all the duplicate user-song pairs before this process.
3. We decompose matrix M^{artist} by SVD (with latent space dimension equal to 180: we choose this size by testing the Pearson Correlation Coefficient between the feature and target values). Finally, we use dot product of user and artist representations to generate a 1-dimension score of user-artist pairs. Because one song will contain more than one artist, we use the maximum values of user-artist pairs for each user-song pair, when we use this user-artist feature.

Through this process, we generate one new feature:

$$F_{SVD\_UserArtist}(a,s) = \max_{j:j \in s}(\hat{a}.\hat{j}) \tag{12}$$

Similarity score between user and song attributes (genre, artist_name, lyricist, composer, language): we want to know how similar are songs a user listened to and song he is listening to now. The following is the workflow of this process. Let us take artist_name for example. We count how many times a user listened songs by that artist. In other words, we generate a count of word vector for each user, where each word indicates one artist. Combining all these count of word vectors, we get a user-artist matrix $M^{artist}_{a,-}$ for user a.

Let (a,s) indicate a user-song pair, we denote A^{artist}_s as the one-hot encoding vector of feature artist of song s. For each user-song pair (a,s), we generate similarity features S as follows:

$$F_{im\_UserSong\_artist}(a,s) = \frac{\left(M^{artist}_{a,-} - A^{artist}_s\right)}{\sum_j \left(M^{artist}_{aj} - A^{artist}_{aj}\right) \cdot \left(A^{artist}_{sj} / \sum_j A^{artist}_{sj}\right)^T} \tag{13}$$

Actually, this similarity measures how similar a particular user's transaction is with all his other transactions. All the other similarity features between user and the attributes of song are generated in the same way. We generate 5 features in this process with x=(genre, artist_name, lyricist, composer, language):

$$F_{sim\_UserSong\_x} \tag{14}$$

Similarity score between user and context attributes: the method to generate this group of features is the same as the previous one. The only difference is that in this part we measure the similarity of the context information inside each user.
We generate 3 features here, with x=(source_system_tab, source_screen_name, source_type):

$$F_{sim\_UserContext\_x} \tag{15}$$

User representation of context information: from Figure 2, we found that context information is important in this challenge. This gives us the idea of generating more features for these context features. Take source_system_tab for example here, we will generate user-source_system_tab co-occurence matrix $M^{source\_system\_tab}$ in the same way we generated M^{artist} in the SVD representation of user-artist. Then we normalize $M^{source\_system\_tab}_{a,-}$ by $\sum M^{source\_system\_tab}_{a,-}$.

We thus get a user-source_system_tab feature for each user a:

$$F_{UserContext\_source\_system\_tab}(a) = \frac{M_{a,-}^{source\_system\_tab}}{\sum_j M_{a,j}^{source\_system\_tab}} \tag{16}$$

We generate representation features for the other context features in the same way. Totally, we generate 46 features here:

$$F_{UserContext\_x}(a) \tag{17}$$

with x = (source_system_tab, source_screen_name, source_type). We thus have defined a total of 167 features, shown in Table 4, out of which 91 come from the SVD representation of (user, song) pairs:

Table 4: Description of features generated by feature engineering

Name	Notation	N°
Language (song)	$F_{Language\_Song}$	1
Year and country (song)from ISRC	F_{Year_Song} $F_{Country\_Song}$	2
Statistic features of genre, artist, composer, lyricist	F_{genre_len} F_{artist_len} $F_{composer\_len}$ $F_{lyricist\_len}$ $F_{artist\_composer\_same}$ $F_{artist\_composer\_lyricist\_same}$	6
Age (user)	F_{Age_User}	1
Registration and expiration date (user)	$F_{Registration\_User}$ $F_{Registration\_User\_Y}$ $F_{Registration\_User\_MY}$ $F_{Registration\_User\_DM}$ $F_{Expiration\_User}$ $F_{Expiration\_User\_Y}$ $F_{Expiration\_User\_MY}$ $F_{Expiration\_User\_DM}$	8
Age gap between (user-song)	$F_{AgeGap\_UserSong}$	1
Count of songs user listened (user-song)	$F_{Count\_UserSong}$	1
SVD rep. (user-song)	$F_{SVD\_UserSong}$	91
CF score (user-song)	$F_{Top-N\_CF\_score}$	1
SVD rep. (user-artist)	$F_{SVD\_UserArtist}$	1
Similarity score (user, song attributes)	$F_{sim\_UserSong\_x}$	5
Similarity score (user, context attributes)	$F_{sim\_UserContext\_x}$	3
User rep. (user, context)	$F_{UserContext\_x}$	46

2.4 Model Training

We obtained our final results through an ensemble of two gradient boosting tree models from LightGBM[6] package with different parameters.

The only preprocessing applied to the features described in the previous section was filling missing values and using Label-Encoder (sklearn) to map string variables to numeric ones. We fill missing value for categorical features by a new category we call "others" and for numeric features by "-1". Since missing values are rarely MAR (missing at random), it is preferable to avoid filling in missing values by mean, median or most frequent categorical variable. This is even more so because we are using tree-based method. One-hot encoding for categorical features and normalization for the numeric features do not help improving tree-based method. Moreover, one-hot encoding costs much more memory and computation time. The idea of filling in missing values by "others" or "-1" is that tree-based methods can handle different variables in one feature very well. In this section, we will introduce how we build our two models and how we blend them into one unique model to get better results. When we did this challenge, the version of sklearn was 0.19.1, the version of LightGBM 2.0.10 and the version of langid 1.1.6.

2.4.1 LightGBM model 1

Training set and Validation set. As shown in paragraph 2.3.2, there are 320,446 user-song pairs (12.53% of test set), which exist both in training and test sets but with different meanings. For these user-song pairs, target 1 in the training set means the user listened to that song again within one month after the user's very first observable listening event. However, target 0 in the training set means the user listened to that song again but not within one month. Therefore, when we train the model, we always eliminate from the training set the duplicate user-song pairs with target variable 0. We thus eliminate 173,931 training examples after this step.

Validation set helps us evaluating the performances of our model offline, tuning parameters and avoiding overfitting on public leaderboard. It is very important to use a suitable validation set here to avoid overfitting by being exposed too often to the public leaderboard.

Features used. All the features generated in section 2.3 are used in our models. All the other user raw features (msno, city, gender, registered_via), song raw features (song_id, song_length, genre_ids, artist_name, composer, lyricist, language) and context raw features (source_system_tab, source_system_name, source_system_type) are used in our models too.

Hyper-parameters. Table 5 shows the hyper-parameters used in our LightGBM model 1 (first column). For the other hyper-parameters not listed here, we simply use default values.

Training. In our approach, we used 10-fold cross-validation (sklearn). Through stratified sampling, we draw from the training set 10 samples (folds) with 90% is used for training and 10% for

[6] http://lightgbm.readthedocs.io/en/latest/index.html

validation. Each fold has almost equal numbers of positive and negative examples. Then, we train one LightGBM model on the training part of the cross-validation fold, and evaluate it on the validation part.

Table 5: Values of hyper-parameters of LightGBM models

Parameter	LightGBM Model 1	LightGBM Model 2
learning_rate	0.05	0.1
max_depth	15	15
num_leaves	2^8	2^8
application	binary	binary
colsample_bytree	0.7	0.9
subsample	0.7	0.9
num_boost_round	3100	3100
early_stopping_rounds	10	10

Blending of 10-fold cross-validation. We have obtained 10 different models with different training and validation sets in our 10 cross-validations samples. We blend these 10 models by simply taking an average of the prediction scores of the obtained 10 models. We give an equal weight to all these 10 models in this blending step.

One should note that, in this fashion, we can evaluate the significance of features without submitting our results on the public leaderboard.

2.4.2 LightGBM model 2

Training set and Validation set. The methods of preprocessing and splitting training and validation sets are the same as for our LightGBM model 1. The only difference here is that we use different parameters of LightGBM algorithm.

Feature used. It is the same as §2.4.1, all the features generated in session 2.3 and user raw features, song raw features, context raw features are used here.

Hyper-parameters. Table 5 (right column) shows the hyper-parameters used in our LightGBM model 2. For the other hyper-parameters not listed here, we simply use default values.

Training and blending. As for LightGBM model 1.

2.4.3 Ensemble model of LightGBM model 1 and LightGBM model 2

After LightGBM model 1 and Light GBM model 2 with their different hyper-parameters have been trained and blended, we make an ensemble of these two models together by simply taking the average of the prediction scores of model 1 and model 2. This generates a stronger model.

With more time, more sophisticated methods could certainly be used for getting better ensembles.

3 EVALUATION RESULTS

For our final result with an ensemble of two LightGBM models, we got an AUC score 0.72930 on the public leaderboard and increased the AUC score to 0.73015 on the private leaderboard, getting us 6th position. This means we did not overfit the public leaderboard. To achieve this, we mainly used three tactics:

4. *Early stopping*: as shown in Table 5, we trained our models in a few iterations. As usual, this regularization technique helps getting robust models.
5. *Use private validation set*. As discussed in sections 2.4.1 and 2.4.2, we validated our models on our private validation set (in each cross-validation fold). We submitted our models to the public leaderboard parsimoniously to avoid learning it too much.
6. *Progressive increase of feature sets*. We used a staged approach to incorporate features. When features did not bring performance increase on our validation set, we dropped them.

In this section, we will now compare the performances of the models presented in section 3 and show the effect of blending and ensemble.

In Table 6, we show the public and private leaderboards AUC scores of our two LightGBM models (1 and 2), blended with 10 cross-validation folds and without blending. Of course, time consumption of a single model is much smaller than the cost for the ensemble (around 1/10 time cost of the final model, we show the time cost of each model in Table 6 too), while performance is degraded but still relatively good.

In Table 6, the AUC performance of the best single model (LightGBM model 1 without 10-fold cross-validation) is 0.72787, which is not far from the performance of the final ensemble model (0.73015). The performances of the blended LightGBM model 1 and LightGBM 2 (with 10-fold cross validation) on private leaderboard are 0.72930 and 0.72893.

Table 6: AUC Score and Time Cost of Different Models

Name	Public LB	Private LB	Time (min)
LightGBM model 1 without 10-fold cross-validation	0.72684	0.72787	60
LightGBM model 2 without 10-fold cross-validation	0.72268	0.72350	62
LightGBM model 1 (blend)	0.72892	0.72930	603
LightGBM model 2 (blend)	0.72797	0.72893	622
Ensemble of models 1 and 2	0.72930	0.73015	1225

The interesting part is not only the final private and public leaderboard scores themselves, but the step-by-step process of how this score is progressively increased. In our approach, we find that as we add more and more suitable features, the performance will increase step-by-step. Some features can even give us a really large improvement. We show three main big improvements of our approach in Figure 4: the blue line shows the increasing AUC score in public leaderboard. There are mainly three big jumps in this process marked by A, B, C. In jump A, we mainly add SVD representations of user and songs features and tune the hyper-parameter of SVD representation. In jump B, we mainly blend the results of 10-fold cross-validation. In jump C, we mainly add similarity score features, CF features and SVD

user-artist representation features. As we add more and more suitable features, the AUC performance increases step-by-step.
Not all features helped. We tried time-based features, which did not work well, maybe because the precise timestamp was not provided. As we said, we evaluated features and only submitted our model with them to public leaderboard when they significantly improved performances on our private validation. Otherwise, we just dropped them from our candidate list.

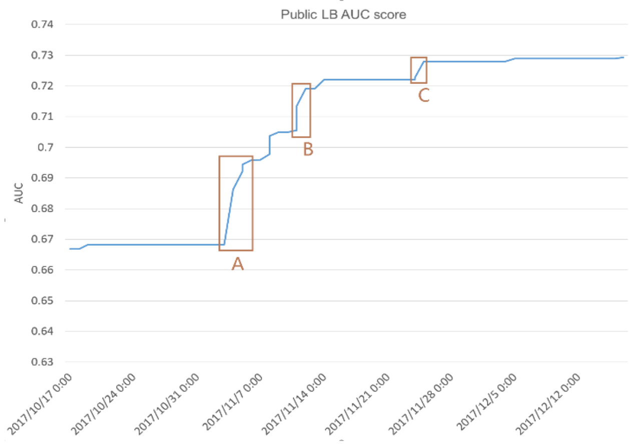

Figure 4 : Public LB AUC score along time

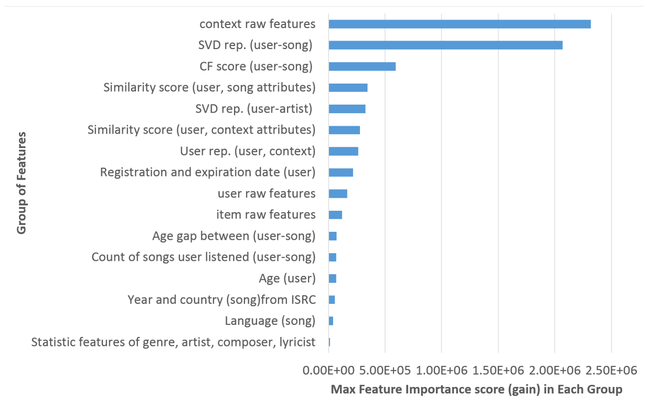

Figure 5: Feature importance

We also want to compare the importance between different group of features that we used in our final LightGBM model. In the Figure 5, we show the maximum Feature importance score (gain) in each feature group. All these feature groups are used in our final LightGBM model. They were introduced in Table 4 and §2.4.1. The context features are the most important features in this task. SVD representation features also have a very high importance score. That is the most important group of features in this feature engineering process.

As the description of our approach shows, we spent most of our efforts on feature engineering, and used plain, simple, out-of-the-box models (LightGBM) or ensemble techniques (average). Obviously, better models might have obtained better results. However, it is our strong belief that, in a limited timeframe, efforts are better spent on feature engineering than on sophisticated models.

4 CONCLUSION

Feature engineering is a really important and irreplaceable ingredient in a predictive analytics project. Especially when the size of dataset is not very large, feature engineering is extremely helpful to improve performances in most cases. Some features from feature engineering did not help or even hurt the performance of the model. However, with our process of progressive validation, we just dropped them. Obviously, when datasets are large, time costs may become an issue, since in our approach, we generate features to then evaluate and possibly retain them. The balance of costs versus performance increase has to be carefully evaluated in such cases.

Because of different distributions between training and test sets, we used features computed on both training and test sets, thus actually leaking information from the future into our model (a practice certainly not recommended in general). This obviously might hurt performances on test set. For example, if we generate a feature for counting the number of users listening to each song, the performance will increase in validation set but decrease in test set. The reason is that the distributions of this feature between validation and test sets are significantly different. However, we had to mitigate the impact of the differing distributions taking that leakage risk when handling the feature engineering process. To limit this risk, we need to take extra care to avoid overfitting.

What is left to do for challenges such as this one includes generating more time-dependent features to get better performances and certainly using neural network-based methods for their capacity to generate features, but we did not have time to complete our work on neural networks in this challenge.

In our future work, we will try to make the feature engineering process more and more automatic and apply this automatic feature engineering process to other challenges.

ACKNOWLEDGMENTS

We want to thank WSDM, KKBOX and Kaggle for holding this recommendation challenge, which allowed us to apply our ideas in practice and learn a lot throughout this process.

REFERENCES

[1] Daniel Bernardes, Mamadou Diaby, Raphaël Fournier, Françoise Fogelman Soulié, Emmanuel Viennet. 2014. A Social Formalism & Survey for Recommender Systems. SIG KDD Explorations, Vol. 16, Issue 2, pp. 20-37, December 2014.
[2] Gabriel Moreira. 2017. How feature engineering can help you do well in a Kaggle competition. KDNuggets News, Vol. 17, n°23, 25, 27. June and July 2017. https://www.kdnuggets.com/2017/06/feature-engineering-help-kaggle-competition-1.html
[3] James Max Kanter, Kalyan Veeramachaneni. 2015. Deep feature synthesis: Towards automating data science endeavors. IEEE International Conference on Data Science and Advanced Analytics, DSAA, Paris 19-21 Oct. 2015.

[4] Gilad Katz, Eui Chul Richard Shin, Dawn Song, 2016. ExploreKit: Automatic Feature Generation and Selection, IEEE 16th International Conference on Data Mining ICDM, 979-984.

[5] Yehuda Koren, Robert Bell. 2011.Advances in Collaborative Filtering. Recommender Systems Handbook, Francesco Ricci, Lior Rokach, Bracha Shapira, Paul B. Kantor editors, Chapter 5. Springer Science and Business Media, LLC, p. 145-186.

[6] WSDM 2018 Cup - KKBOX's Music Recommendation Challenge. 2018. https://www.kaggle.com/c/KKBOX-music-recommendation-challenge, https://wsdm-cup-2018.KKBOX.events/

[7] WSDM 2018 Cup – KKBOX's Churn Prediction Challenge. 2018. https://www.kaggle.com/c/KKBOX-churn-prediction-challenge