Truncated SVD-based Feature Engineering for Music Recommendation

KKBOX's Music Recommendation challenge at ACM WSDM Cup 2018

Nima Shahbazi
York University
nima@cse.yorku.ca

Mohamed Chahhou
Sidi Mohamed Ben Abdellah University
mchahhou@hotmail.com

Jarek Gryz
York University
jarek@cse.yorku.ca

ABSTRACT

This year's ACM WSDM Cup asked the ML community to build an improved music recommendation system using a dataset provided by KKBOX. The task was to predict the chances a user would listen to a song repetitively after the first observable listening event within a given time frame. We cast this problem as a binary classification problem and addressed it by using gradient boosted decision trees. To overcome the cold start problem, which is a notorious problem in recommender systems, we create truncated SVD-based embedding features for users, songs and artists. Using the embedding features with four different statistical based features (users, songs, artists and time), our model won the ACM challenge, ranking second. There was no music domain knowledge needed for creating the features, and we only relied on the information gain and prediction accuracy for feature selection.

CCS CONCEPTS

• **Information systems → Recommender systems**;

KEYWORDS

Truncated SVD, Cold start, Gradient boosting

1 INTRODUCTION

Until recently, people listened to their own local music library and rarely curated collections online. But the era of local music libraries is over. Personalization algorithms and unlimited streaming services like YouTube, Spotify, etc. are emerging. People now listen to all kinds of music and algorithms still struggle in some key areas. Without enough historical data, how can an algorithm predict whether a listener will like a new song or a new artist? And how can it recommend songs to brand new users?

The popularity of online content services and social media has demonstrated the value of providing relevant information to users. Recommender systems have proven to be an effective tool for this purpose and are receiving increasingly more attention. Also, recently the amount of available training data has increased enormously and advances in hardware (like GPUs) have made it possible to tackle these problems in a reasonable amount of time.

One common approach for building an accurate recommender model is collaborate filtering (CF). CF, used widely across various domains [9], exploits the overall behavior or taste of other users to suggest relevant preference to a specific user. Many web services such as Netflix, Spotify, YouTube, KKBOX[1] use CF to deliver personalized recommendation to their customers.

The ACM WSDM Cup challenged the ML community to improve KKBOX model and build a better music recommendation system using their dataset[2]. KKBOX currently uses CF based algorithm with matrix factorization but expects that other techniques could lead to better music recommendations.

In this task, we want to predict songs which a user will truly like. Intuitively, if a user much enjoys a song, s/he will repetitively listen to it. We are asked to predict the chances of a user listening to a song repetitively after the first observable listening event within a given time window. If there are recurring listening event(s) triggered within a month after the user's very first observable listening event, its target is marked 1, and 0 otherwise. More formally, let's assume an event $E(U, S, T_1)$ in which a user U listened to a song S and it occurred at time T_1. If we observe a subsequent event $E'(U, S, T_2)$ where $T2 - T1 < 1$ month (that is, we got repetitive listening within one month), event E will be marked as 1 otherwise it will be marked as 0. The submissions are evaluated on area under the ROC curve between the predicted probability and the observed target.

For this competition, KKBOX has provided a training data set that consists of information of the first observable listening event for each unique user-song pair (E) within a specific timeframe. The training and the test data are selected from users' listening history in a given time period and have around 7 and 2.5 millions unique user-song pairs respectively. Although the training and the test sets are split based on time and ordered chronologically, the `timestamps` for train and test are not provided. It is worth mentioning that this structure also suffers from the cold start problem: 14.5% of the users and 26.6% of the songs in test do not appear in the training data. Table 1 contains some statistics on users and songs in the training and test data provided by KKBOX which clearly shows the existence of the cold start problem. Hence the main task is to predict whether or not a new listener will like a new song or a new artist. We used embedded features to overcome this problem (described in detail in Section 2.1).

[1]KKBOX is an Asia's leading music streaming service, holding the world's most comprehensive Asia-Pop music library with over 30 million tracks.
[2]https://www.kaggle.com/c/kkbox-music-recommendation-challenge/data

Table 1: users and songs distribution in train and test set

Train	Test
7.377.418 events	2.556.790 events
30.755 unique users	25.131 unique users
359.966 unique songs	224.753 unique songs
9.272 users are in train but not in test	3.648 users are in test but not in train; this corresponds to 14.51% new users
195.086 songs are in train but not in test	59.873 songs are in test but not in train; this corresponds to 26.64% new songs

2 OUR APPROACH

The performance of any supervised learning model relies on two principal factors: predictive features and effective learning algorithm. The feature engineering approach exploits the domain knowledge to extract features from the training data set that should generalize well to the unseen data in the test set. The features are in general implicit information contained in the training data set which the algorithms are not able to extract by themselves. The quality and quantity of the features have a direct impact on the overall quality of the model.

Finding the relevant features requires understanding and subsequent analysis of the problem structure. Judging the quality of a new feature is done by examining the information gain of the feature first and then comparing the performance of the model before and after the feature is added. Unfortunately, finding such good features is not an easy task and is also computationally expensive. Most of the time, the engineered features lead to only small improvements in the performance of the model; to achieve a noticeable improvement requires engineering of hundreds of features.

Table 2 summarize all features provided by KKBOX. In this work, we developed a set of hand-crafted features which can be grouped into the following classes: session features, user-based statistical features, song-based statistical features, artist-based statistical features and embedding features based on user, song, artist and meta-data from the dataset. These features combined with our learning algorithm proved to be quite powerful and gave us the second-place ranking with the score difference of 0.00094 from the first place.

2.1 Feature Engineering

Before turning into feature engineering problem, we would like to highlight some important aspects of our analysis. We noticed that statistical features based on the target feature did not work well enough during the training stage. This is probably due to the dynamics involved in the data structure and the sampling strategy made by KKBOX as shown in Figure 1. Figure 1 shows the evolution of the target mean over time. The plot is set up by aggregating observations target mean in bins of size 100000. In can be seen from the figure that the target distribution is significantly decreasing over time and consequently we should expect a target mean decline in the test dataset too.

Further analysis of the given features shows a strong correlation between `source_type` feature and the target value. In Figure 2, we plot the distribution of songs count per `source_type` categories for the first 1 million observations in the train dataset. We can observe that `local_library` and `local_playlist` categories have the highest count with a large number of re-listening.

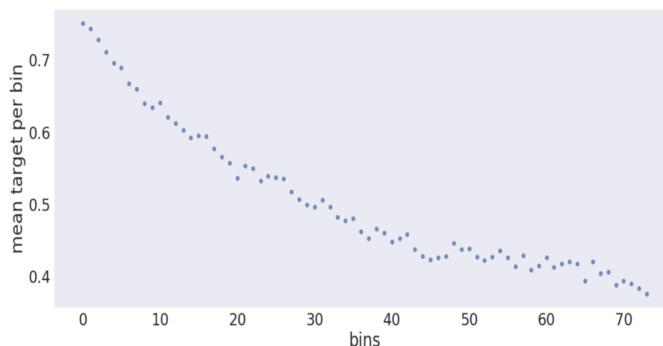

Figure 1: Evolution of repeated listening (target) in time.

In Figure 3, we plot the same distribution as above but for the last 1 million observations in the train dataset and we noticed a huge drop of re-listening in `local_library` and `local_playlist`. Also, the `online_playlist` counts have increased significantly. It seems that users behaviors have changed over time by online users who are more interested in what other users listened to via the `online_playlist`. As a result, they are more interested in discovering new songs rather than re-listening again to what they already liked. Still, we can see from the figure that listening to what other people like does not guarantee that they will re-listen to the song which explains why the target mean drops over time.

2.1.1 *Session Features.*

The provided dataset does not include any `timestamps` or user session information. However, since the data is ordered chronologically, we were able to approximate user sessions using the following approach:

1. merge the train and test dataset
2. `groupby` users
3. for each user shift the data by one to the right
4. take the difference between the original and shifted indices and call it `diff_ind`
5. small `diff_ind` values[3] correspond to observations within the same session and high values corresponds to the start of a new session
6. consequently `diff_ind` value for the start of a new session is assigned for the entire session

For example, suppose that user X has {1,2,3,4,20,21,23,26,100,105,111} indices in the data set. First, we shift the indices to the right by one {0,1,2,3,4,20,21,23,26,100,105} and then subtract the shifted ones from the original ones to yield {1,1,1,1,16,1,2,3,74,5,6}. Here, the session

[3] A value here can be any number and therefore is a `hyperparameter` in our model

Table 2: KKBOX datasets

Train and Test	Songs Meta-data	Users Meta-data
user_id	song_length	city
song_id	genre_ids	age
source_system_tab (tab name)	artist_name	gender
source_screen_name (layout name)	composer	registration_method
source_type (entry point)	lyricist	registration_date
target (train only)	language	expiration_date
	song_name	
	ISRC: song code	

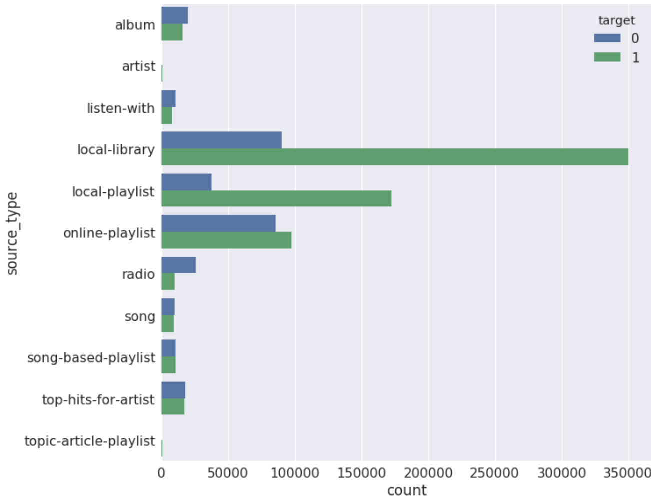

Figure 2: Number of songs per source_type for the first 1 million observations and the corresponding target values

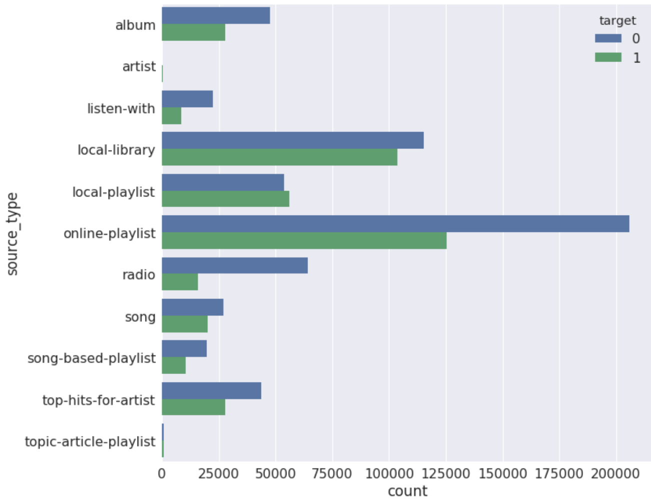

Figure 3: Number of songs per source_type for the last 1 million observations and the corresponding target values

change appeared at 16 and 74 which were chosen as the big values from our hyperparameter tuning. So, for this user we have 3 different sessions, where the session values are {1,1,1,1,16,16,16,16,74,74,74}.

The following features are created based on a session feature:

- user session value
- number of sessions for each user_id
- user session id: concatenation of user_id and session (call it user_session_id)
- transition in session: change in session marked with 1; otherwise 0
- min, max, mean, median and std of session values for each user_id, song_id and artist_name
- number of session changed for each user_id, song_id and artist_name
- number of song_id and artist_name for each session
- the first index for each user session (call it first_ind(user_id, session))
- the first index for each song's session (call it first_ind (song_id, session))
- the first index for each artist's session (call it first_ind (artist_name, session))
- max, mean, median and std of first_ind for each user, song, and artist.

2.1.2 Song and Artist Features. Figure 4 shows that some songs are very popular and have been played more frequently than the others. We also notice a large variance in the target value for a large number of songs as the number of times a song is played increases. But the most important fact is that the chances of re-listening increases with its popularity (number of times it is played). Hence, we have introduced seven features to capture the popularity of a song and its artist:

- year of the song which is derived from ISRC feature
- country of the song which is derived from ISRC feature
- total count of each song_id
- cumulative count of each song_id
- number of times a song_id appears in each source_system_tab category
- cumulative count of each artist_name
- number of times an artist_name appears in each source_system_tab category

2.1.3 User Features. Users are the main focus in our feature engineering approach. We created several different kinds of features,

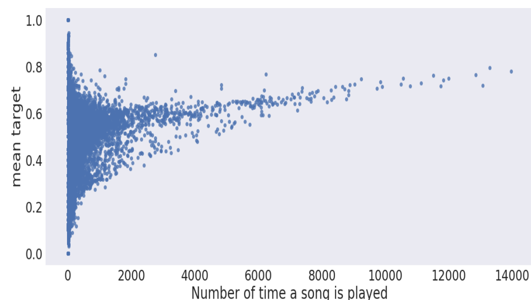

Figure 4: Distribution of played songs

most of which are statistical features based on interactions between a user and a song or an artist. These statistics are computed per session, as summarized in Table 3.

We also tried to capture the changes of user behavior over time with the following approach: for each user, we look at how the number of songs s/he listen to per session changed over time. For that, we created two linear regression models: the first model is fitted to the number of songs per user session and the second one is fitted to the number of artists per user session. Finally, the following features are extracted from the linear models: the *slope* of the model, the *first* and *last* predicted values, and the difference between the first and the last predicted values.

2.1.4 Embedding Features. The embedding features or latent factors are based on matrix factorization techniques such as "singular value decomposition" and it represents the key element in our model.

The *singular value decomposition* (SVD) of an nxd matrix A expresses the matrix as a the product of the three simple matrices:

$$A = USV^T$$

where:

1. U is an nxn orthogonal matrix.
2. V is an dxd orthogonal matrix.
3. S is an nxd diagonal matrix with nonnegative entries, and with the diagonal entries sorted from high to low (as one goes "northwest" to "southeast").

A set of latent user-based features and a set of song-based features can be derived from the user-song interaction matrix using SVD technique [5, 8].

Since our user-song matrix is huge and sparse (there are 34403 users and 419839 songs) a dimensionality reduction technique called *truncated-SVD* is used to approximate the user-song matrix and decompose the latent factors (or feature embedding) [7].

Truncated-SVD consists in building rank-k approximation A_k to the rank r matrix A by using the k most significant singular components, where $k < r$, that is:

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T, A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k S_k V_k^T, A \approx A_k$$

where σ_i is the i-th singular value of A, and u_i and v_i are the corresponding singular vectors. The low rank approximation reveals hidden links between users or songs only latent in the original data matrix. Also, from a mathematical view, the latent vectors A_k obtained from the truncated SVD are the best rank-k approximation in the sense that the Frobenius norm $||A - A_k||_F$ is minimized [2]. From a practical point of view, truncated SVD is fast and the decomposition is unique which is a nice property that allows reproducible results. To create the embedding features based on truncated SVD we proceed as follows. First, the sparse matrix A is created and then the k dimensions of the matrices U or V are extracted as embedding factors.

- feature X represents the row of the matrix.
- concatenation of $Y_1, Y_2, Y_3, \ldots$ features represent the column of the matrix.

Each entry of the matrix is equal to 1 if X and Y_1 or X and Y_2 or X and Y_3,... appear in the data.

By setting the X and Y matrices with different features, the latent factors stored in U and V matrices are extracted as follows:

1. set $X = user_id$ and $Y = song_id$ with $k = 35$, return both U and V matrices as embedding features
2. set $X = user_id$ and $Y = artist_name$ with $k = 5$, return both U and V matrices as embedding features
3. set $X = user_id$ and $Y = source_type$ with $k = 10$, return only U as embedding features
4. set $X = song_id$ and $Y = source_type$ with $k = 10$, return only U as embedding features
5. set $X = user_id$ and $Y = genre_id$ with $k = 10$, return only V as embedding features
6. set $X = user_id$ and $Y_1 = song_id, Y_2 = artist_name, Y_3 = genre_id, Y_4 = source_type$ with $k = 10$, return only U as embedding features
7. set $X = song_id$ and $Y = song_id$ with $k = 5$, return only U as embedding features. In this matrix for each user, we took the last 30 songs s/he listened to (it may be a re-listening or not). Two songs s_1 and s_2 have entry equal to 1 if a user listened to both of them in his last 30 listenings. The idea behind this matrix creation is to overcome the cold start problem related to the songs.

Two important issues must be addressed in the use of truncated SVD above: 1) how to set the value for the parameter k (the number of embedding factors to consider), and 2) whether to use matrix U or V for the embedding space of the users, songs or artists. The optimal k value, or the choice between U or V matrix, is obtained by repeated trials, taking the value which maximizes the prediction accuracy, which for this task was the area under the ROC curve.

By using the aforementioned embeddings for users we anticipated that the ones who listened to the same songs or artists will be close to each other in the new embedding space. Also, we anticipated that the songs and artists will also be close to each other in the new embedding space if they were listened to by the same users.

2.1.5 Dropped Features. The following features were dropped for various reasons before fitting the models :

- `registration_init_time`, `expiration_date`: These features represent dates and are not useful in their raw format; they were used to create the duration feature (`expiration_date - registration_date`) in days.

Table 3: Features based on users

Feature Notation	Feature description
Yr(user_id)	User's registration year
Duration(user_id)	Number of days between user's expiration date and registration date
mean((user_id,sessions),song_id)	Mean value of number of songs per user sessions
std((user_id,sessions),song_id)	Standard deviation of number of songs per user sessions
sum(user_id,song_length)	Sum of songs length for each user
cumcount(user_id)	Cumulative count of user's activity
cumcount(user_id,artist_name)	Cumulative count of user-artist interaction
cumcount(user_id,genre_id)	Cumulative count of user-genre interaction
n_unique(user_id,genre_id)	Number of unique genre categories for each user
n_unique(user_id,source_system_tab)	Number of unique system_system_tab categories for each user
n_unique(user_id,artist_name)	Number of unique artist counts for each user
n_unique(user_id,language)	Number of unique language categories for each user
n_unique((user_id,session),genre_id)	Number of unique genre categories for each user's session
n_unique((user_id,session),artist_name)	Number of unique artist names for each user's session
n_unique((user_id,artist_name),session)	Number of unique session values for each user's artist
n_unique((user_id,artist_name),song_year)	Number of unique song years for each user's artist
n_unique((user_id,artist_name),source_system_tab)	Number of unique source_system_tab categories for each user's artist
n_unique((user_id,artist_name),source_screen_name)	Number of unique source_screen_name categories for each user's artist
n_unique((user_id,artist_name),song_country)	Number of unique song countries for each user's artist
n_unique((user_id,artist_name),genre_id)	Number of unique genres for each user's artist
n_unique((user_id,artist_name),gender)	Number of unique genders for each user's artist
n_unique((user_id,session),song_id)	Number of unique songs for each user's session
n_unique((user_id,source_system_tab),song_id)	Number of unique songs for each user's source_system_tab category
n_unique((user_id,artist_name),song_id)	Number of unique songs for each user's artist
n_unique((user_id,session,artist_name),song_id)	Number of unique songs for each artist in a user's session
merge_count(user_id,session)	count number of the occurrence of merged user and session
merge_count(user_id,artist_name)	count number of the occurrence of merged user and artist
merge_count(user_id,source_type)	count number of the occurrence of merged user and source_type
merge_count(user_id,source_screen_name)	count number of the occurrence of merged user and source_screen_name
merge_count(user_id,source_system_tab)	count number of the occurrence of merged user and source_system_tab
merge_count(user_id,genre_id)	count number of the occurrence of merged user and genre
merge_count(user_id,artist_name,song_year)	count number of the occurrence of merged user,artist and song year
merge_count(user_id,artist_name,source_screen_name)	count number of the occurrence of merged user,artist and source_screen_name
merge_count(user_id,artist_name,source_type)	count number of the occurrence of merged user,artist and source_type
merge_count(user_id,artist_name,source_system_tab)	count number of the occurrence of merged user,artist and source_system_tab
merge_count(user_id,artist_name,composer)	count number of the occurrence of merged user,artist and composer
merge_count(user_id,artist_name,language)	count number of the occurrence of merged user,artist and language
merge_count(user_id,artist_name,song_country)	count number of the occurrence of merged user,artist and song country
merge_count(user_id,artist_name,session,source_type)	count number of the occurrence of merged user,artist,session and source_type
artist_first_time_seen	difference between sessions values of the first and current time a user listened to an artist
artist_last_time_seen	difference between sessions values of the last and current time a user listened to an artist

- composer, lyricist: These features contain a lot of NULL values and their usage did not show any improvement. We also tried to replace missing values in artist_name feature by the composer values but without much success.
- user_id, song_id and artist_name were the main features for all of our feature engineering approach and led to overfitting if they were used in their raw format. Instead, we used their embeddings from truncated SVD approach to tackle the cold start problem.

- song_name: we tried some features extraction and TF-IDF on the song's names but it didn't help.

2.2 Training and Validation

In order to examine the performance of the engineered feature more quickly, we used a subset of the training data (last 41% observation) to create our own training and validation sets. From this subset, the last 877417 observations were used for validation and the rest for the training set. The created validation and training sets have the same user and song distribution as it appears in the original training and test data (in order to have the same unseen users and songs distribution). Within this schema, every time that we had an improvement on our local validation set, it was guaranteed that we will get the same improvement on the original test data. Also, as we have to examine lots of features, having a small validation schema for the test set is vital. As a result, we came up with indices 4400000 to 6500000 for training and from 6500001 to 7377417 for validation.

As we are going to experiment exclusively with tree based algorithms, in our experience this kind of learners do not benefit much from one-hot encoding of categorical variable. So the only preprocessing we applied to categorical variables was mapping them to numerical ones and filling the missing values by constant numbers only.

The decision to restrict the algorithms to tree based ones was mainly because of our experience of solving similar classification recommender problems. We also tried Neural Network classifier learned embedding (with a score less than the tree-based algorithms). But due to time constrains we dropped the neural network architecture and focused more on our engineered feature with tree based algorithms (although we could have used them to reduce the variance of the predicted results).

Using the complete training set as input, we computed all of the features described above and fed these results into our classification model for the test set prediction.

2.2.1 *Model Selection and Tunning.*

For binary classification tasks, there are various learning algorithm than can be used: logistic-regression, support vector machines (SVM), neural nets, random forest, gradient boosting decision trees, etc. In this competition, we used Microsoft LightGBM implementation of gradient boosting decision trees as our classifier which was presented as NIPS'17 due to its simplicity and superior accuracy in many real world applications [6].

We have the total number of 185 features described above. A subset of those features was used to train five different models with different hyper parameters. There were 80 features in common between all models but the rest were different. The average Pearson correlation of the five-model prediction was around .89, which gives us a good boost in prediction accuracy by using blending or stacking, described in the following section.

For model parameters tuning, after several experiments, we found that the best performance was achieved for the parameters summarized in Table 4. Parameters used in Table 4 are described in the Github repository[4].

[4]https://github.com/Microsoft/LightGBM

Table 4: Model parameters

parameter	values		
learning_rate	0.1	0.1	0.1
bagging_fraction	0.9	0.8	0.8
sub_feature	0.8	0.4	0.4
min_hessian	50	500	1000
max_depth	9	63	16
num_leaves	511	200	250
num_rounds	850	80	900

Amazon EC2 c5.4xlarge-c5.9xlarge instances were used for validation-training with 16-36 CPUs and 32-72 GiB-RAM respectively. Each validation run took around 20 minutes and the time for training was around one hour only.

2.2.2 *Blending and Stacked Generalization.*

In general stacking is ensemble of models combined sequentially [10]. Blending is just averaging the output predictions of each model with different weights. While both techniques have improved the score in this competition, we used blending approach due to its simplicity and slightly better results on our validation set.

Bagging method (averaging predictions from single models with the same features and the same parameters but with different random seeds) was used for three of the models, and two of the models are just single run.

The final model was the weighted average of the five models' predictions. To find the best weights for the model blending, we used an optimization function, which is based on Nelder-Mead, quasi-Newton and conjugate-gradient algorithms [1, 3, 4].

3 RESULTS

Our model achieved 0.74693 AUC on private leaderboard with the score difference of 0.00094 from the first place. We should highlight that our models took at most 6 hours to run and had maximum number of 185 features.

Table 5 shows the improvement of the AUC score as we replace the user, song and artist with their corresponding embeddings and after adding engineered features. Note that each row in this table adds a new feature to the features introduced in the rows above. Table 6 shows the average importance gain for different set of features derived from LightGBM importance function.

Truncated SVD-based embedding features proved to be the most important and result in an increase of nearly 0.07 in the AUC score from the raw feature set. The rest of the improvement came from blending models and bagging.

4 CONCLUSION

We have presented our solution to the 2018 ACM WSDM recommender system competition. Our team, *Magic Recommenders* placed second and become one of the winners of the competition challenge. We were able to combine embedding features and statistical features to come up with a promising AUC score for this task. For future work, a promising area of research is to further explore the user and song behavior and interaction. Users typically have specific

Table 5: Result on AUC score for different set of features

Features	Validation AUC score
Raw features without user_id, song_id, artist_name	0.65510
Raw features and four additional features: Duration(user_id), song_year, song_country and Yr(user_id)	0.66450
replacing user_id, song_id with their corresponding embedding	0.68871
replacing artist_name with it's embedding	0.69122
adding the rest of embeddings describe in 2.1.4	0.72145
session, user, song and artist engineered features	0.74257

Table 6: Average importance gain for different set of features in LightGBM

Features	Average importance gain
user_id and song_id embedding features	0.17778
artist_name embedding features	0.11949
rest of embedding describe in 2.1.4	0.10149
session engineered features	0.13310
user engineered features	0.19250
song engineered features	0.18310
artist engineered features	0.09254

taste when a new song by an artist is released and this taste can change over time. Another interesting area for future work is to explore the matrix factorization by the actual target value and not the appearance of users and songs or artists.

ACKNOWLEDGMENTS

The authors would like to thank KKBOX for providing the data and offering a really challenging problem. Also, many thanks to WSDM and Kaggle for hosting this exciting competition.

REFERENCES

[1] Claude J. P. Belisle. 1992. Convergence Theorems for a Class of Simulated Annealing Algorithms on âĎÌ d. *Journal of Applied Probability* (1992). https://doi.org/10.2307/3214721
[2] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. 1999. Matrices, Vector Spaces, and Information Retrieval. *SIAM Rev.* (1999). https://doi.org/10.1137/S0036144598347035
[3] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. A Limited-Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing* (1995). https://doi.org/10.1.1.15.7343
[4] R. Fletcher and C. M. Reeves. 1964. Function minimization by conjugate gradients. *Comput J.* (1964). https://doi.org/10.1093/comjnl/7.2.149
[5] Gene H Golub and Charles F Van Loan. 1996. *Matrix Computations.*
[6] Guolin Ke, Qi Meng, Taifeng Wang, Wei Chen, Weidong Ma, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems 30* (2017).
[7] Gunnar Martinsson. 2010. Randomized methods for computing the Singular Value Decomposition (SVD) of very large matrices. *Slides* (2010). https://doi.org/10.1016/S0022-5193(05)80649-6
[8] C.D. Meyer. 2000. Matrix analysis and applied linear algebra. *Matrix* (2000).
[9] Xiaoyuan Su and Taghi M. Khoshgoftaar. 2009. A Survey of Collaborative Filtering Techniques. *Advances in Artificial Intelligence* (2009). https://doi.org/10.1155/2009/421425
[10] David H. Wolpert. 1992. Stacked generalization. *Neural Networks* (1992). https://doi.org/10.1016/S0893-6080(05)80023-1