Jian Li, Weiyi Chen, Ethan Cowan
jil761@g.harvard.edu
CS181-S16

Collaborators: Kaggle team name:
L.C.C.

Practical 3: Preferences in Streaming Music

1 Introduction

This report illustrate how we utilize matrix factorization in machine learning to predict how many plays users will listen to of a particular artist on a music streaming service.

We treat this practical as a recommender system. Recommendations can be generated by a wide range of algorithms. While user-based or artist-based collaborative filtering methods are simple and intuitive, matrix factorization techniques are usually more effective because they allow us to discover the latent features underlying the interactions between users and artists.

2 Methods and Motivations

In this section, we will go through the theoretical methods of matrix factorization. We will proceed with the assumption that we are dealing with user-artist number of plays in a recommendation system.

2.1 Motivations

In our recommendation system, there is a group of users and a set of artists. Given that each users have number of plays on artists in the system, we would like to predict how many times the users would play the artists that not given, such that we can make recommendations to the users. In this case, all the information we have about the existing values can be represented in a matrix. Assume now we have 5 users and 4 artists, and number of plays are integers, the matrix may look something like this (a hyphen means data not given):

	A1	A2	A3	A4
U1	5	3	-	1
U2	4	-	-	1
U3	1	1	-	5
U4	-	1	5	4

Hence, the task of predictions can be considered as filling in the blanks (the hyphens in the matrix) such that the values would be consistent with the existing values in the matrix.

The intuition behind using matrix factorization to solve this problem is that there should be some latent features that determine how many times a user listens to an artist. For example, two users would listens to a certain artist very frequently if they both like the artist, or if the artist is an action artist, which is a genre preferred by both users. Hence, if we can discover these latent features, we should be able to predict a value with respect to a certain user and a certain artist, because the features associated with the user should match with the features associated with the artist.

2.2 Mathematics

Having discussed the motivations behind matrix factorization, we can now go on to work on the mathematics. Firstly, we have a set U of users, and a set D of artists. Let $\mathbf{R}$ of size $|U| \times |D|$ be the matrix that contains all the values that the users have assigned to the artists. Also, we assume that we would like to

discover K latent features. Our task, then, is to find two matrics matrices $\mathbf{P}$ (a $|U| \times K$ matrix) and $\mathbf{Q}$ (a $|D| \times K$ matrix) such that their product approximates $\mathbf{R}$:

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$$

In this way, each row of $\mathbf{P}$ would represent the strength of the associations between a user and the features. Similarly, each row of $\mathbf{Q}$ would represent the strength of the associations between an artist and the features. To get the prediction of number of plays of an artist d_j by u_i, we can calculate the dot product of the two vectors corresponding to u_i and d_j:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$$

Now, we have to find a way to obtain $\mathbf{P}$ and $\mathbf{Q}$. One way to approach this problem is gradient descent, first initialize the two matrices with some values, calculate how different their product is to $\mathbf{M}$, and then try to minimize this difference iteratively.

The difference here, usually called the error between the estimated value and the real value, can be calculated by the following equation for each user-artist pair:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

Here we consider the squared error because the estimated value can be either higher or lower than the real value.

To minimize the error, we have to know in which direction we have to modify the values of p_{ik} and q_{kj}. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables separately:

$$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(q_{kj}) = -2e_{ij} q_{kj} \frac{\partial}{\partial q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij} p_{ik}$$

Having obtained the gradient, we can now formulate the update rules for both p_{ik} and q_{kj}:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{kj} \quad q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij} p_{ik}$$

Here, α is a constant whose value determines the rate of approaching the minimum. Usually we will choose a small value for α, say 0.0002. This is because if we make too large a step towards the minimum we may run into the risk of missing the minimum and end up oscillating around the minimum.

Using the above update rules, we can then iteratively perform the operation until the error converges to its minimum. We can check the overall error as calculated using the following equation and determine when we should stop the process.

$$E = \sum_{(u_i, d_j, r_{ij}) \in T} e_{ij} = \sum_{(u_i, d_j, r_{ij}) \in T} (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

2.3 Implementation

We have discussed the intuitive meaning of the technique of matrix factorization and its use in collaborative filtering. In fact, there are many different extensions to the above technique.

2.3.1 scikit-learn Decomposition

An important extension is the requirement that all the elements of the factor matrices (**P** and **Q** in the above example) should be non-negative. In this case it is called non-negative matrix factorization (NMF), and this is implemented in scikit-learn module. One advantage of NMF is that it results in intuitive meanings of the resultant matrices. Since no elements are negative, the process of multiplying the resultant matrices to get back the original matrix would not involve subtraction, and can be considered as a process of generating the original data by linear combinations of the latent features.

2.3.2 crab - scikits.recommender

Crab, as known as scikits.recommender, is a Python framework for building recommender engines integrated with the world of scientific Python packages (numpy, scipy, matplotlib). The engine aims to provide a rich set of components from which you can construct a customized recommender system from a set of algorithms and be usable in various contexts.

2.3.3 python-recsys

Python-recsys is another module for implementing a recommender system, which is build on top of Divisi2, with csc-pysparse. Compared to crab, it is able to utilize the sparsity of matrix and SVD to do fast calculation on huge data.

3 Features or Data

We were provided some basic self-reported demographic information about many, but not all, of the users, such as sex, age, and location. We are also provided the name of the artist and their MusicBrainz1 ID. There are about 233,000 users and 2,000 artists. The training set has over 4.1M user/artist pairs and the test set is of a similar size. As we are going to use recommender system to identify similarities of user, we will mainly use the training set as our data, and user/artist information will be used to identify bias.

3.1 Prepossessing

Prior to matrix decomposition, we utilize methods Imputer in sklearn.prepossessing to see which one most approach to real data. Except for global median and user median given, we have also tried user mean, user most frequent (most frequent known value to fill unknown entries), artist mean, artist median, artist most frequent. Below is the table of errors -

Filling strategy	Kaggle error
global median	201.78204
per-user median	137.79146
per-user mean	162.29745
per-user most frequent	158.38144
per-artist median	198.92070
per-artist mean	-
per-artist most frequent	-

The last two has been tested via cross-validation with high errors, so did not submit to Kaggle in order not to waste submission opportunities. In conclude, the given per-user median has the lowest error, and it could be a start place for further steps.

3.2 Normalization

The number of plays in train.csv varies from 1 to 5000, while usually recommender system utilizes rating data range from 0 to 5. So we will need to normalize our data into scaled values, we have two ways to do this according to the constraints

3.2.1 Per-user median normalizer

Usual normalizer samples individually to unit norm. Each sample (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other samples so that its norm equals one. This transformer is able to work both with dense numpy arrays and scipy.sparse matrix. Scaling inputs to unit norms is a common operation for text classification or clustering for instance.

Our normalizer is similar but uses a different formula to normalize -

$$\text{normalized val} = (\text{org val} - \text{user median})/\text{user median}$$

This guarantees the preprocessing filled entry with user median value is normalized to 0, therefore the matrix will still be highly dense for faster computing. The module we use for normalization is Normalizer.

3.2.2 Minmax scaler

This transforms features by scaling each feature to a given range. This estimator scales and translates each feature individually such that it is in the given range on the training set, i.e. between zero and one. The transformation is given by:

$$X_{std} = (X - X.min(axis = 0))/(X.max(axis = 0) - X.min(axis = 0))$$

$$X_{scaled} = X_{std} * (max - min) + min$$

This transformation is often used as an alternative to zero mean, unit variance scaling, and it is essential for NMF method since this requires the input matrix to be non-negative.

4 Approaches

Crab or python-recsys contains several recommender algorithms implemented, in fact, starting with conventional user-based and artist-based recommenders. There is also a experimental implementation based on the singular value decomposition (SVD). In this section, we will go through our back-and-forth approach to test our data and improve our score.

4.1 Build a recommender system

In crab, the MatrixPreferenceDataModel implementation stores and provides access to all the user and item data as also the associated preferences needed in the computation. The UserSimilarity defines the notion of how similar two users are; this is based on one of many possible metrics or pairwise distance calculations. A NearestNeighborhood implementation defines the notion of a group of users that are most similar to a given user. At least, the UserBasedRecommender implementation pulls all these components together in order to recommend items to users, and related features. To help visualize the relationship between the components, check the figure below.

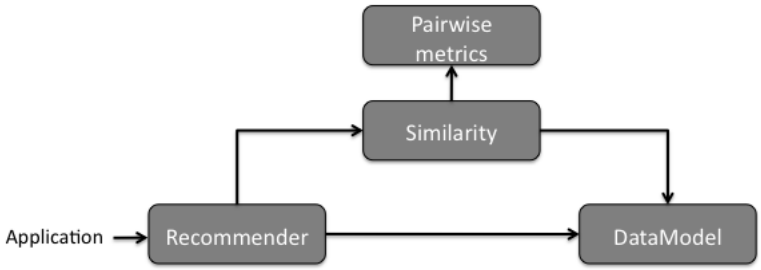

Figure 1: Crab recommender system

4.2 Evaluate recommender

A recommender engine is a tool to produce the best recommendations for the number of plays in a user-artist pair. We know that a best possible recommender would be an engine that could predict exactly how much we would like every possible item that we have not expressed any preference for and rank them based on these future preferences. In fact, most recommender engines operate by trying to do this, estimating values for some or all other artists. To evaluate the quality of those recommendations we will need tools and metrics that focus on how closely the estimated preferences match the actual preferences.

4.3 Training and Testing Data

Actual preferences do not exist because we are not given a full matrix. This can be simulated in a recommender engine by setting aside a small part of the real data set as test data. These test preferences are not present in the training data (which is all data except the test data.) The recommender is asked to estimate preference for the missing test data, and estimates are compared to the actual values.

5 Experiments on Matrix Factorization

To summarize, the idea of matrix factorization is to project user data and product data to a factor space then we can use inner product in this space to determine the rating a user will give to a product. The main challenge of this approach is that we have a sparsely sampled matrix for all user and product pairs. In this project we have 233k users, 2k artists, but only 4.1m samples, which is only 0.89% of 466 milion entries in the full 233286 by 2000 matrix. Dealing with the missing data is the main challenge.

We tested the following algorithms: SVD with imputation (replacing missing data with estimates), adding biases in the feature space and observation space, and probabilistic matrix factorization.

5.1 SVD with Imputation

Singular value decomposition sits at the heart of many matrix factorization algorithms such as PCA. Here we apply a similar approach to capture most of the information in the feature space by choosing the largest singular values. In order to address the large amount of missing data, we use user median to fill the gaps. The idea of this imputation is that SVD should at least give results close to the user median. Note one difference of this problem with scoring problems such as the netflix rating challenge is that here the number of plays is unbounded and could depend on how often each user plays music, so we use normalize all plays by median. A second consideration is that to store such large sparse matrix efficiently, we prefer to leave the missing values to zero so we can use a sparse representation of the matrix. Therefore, we subtract user mean from each play then divide it by user median to get a sparse matrix. Note all our matrix factorization methods are applied after this normalization.

After that we run SVD with different numbers of features K then pick K based on two criteria: first a large amount of energy is captured in the K singular values. To do that we run a full SVD, then plot of the charge of energy conserved as K increases in Fig. 2. Secondly, the reconstructed matrix should give improvement of out-of-sample MAE based on cross validation. Throughout this section we use 80% of samples for training the model, and the other 20% for cross validation. The results of different values of K is in Table 1.

We choose $K = 400$ because it gives the best out-sample MAE and it captures a good percentage of the energy. We submit the full training results to Kaggle and get a score of 137.72 on public scoreboard, with a marginal improvement of 0.07 point over user median.

The problem with SVD with imputation is that since we only have less than 1% of matrix elements available, the results will be overwhelmingly driven by the filled missing values. In other word, signal to noise ratio is very low. The true samples have limited influence over where the SVD results. We try to address issues by using two different approaches discussed in the next two sub sections. One is to including the user profile information. The other is to use probabilistic matrix factorization and optimize only over samples available.

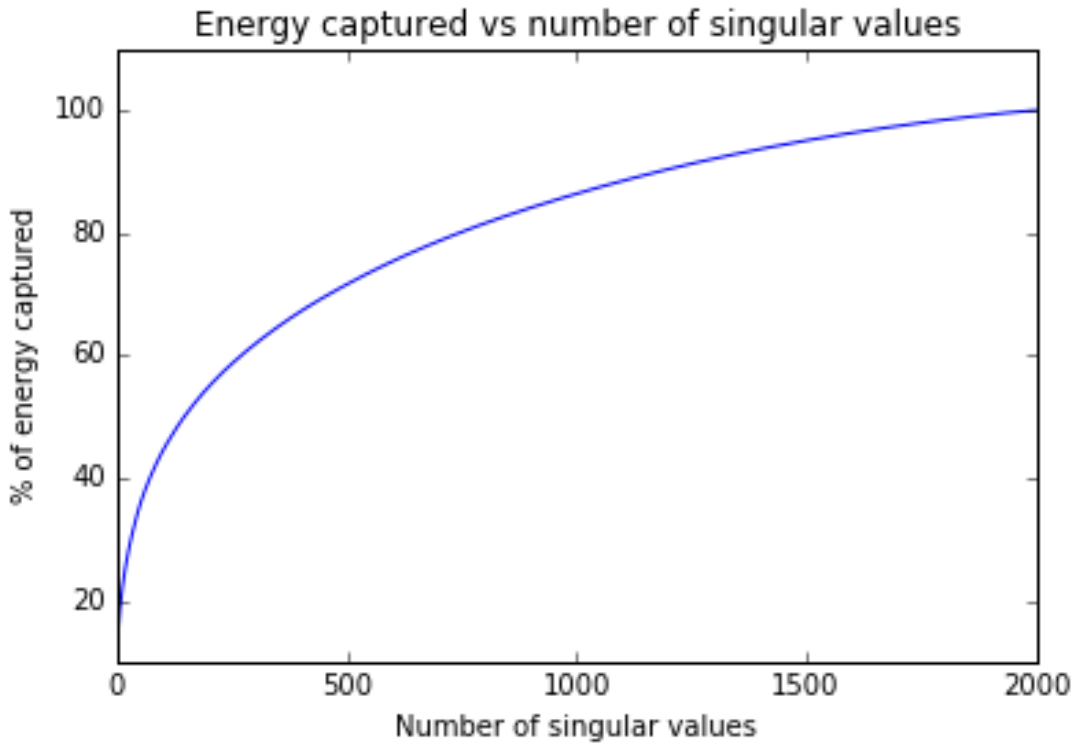

Figure 2: Energy conserved for different number of singular values.

5.2 Use of user profiles

The idea is that people with similar profiles will share common taste for music so we can add some biases to our estimate. A close observation of the age data reveals that there is some noise with either small age or large age (> 100), which is probably due to user not willing to reveal their real age. So after filtering we can see the age distribution in Fig. 3. We then divide the user into 5 different age groups. Alternatively we can set up some similarity matrix of age. For country data, we only use 43 countries out of total 239

K	In-sample MAE	Out-sample MAE	Out-sample benchmark	Energy Captured (%)
200	77.73	137.92	137.93	55
400	53.49	137.87	137.93	67.3
750	30.90	137.89	137.93	80.2

Table 1: SVD with imputation. Changes of results for different K.

countries that have more than 500 users each. We test two ways to add this bias. The first one is to add

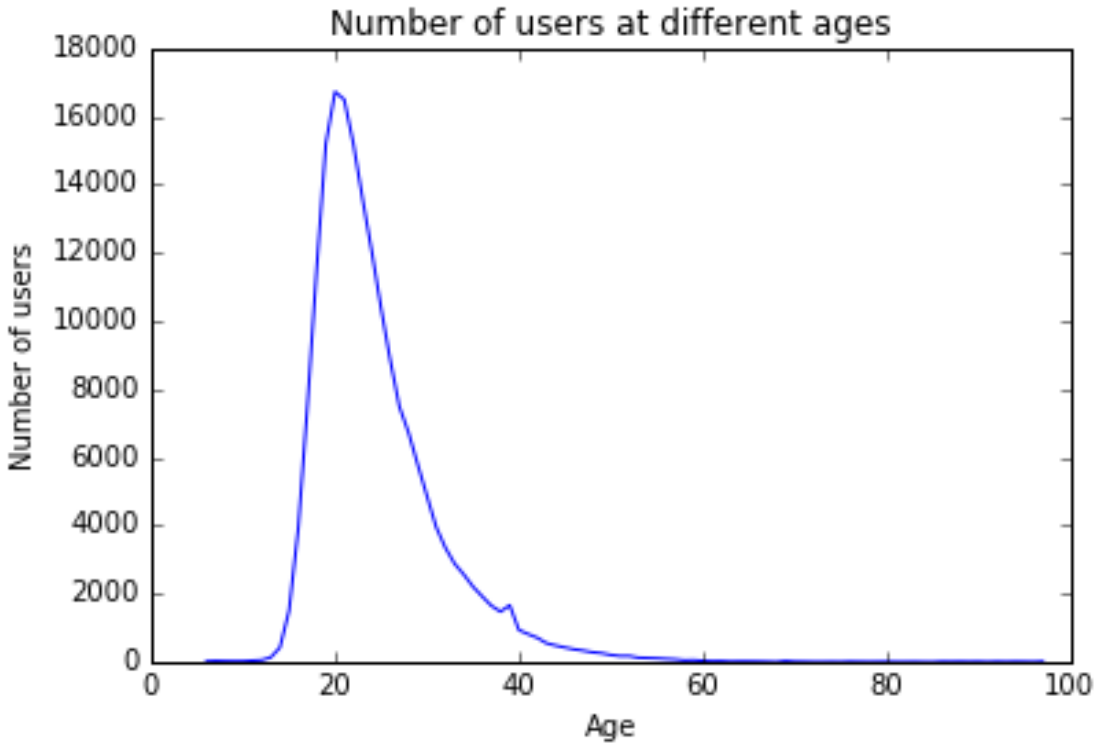

Figure 3: User age distribution.

in feature space. So it's applied on top of SVD results from the previous section. We pick out groups that belong to each subset of gender, country, or age group, then take the mean of user feature vector in each group as bias. We then add these bias to each user feature vector if the person belongs to a given group. Then we can reconstruct the full matrix and estimate the number of plays for each pair. This method has a cross validation score of 137.977 vs 137.937 using user median. So it's under performing methods with no bias.

The second way is to add the bias to the observation directly. So in the normalized space (after filling in missing data), we take the mean of each subgroup over the plays of all 2000 artists. So this essentially use the full 2000 vector as feature vector and calculate the bias based on that. Then we add this bias to user median as our estimate. We get 138.04 in out-sample MAE, worse than benchmark and the first method. So it doesn't work to model the bias explicitly.

5.3 Probabilistic Matrix Factorization

We implemented the probabilistic matrix method explained in the Mathematics section to better address the missing value problem. We use the results of SVD with imputation as the initial point of our optimization. In this formulation, the loss function only depends on the available sample points. We use stochastic gradient descent method. Basically as we run through four million training samples, we evaluate the gradient that only relates to each sample point and continue to make small gradient descent steps. If we have a fixed learning rate, we may run into problem with outliers, given that the update step depends on the L1 norm of e_{ij}. So we plot the distribution of error terms in Fig. 4. So we find 3.2% of the errors are below 2.0, and we use 2.0 as a threshold to filter out outliers which can destroy the small step gradient descent.

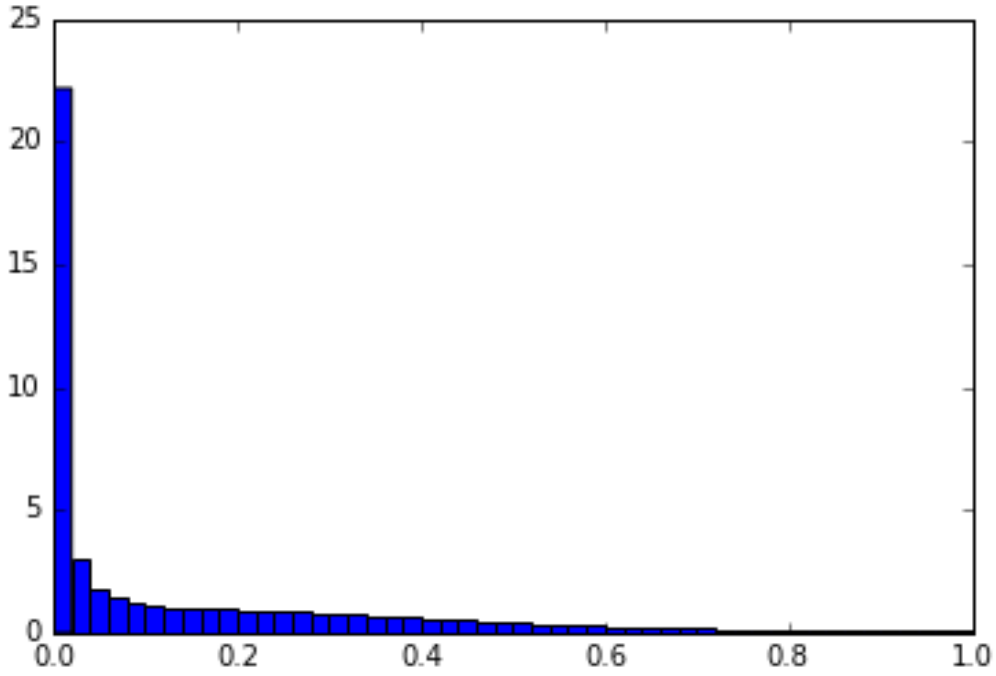

Figure 4: Histogram of error distribution in (%).

We first run some small scale test and cross validation to determine learning rate (0.00001) and regularization parameter (0.02). Then the optimization is run over the whole training set. The cost changes through PMF optimization is illustrated in Fig. 5. The overall trend is going down, although some noisy gradient may cause it to go up in certain iterations. Due to time constraint we only run it for one epoch, meaning a full sweep of the sample data. Additional epochs may further reduce the cost function. The public score, however, is 137.809, slightly worse than user median of 137.791. Adding bias to the cost function, or more robust optimization may further improve the results.

6 Results and Discussion

Below is the results of different approaches or utilizing different modules. Our Kaggle team is L.C.C. and our best Kaggle public score is 137.72 from SVD with 400 features.

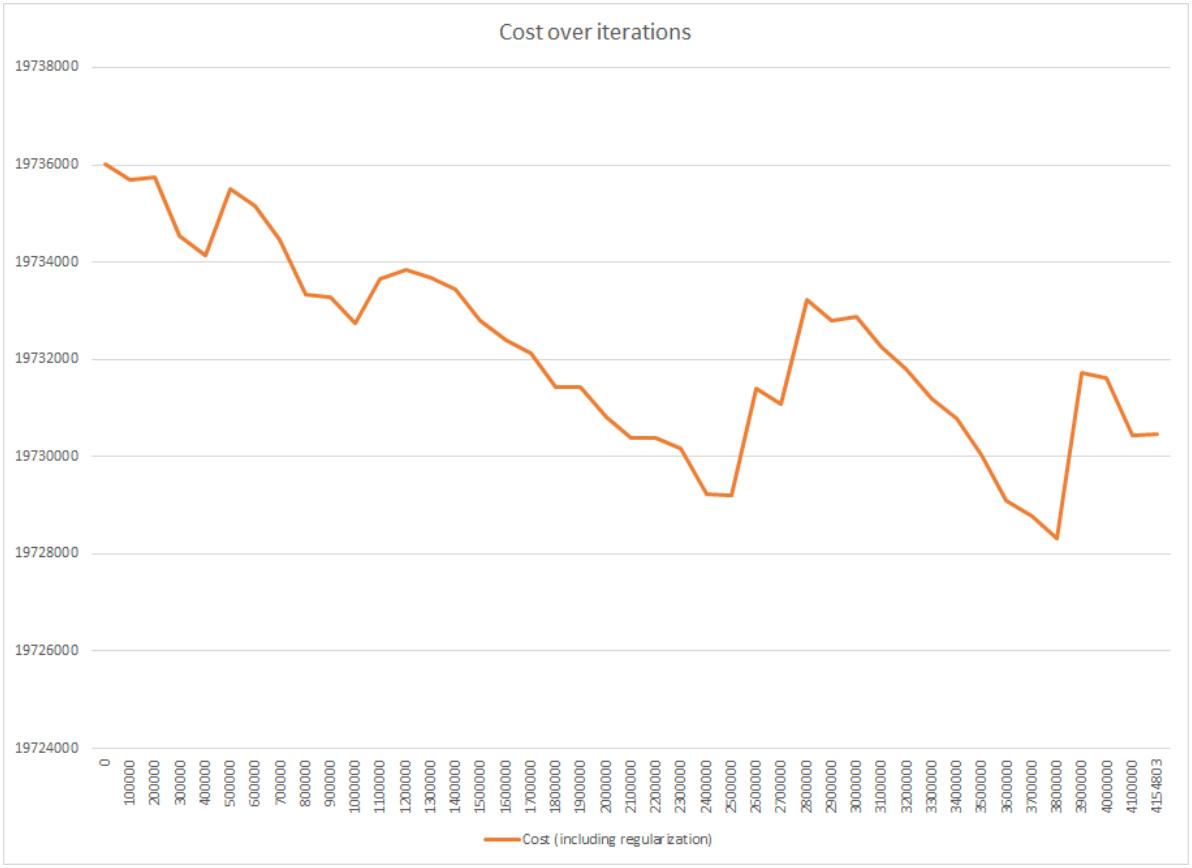

Figure 5: PMF cost change over iterations.

Approach	Kaggle Score
sklearn.PCA	failed
sklearn.incrementalPCA	137.79146
sklearn.NMF	148.60532
crab	failed
recsys, 1000 features	175.12698
recsys, 100 features	181.08562
svd 400	137.72020
pmf 400	137.80913

Sklearn.PCA failed to generate the results because its time complexity is $O(n^3)$ so it is unable to finish in a short time, we change it to sklearn.incrementalPCA approach which utilizes the density property of the matrix and decomposite the matrix very quickly, but the result does not beat the benchmark.

Crab failed from different reasons, the matrix density is less 1%, which is too sparse for crab module, so it is unable to find similarities between users and hard to generate predictions for most users.

Recsys works at a high-performance speed, however still not beating the benchmark, a possible reason is that the module was designed for ratings ranged from 0.0 to 5.0, our value as of number of plays varies too much, and it is hard to normalize to 0.0 to 5.0 with the same distribution rule. In addition, adding the number of features would improve the score using the recommender system.