Recommender system using Alternating Least Squares

Mialimanana Andrianina Bien-Aimé Razafindrakoto [1]

Abstract

Recommender systems are a a family of algorithm in Machine Learning that uses a system of ratings to predict the preference of the user regarding a set of items. It has been used in contest in the beginning of the 2010s by big corporates like Netflix, Yahoo, Xbox etc. for enhancing the user experience and bringing to main stream the best algorithm for such uses in that time. In this short report, we are going to discuss about one of those algorithms that are used in recommender systems, namely the *Alternating Least Squares* algorithm. We will discuss on the core principle of the algorithm, the dataset used for demonstrating it and will implement some additional features that will further improve the quality of the prediction which is the main goal of the recommender system family.

1. Introduction

In recent years, Machine learning has been used and improved in many fields in order to advance its powerful insight in solving numerous problems surrounding data. For our purpose, a Machine Learning algorithm will be used to make predictions in the audio-visual realm : suggesting the next content to the users to maximize the user experience. During the first steps of this study, it was not much obvious how to make predictions from such datas such as ratings, interactions with items, hours playing some genres of games, etc. All comes down to one word : *embeddings*. It allows us to map the items into a part of the space, which in our case would be the space of $n-$tuples of real numbers $\mathbb{R}^K$, where K is a positive integer, the dimension of the embeddings, that is our data will be written as follows : $(x_1, x_2, \ldots, x_K) \in \mathbb{R}^K$. This embeddings will allow us to do calculus on our data to train it with our endgoal algorithm. During such a process, or more precisely before such a process, we have to make our way through the exploration of the data. We need to choose some approporiate data structure in order for us to operate on the (really big) data. A really appropriate data structure would already help us in the making of the embeddings talked about earlier and we will mention the data structures used during the experiment later on.

Before describing our algorithm, our main goal is to write and implement an algortihm that would train on such data structure and to provide us the necessary embeddings to make future predictions for an oncoming user. In that case, our overall plan for this report is then divided in three parts : the datasets that is used for the training along with some data exploration, a description of the Alternating Least Squares Algorithms, and a display of some of the results during the training process. In the Appendix, we will describe the derivation of some of the formulas needed in the updating of our parameters.

2. The datasets

For our study, we would use the MovieLens dataset, which is made up of 32M ratings made by 200948 users on 87585 movies. Especially we will be using the *ratings.csv* file in the folder, which is the listings of all the ratings made by all the involved users during the time the data were collected. It is composed of 3 columns : userId, movieId, and ratings, which can be viewed as a list of a tuple of two integers, representing the user and the item, and another composed by a $5-$star ranking system to quantify the preference of the user toward a certain movie that he watched. We will also be using the small version of this dataset, which is composed of 100K ratings, which will be handy for the early test of our algorithm. It is also noted that we will use the file named *movie.csv* which is also a list of tuples composed of 3 columns : movieId, title, genres , which will be useful for making the prediction and the interpretation of the results made by our model, that is, if a certain movie is of a certain genre, are the other suggestions of the same kind too. For a starter, in Figure 1, we have the rankings of all the movies ratings and user ratings in one single plot. It represents how frequent some movies are being ranked than others,

[1]African Institute for Mathematical Sciences (AIMS) South Africa, 6 Melrose Road, Muizenberg 7975, Cape Town, South Africa. Correspondence to: Mialimanana Andrianina Bien-Aimé Razafindrakoto <andrianinaba@aims.ac.za>.

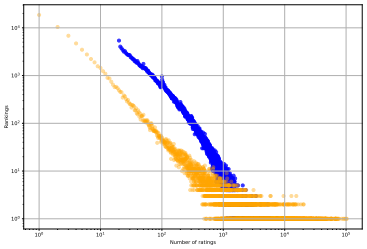

Figure 1. Power law of the distribution

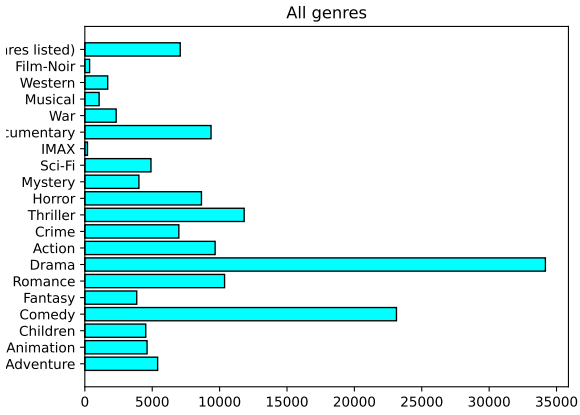

Figure 2. Genre distribution, ratings.csv

and how frequent some user have ranked some movies than others. Also, we can see in Figure 2, the distribution of all the genres of the movies that were being ranked during the collect of data. We will use these features later in the report. This latter would explain why, when doing the prediction, some genres like *Drama* and *Comedy* are recurrent in the prediction's genre. But even with that, we will later see how to use that in our favor.

2.1. The power law

Regarding the figures showing the ranking of all the rating, it is noted that we used the *log-log* scale in order the capture the "big values" attributed to some elements of the datasets. It means that some elements are more recurrent than others. In our case we could discuss that, in the movies for example, some movies (not so many) have way more ratings than some in the average. On the normal scale, the distribution would have a long tail. It is were those few movies would dwell and the rest of the movies would have way fewer rankings but will be on higher numbers. A power law is easily recognizable : it presents as a stright line or approaches on in the *log-log* scale. Our data follows mostly a power law by that latter observation. It is more consise to describe it in a mathematical way as follows :

$$p \sim \frac{1}{t^d}$$
$$\log p \sim -d \log t$$

The first line mostly says that the distribution of the data is an hyperbolic shape and the second line describes the straight line property of the distribution in the *log-log* scale as shown in the figure.

Especially, in the figure, we can interpret as follows : the ranking of a specific movie is inversely proportional to how many ratings it got. Thus, in the figure, those who are at the top barely got any ratings, the least popular of them all and those in the basis of the chart are the movies that got most of the rankings

2.2. The data structures

With the datasets in hand, we need a way to make it readable for our algorithm, in order for us to do fruitful analysis on such a massive dataset. A naive approach, and the first idea to come up was to make everything into a list of tuples of all the items and the movie separately. Such an approach would hinder very much on the comprehension and the training of data since we would have a cluster of random users and movies along with some ratings associated to them.

Such a poor approach aside, we would need two data structures for this problem. The first one is described as follows : take a new *user* in the line, if it is the first time that we see him, add him into our list of all users, give him a label and add the movie that he rated with the rating, and if it is the first time that we see the *movie* then add it to our list of movies along with the user who rated it and the rating attributed to it. In that case we end up with a *list of lists of movies* with the internal being all the movie and ratings by a specific user and a *list of lists of users*, the internal lists being all the users who rated a certain movie. This way, we can keep track of all the users and movies respectively in order of appearance and all the necessary correpondence.

Our second data structure rely mostly on the first one. It is mostly a flattened version of the *list of lists* type of data structure that we used, with another array to keep track of the index of the entity (the movie only for this one). This one will be used near the end of the report in order to make more precise predictions based on the genres of the items that the user has interacted with, that is we will use the features in our hand in order to improve the capacity of our model to make prediction.

A proper way of using these data structures will be shown during the update rules of the vectors embeddings later on.

3. Alternating Least Squares

We are now entering the main section of this report. In order to do Machine Learning on the datasets that we talked about in the previous section, we have to build a proper algorithm to train our data on.

Alternating Least Squares (ALS) is a matrix factorization algorithm used to build recommendation systems by factorizing a user-item interaction matrix into two smaller matrices representing users and items. It works by iteratively optimizing one of the matrices while keeping the other fixed, alternating between minimizing the error for user factors and item factors until the predicted ratings are as close as possible to the original ratings.

Before entering the description of the algorithm, we have to fix some conventions of writing along with a quick description of our data. An entity, a user or a movie, is described as a vector living the space $\mathbb{R}^K$. We denote by N the total number of movies, and by M the total number of users. In that case, a movie embedding would be a vector $v_n = (a_1, a_2, \dots, a_K)^T \in \mathbb{R}^K$ and for the user, it would be $u_m = (b_1, b_2, \dots, b_K)^T \in \mathbb{R}^K$.

We denote by $\mathbf{U} = \begin{bmatrix} — & u_1^T & — \\ — & u_2^T & — \\ & \dots & \\ — & u_M^T & — \end{bmatrix}$ the list of all the users made into a single matrix of shape $M \times K$ and the same for the movies : $\mathbf{V} = \begin{bmatrix} — & v_1^T & — \\ — & v_2^T & — \\ & \dots & \\ — & v_N^T & — \end{bmatrix}$ which is of shape $N \times K$. To each user and movie, we will assign respectively a bias $b_i^{(u)} \in \mathbb{R}, i \in \{i, \dots, M\}$ and $b_j^{(i)} \in \mathbb{R}, j \in \{i, \dots, N\}$. It will serve as the intercept of our model.

3.1. General description

A brief description of the **ALS algorithm** is that following some update rule that we will derive from the loss function, we will update iteratively, first the bias and then the embeddings. In other words, it is just like the following pseudo-code :

Algorithm 1 Alternating Least Squares

```
Input:user_data U, movie_data V
repeat
  for i = 1 to M do
    update user_bias
    update user_embedding
  end for
  for j = 1 to N do
    update movie_bias
    update movie_embedding
  end for
until
```

It is rather quite simple to make sense of the algorithm : *keep updating in order, first the biases, then the embeddings until we decide to stop*. In the next sections, we will describe how to update those biases and vector embeddings. We will first talk about the loss function.

3.2. Loss function

As in any Machine Learning problem, our main purpose is to minimize or maximize a certain loss function which is used to quantify our model's capability and for the updating of the parameters. For our settings, our main goal is to minimize the model's prediction of the ratings. The prediction is defined as follows :

$$\hat{r}_{mn} = u_m^T \cdot v_n + b_m^{(u)} + b_n^{(i)}$$

Simply speaking, $u_m^T \cdot v_n = \sum_i u_i \cdot v_i \in \mathbb{R}$ is the scalar product of two vectors. It is used to describe how related two vectors are. Added with both respective biases which serves as the intercept, we got our prediction. We would then minimize, over all user, for example, the quantity :

$$\mathcal{L} = \frac{\lambda}{2} \sum_{m=1}^{M} \sum_{n \in \Omega(m)} (r_{mn} - (u_m^T \cdot v_n + b_m^{(u)} + b_n^{(i)}))^2$$

where λ is a positive real number and $\Omega(m)$ is the set of all movies that the user with index m has given a rating.

This expression represents already most of the information that we need in order to make the update rules of the algorithm. However, we need to add some constraints in the expression in order to have more regularized quantities. That is, we have to add the regularization constants to both the vector embeddings and the biases.

$$\begin{aligned} \mathcal{L} = &\frac{\lambda}{2} \sum_{m=1}^{M} \sum_{n \in \Omega(m)} (r_{mn} - (u_m^T \cdot v_n + b_m^{(u)} + b_n^{(i)}))^2 \\ &+ \frac{\tau}{2} \sum_{n=1}^{N} u_m^T \cdot u_m + \frac{\tau}{2} \sum_{n=1}^{N} v_n^T \cdot v_n \\ &+ \frac{\gamma}{2} \sum_{n=1}^{N} {b_n^{(i)}}^2 + \frac{\gamma}{2} \sum_{m=1}^{M} {b_m^{(u)}}^2 \end{aligned}$$

This way, we still minimize the original loss function but now, with the regularization part, we add some constraint in the vector embeddings in order for their norms to be in some neighborhood of that are not to "big" with respect to the norm.

From now on, we will state the update rules for the biases and the vector embeddings. In the next section.

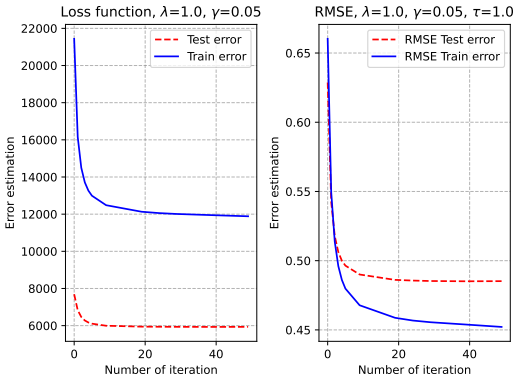

Figure 3. Bias update

3.3. Update rules

Like for any (some of them) minimization problem, we can find the update rules of the parameters, by doing a derivative in the direction of a specific element, and by solving for that element on 0. This way, the new element that we will get will go toward the minimum in respect to that element.

3.3.1. Biases

For the biases, it is quite simple since the biases in a real number. Therefore, the derivative is the derivative in the usual way. by working out the derivation, we will get the update rule for the bias :

$$b_m^{(u)} = \frac{\lambda \sum_{n \in \Omega(m)} (r_{mn} - (u_m^T \cdot v_n + b_n^{(i)}))}{\lambda |\Omega(m)| + \gamma} \tag{1}$$

In the same fashion, the update rule for the movie biases is :

$$b_n^{(i)} = \frac{\lambda \sum_{m \in \Pi(n)} (r_{mn} - (v_n^T \cdot u_m + b_m^{(u)}))}{\lambda |\Pi(n)| + \gamma}$$

Interpretation : Notice that we only "removed" the term that we want to update in the numerator, and, if not for the regularization term, it all goes down to averaging over the numerator. Hence, γ avoids such crude update by adding in the denominator. We see in Figure 3 that we already have some nice monotonically descending curves by just the bias update only.

3.3.2. Vector embedding

The update of the user vector goes in the same fashion as the bias update. It is noted that now, we are doing the derivation with respect to a vector, not a scalar, and in that case, after solving for 0 in the gradient, we would get another vector.

The update rule for the user vector is as follows :

$$u_m = \mathbf{A}^{-1} \left(\lambda \sum_{n \in \Omega(m)} v_n (r_{mn} - b_m^{(u)} - b_n^{(i)}) \right) \tag{2}$$

where

$$\mathbf{A} = \left(\lambda \sum_{n \in \Omega(m)} v_n \cdot v_n^T + \tau \mathbf{I} \right)$$

And we also get the movie embedding vector update like :

$$v_n = \mathbf{B}^{-1} \left(\lambda \sum_{m \in \Pi(n)} u_m (r_{mn} - b_n^{(i)} - b_m^{(u)}) \right)$$

with

$$\mathbf{B} = \left(\lambda \sum_{m \in \Pi(n)} u_m \cdot u_m^T + \tau \mathbf{I} \right)$$

Interpretation: The most notable part is the first term : *if not for the regularization term, that inverse would most of the time be a singular matrix, i.e. not invertible*. In fact, it makes the computation of the inverse matrix more stable than if there was no regularization term since most of the time, a user may have made few ratings, and this matrix would be singular.

3.3.3. First test

Before going any further, we have to use a metric to measure the performance of our data on unseen data (the test or validation set). The metric that we will use in all of the testing is the **Root Mean Squared Error (RMSE)**. For all the plots representing the model performance, the left plot would be the loss function and the right plot would be the RMSE on both the test set and the training set.

From now on, we can begin to work on the implementation and the discussion on the results. Below we see some plots for the 100K dataset. For example, the figure Figure 4 represents the result of a training loop on the 100K dataset, with the hyperparameters : $K = 8, \lambda = 1.0, \gamma = 0.05$. The loss function actually goes down monotonically, which is a good sign that we are heading for a minimum for all the parameters. Also, the RMSE is convincing since even on unseen dataset, the model still performs good enough because the test RMSE is still in the vicinity of the train RMSE.

3.3.4. Hyperparameter choice

In the little demo above, we fixed our hyperparamater. Of course, not every hyperparameter gives raise to same result. Some are more prone to give better prediction than others, while some some tend to overfit which is not desirable.

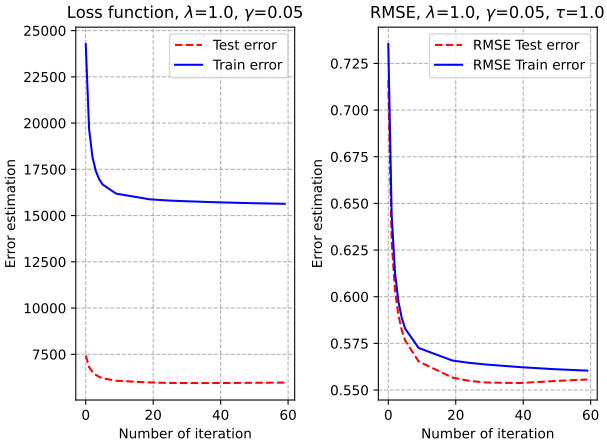

Figure 4. $K = 8, \lambda = 1.0, \gamma = 0.05$

Table 1. Bad calibrated and no features

TITLE	GENRES
DESTINATION: PLUTO BEYOND THE FLYBY	DOCUMENTARY
IN THE VALLEY OF DEATH	ADVENTURE, WESTERN
PILGRIM'S PROGRESS – JOURNEY TO HEAVEN	(NO GENRES LISTED)
WESLEY	(NO GENRES LISTED)
CRAZY BIBLE	(NO GENRES LISTED)
A DRAMATIC FILM	DOCUMENTARY
MY SIX LOVES	COMEDY
SPONGEBOB SQUAREPANTS: TIDE AND SEEK	ANIMATION, COMEDY
SPONGEBOB SQUAREPANTS: HEROES OF BIKINI BOTTOM	ANIMATION
THE SOUND OF VIOLET	COMEDY, DRAMA, ROMANCE

First of all, we can talk about the embedding dimension K. It plays a big role in whether the model will do good during the prediction process or perform badly. It is also noted that the training time scales according to the embedding dimension. For example, for $K = 5$ the training time is quite quick even wihout parallelization but for $K = 20$, it takes way longer and even hours. This is due to the inverting of **A** and **B** during the training process, since for every movie we need to do a matrix inverting and same goes for every user. In term of complexity, inverting a matrix takes about $\mathcal{O}(K^3)$ time complexity. In that case, one loop iteration of the algorithm would take on average $\mathcal{O}(N \times K^3 + M \times K^3)$, to go through all the movies and users. This is quite slow especially when training on the full datasets composed of all 32 millions ratings. Also, there are some risks of numerical instability during the inversion process. One way to avoid it is using `np.linalg.solve` which is used for solving systems of equations using matrix multiplication but can still be used for our purpose by solving for $\mathbf{x}$ the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. Also the parallelization process using packages like `Numba` for example, helps to reduce considerably the training time since almost all of the implementation in our algorithm can be vectorized in a way that the package can accelerate nicely.

Next up, the choice of λ, γ and τ which are the regularization constants, used for tuning the model and bettering the model's endgoal. One thing to remark is that if we divide τ and γ by λ, our Loss function would be of the form :

$$\begin{aligned}\mathcal{L} = &\frac{1}{2}\sum_{m=1}^{M}\sum_{n\in\Omega(m)}(r_{mn} - (u_m^T \cdot v_n + b_m^{(u)} + b_n^{(i)}))^2 \\ &+ \frac{\tau'}{2}\sum_{n=1}^{N} u_m^T \cdot u_m + \frac{\tau'}{2}\sum_{n=1}^{N} v_n^T \cdot v_n \\ &+ \frac{\gamma'}{2}\sum_{n=1}^{N} {b_n^{(i)}}^2 + \frac{\gamma'}{2}\sum_{m=1}^{M} {b_m^{(u)}}^2\end{aligned}$$

So most of the tuning would rely on the value of τ' and γ'. Hence the model's performance would depend on the trade-off of those two values. Their value would be decided either during the process of the training by using techniques like `grid search` or either during the prediction process during which we decide which set of hyperparameter would yield a good prediction.

4. Going bigger

With the actual dataset, the 100K, we get a dummy model that may work, and actually predict well on this small dataset. However, in order for our prediction of the next content to be more consistent and to make sense, we need more datasets. From now on, we will use the 32M dataset to make predictions.

Basically, we still need the same algorithm as before with the smaller dataset. But the astounding amount of information in this new dataset will slow down considerably our training and we are more prone to some numerical instability as we go higher in the dimension of the embeddings. To resolve that, we need parallelization and speed. The python package `numba` is a good choice for our actual setting.

Table 2. Star Wars and Adventure Film Franchises

TITLE	GENRES
STAR WARS: EPISODE IV	ACTION, ADVENTURE, SCI-FI
STAR WARS: EPISODE V	ACTION, ADVENTURE, SCI-FI
RAIDERS OF THE LOST ARK	ACTION, ADVENTURE
STAR WARS: EPISODE VI	ACTION, ADVENTURE, SCI-FI
STAR TREK II	ACTION, ADVENTURE, SCI-FI, THRILLER
INDIANA JONES: LAST CRUSADE	ACTION, ADVENTURE
STAR WARS: EPISODE I	ACTION, ADVENTURE, SCI-FI
STAR WARS: EPISODE II	ACTION, ADVENTURE, SCI-FI, IMAX
STAR WARS: EPISODE III	ACTION, ADVENTURE, SCI-FI
STAR WARS: EPISODE VII	ACTION, ADVENTURE, FANTASY, SCI-FI, IMAX

With the full dataset, we can already make some decision for a new user and show, for example, the first ten in the prediction. The way to do a prediction is then :

$$\hat{r}_{new,n} = u_{new}^T \cdot v_n + b_{new}^{(u)} + b_n^{(i)} \tag{3}$$

This quantity is then evaluated on all the movie datasets, on all embeddings and all the biases and we can display some elements in order to make sense of the prediction, like the 10 highest result for example.

Some results of this process is displayed in **??**. It is noted that in this table, we are doing the updates in a simpler way, no additional information apart from the movie and user embeddings. In the next section, we will discuss how to add features to make our model more interesting and incorporate new elements to the training process.

5. Cold start

Imagine you are a new user that wants to see some movie and rely on the recommender system to suggest you some more. How can the model know any more information from you if you only watched one movie through all the catalogue? One way to to solve that is by considering additional features from the data that could be used by the model to point out the space of item related to those additional features. For our case, these additional features would be the genres of the movies. We can still proceed in the same way as from the start : make embeddings for those new structures. But we need a way to integrate those informations into the movies' themselves. In that case, we can generate

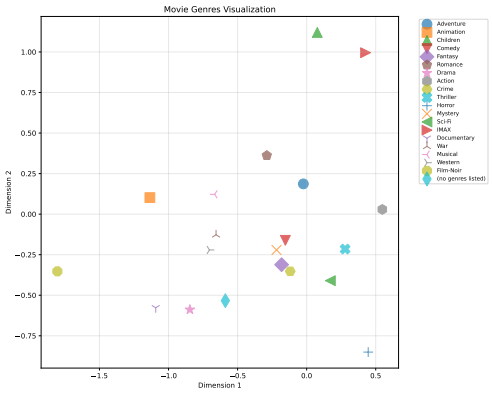

Figure 5. Features in dimension 2

the movies in such a way that these new features are part of their internal configurations i.e. the vector embeddings.

5.1. The setting

We represent the features as :

$$p(f_k) \sim \mathcal{N}(f_k; \mathbf{0}, \beta\mathbb{I})$$

and

$$p(v_n) \sim \mathcal{N}(v_n; \frac{1}{\sqrt{F_n}} \sum_{k \in F_n} f_k, \tau\mathbb{I})$$

where

$$F_n = \{f_i \text{ features of the movie indexed } n\}$$

We can say then that the movie is a linear combination of all of its genres. That way we can point out some insightful information about the movie to the model even if the user only did one request.

In order for our model to incorporate those new features we need to update our loss function to handle the changes.

$$\begin{aligned}\mathcal{L} = &\frac{1}{2}\sum_{m=1}^{M}\sum_{n\in\Omega(m)}(r_{mn} - (u_m^T \cdot v_n + b_m^{(u)} + b_n^{(i)}))^2 \\ &+ \frac{\tau'}{2}\sum_{n=1}^{N}(v_n - \frac{1}{\sqrt{F_n}}\sum_{k\in F_n} f_k)^T \cdot (v_n - \frac{1}{\sqrt{F_n}}\sum_{k\in F_n} f_k) \\ &+ \frac{\tau'}{2}\sum_{n=1}^{N} u_m^T \cdot u_m \\ &+ \frac{\gamma'}{2}\sum_{n=1}^{N} {b_n^{(i)}}^2 + \frac{\gamma'}{2}\sum_{m=1}^{M} {b_m^{(u)}}^2\end{aligned}$$

5.2. Updating the features

Form the last section, we got a new loss function to minimize. The new term that was added in the loss function

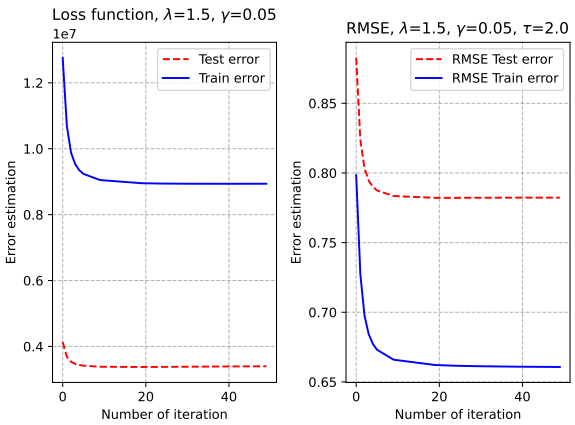

Figure 6. Update with features

Algorithm 2 Alternating Least Squares with features

Input:*user_data* **U**, *movie_data* **V**
repeat
  for $i = 1$ **to** M **do**
    update user_bias
    update user_embedding
  end for
  for $j = 1$ **to** N **do**
    update movie_bias
    update movie_embedding
  end for
  for $k = 1$ **to** $num\_features$ **do**
    update features
  end for
until

actually affects both the vector embeddings of the movies and the features, the users embeddings and all the biases are all left untouched. This way, the model can *pinpoint* the genres of a certain movie during the training and will use that information during the prediction process. The new update rules are as follows :

$$v_n = \mathbf{B}^{-1}\left(\lambda \sum_{m\in\Pi(n)} u_m(r_{mn} - b_n^{(i)} - b_m^{(u)}) + \tau\frac{1}{\sqrt{F_n}}\sum_{k\in F_n} f_k\right)$$

with

$$\mathbf{B} = \left(\lambda \sum_{m\in\Pi(n)} u_m \cdot u_m^T + \tau\mathbf{I}\right)$$

We see that the features corresponding to one specific movie are all added in the new update rule of the movie vector. As for the features, it is more interesting :

$$f_k = \frac{\sum_{n:f_k\in F_n}\frac{1}{\sqrt{F_n}}\left(v_n - \frac{1}{\sqrt{F_n}}\sum_{k\neq i} f_i\right)}{\left(\sum_{n:f_k\in F_n}\frac{1}{F_n}\right)+1}$$

Boardly speaking, in the numerator, we substract from the movie vector, all the features except the one that we are focusing on, and we averaging on all the movies that has this actual feature as genre. This way, we still get like a scaled version of the actual feature vector.

The result on all this process is displayed in Figure 6. With this new updating process, our main algorithm would be something like in 2.

6. The results

In order to quantify our result, we have to make some predictions according to the user's preference and ratings. Ideally, to show a new user some good recommendations, he has to make a high ratings on his favorite movie like 5 stars or a 100/100 ratings. That would help the model give him new recommendation because the model would evaluate *exactly* what kind of movie does the user wants to see next. We hope that the features of the movie would be captured by the model instead of just the vector embeddings. It is noted that we can already get some good results even without the features, but needed to make some more tweaking in order to make the result consistent enough as shown in the Table 1. That is the hyperparameter always play a big role in the end result even with the features.

To make a new prediction, we have to create a new profile for the user, that is we have to create an embedding for the new user along with a bias. Creating a new user is like doing the training process but for only one user and only one item. We have to iterate on the update rule of user given in equation 1 and equation 2.

After creating a new user, we go through the rating process over all the movies again. And so, in Table 3 we can see some good results after adding the features to our model when the user have watched Toy Story for example. This is a good set of recommendation, just by adding the features.

7. Summary

The main content of this report was not the full Matrix factorization on its own. Even so, we were able to do this with the help of the basics of Matrix factorization and achieved these nice results. Experiencing with the datasets like MovieLens32M has helped to show that, in an almost simple manner, we were able to achieve good prediction on a big datasets. More complex structures could be added to the model in order to make the prediction a lot more precise or to achieve a better error metric by using the tags for example. However, our model is already (almost) production ready

Table 3. Chilren Genre

TITLE	GENRES
TOY STORY	ADVENTURE, ANIMATION, CHILDREN, COMEDY, FANTASY
TOY STORY 2	ADVENTURE, ANIMATION, CHILDREN, COMEDY, FANTASY
MONSTERS, INC.	ADVENTURE, ANIMATION, CHILDREN, COMEDY, FANTASY
FINDING NEMO	ADVENTURE, ANIMATION, CHILDREN, COMEDY
INCREDIBLES	ACTION, ADVENTURE, ANIMATION, CHILDREN, COMEDY
TOY STORY 3	ADVENTURE, ANIMATION, CHILDREN, COMEDY, FANTASY, IMAX
A DRAMATIC FILM	DOCUMENTARY
SPONGEBOB SQUAREPANTS: TIDE AND SEEK	ANIMATION, COMEDY
SPONGEBOB SQUAREPANTS: HEROES OF BIKINI BOTTOM	ANIMATION
MOOMIN AND MIDSUMMER MADNESS	ANIMATION, CHILDREN

and gives enough power to improve the user experience in a (almost) real use case.

References

Paquet, U. Applied ml at scale lecture slides. Technical report, African Institute for Mathematical Sciences, South Africa, 2025.

Y. Koren, R. B. and Volinsky, C. Matrix factorization techniques for recommender systems. Technical report, Yahoo Research,, Yahoo Research, August 2009.