Department of Computer Science and Information Engineering
College of Electrical Engineering & Computer Science
National Taiwan University
master thesis

Large-scale Collaborative Filtering Algorithms

Ma Chih-Chao

Advisor: Lin Chih-Jen, Ph.D.
中文摘要

隨著電子商務的市場爆炸性地成長，給予各種不同的客戶個人化的建議逐漸成
為重要的議題。協同過濾為一項可組織與分析客戶喜好，並給予適當建議的重要技
術。在此論文中，我們針對大規模的協同過濾演算法進行研究，以在可接受的時間
內處理大量的資料。我們以著名的奇異值分解法做為演算法的基礎，並提出一些改
進的方式，亦針對後處理的方法進行討論。我們參與了Netflix Prize此一關於預測對
電影之喜好的競賽，並且得到良好的結果。

關鍵詞：協同過濾、推薦系統、奇異值分解法、矩陣分解、Netflix Prize。
ABSTRACT

As the market of electronic commerce grows explosively, it is important to provide customized suggestions for various consumers. Collaborative filtering is an important technique which models and analyzes the preferences of customers, and gives suitable advices. In this thesis, we study large-scale collaborative filtering algorithms to process huge data sets in acceptable time. We use the well-known Singular Value Decomposition as the basis of our algorithms, and propose some improvements. We also discuss post-processing methods. We participate at the competition of Netflix Prize, a contest of predicting movie preferences, and achieve good results.

KEYWORDS: collaborative filtering, recommendation system, singular value decomposition, matrix factorization, Netflix Prize.
TABLE OF CONTENTS

中文摘要 .......................................................... ii
ABSTRACT ............................................................ iii
LIST OF FIGURES ...................................................... vi
LIST OF TABLES ....................................................... vii
CHAPTER

I. Introduction ......................................................... 1

1.1 Problem Definition .............................................. 1
1.2 Evaluation Criteria ............................................. 2
1.3 Related Work .................................................... 3
1.4 Summary .......................................................... 5

II. Singular Value Decomposition ..................................... 7

2.1 Formulation ....................................................... 7
2.2 Optimization .................................................... 12
2.3 Variants ......................................................... 16
2.4 Compound SVD .................................................. 19

III. Post-Processing ................................................... 24

3.1 Residuals ......................................................... 24
3.2 $K$-Nearest Neighbors ......................................... 25
3.3 Regression ....................................................... 28

IV. Experiments ........................................................ 32

4.1 Data Sets ......................................................... 32
4.2 Experimental Results ........................................... 33

V. Conclusions ........................................................ 38
LIST OF FIGURES

Figure

4.1 Plots of RMSE versus training time .................................. 35
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Best RMSE achieved by each algorithm</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>RMSE for each post-processing algorithm</td>
<td>36</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

Collaborative filtering is a technique of predicting the preferences and interests of users from the collections of taste information given by a large amount of users. It has the basic assumption that people who agreed in the past will also agree in the future. Collaborative filtering can be used for a recommendation system that automatically suggests users his/her preferred products.

1.1 Problem Definition

Suppose that a database collects the preferences of \( n \) users for \( m \) objects as numerical scores. For example, a user can score a movie he or she has watched by a rating of 1 – 5 stars. Usually, a user does not score all objects in the database, but only those which the user has touched. Furthermore, some users may score many objects, but others only score a few.

Let \( V \in \mathbb{R}^{n \times m} \) be the matrix of the collected scores in the database, and \( I \in \{0, 1\}^{n \times m} \) is its indicator such that \( I_{ij} = 1 \) if object \( j \) is scored by user \( i \) and 0 if the score is missing. In most cases, \( V \) is sparse and unbalanced so that the numbers of scores for every user or object are unequal with a large variance. The existing scores in \( V \) work as the training data of collaborative filtering algorithms, and the goal is to predict the missing scores in the database. Let \( A \in \mathbb{R}^{n \times m} \) be a sparse matrix including
all or part of the missing votes as its nonzero entries. A collaborative filtering algorithm
aims at predicting the values in $A$. The performance of collaborative filtering can be
measured by the error between the prediction values and the ground-truth.

1.2 Evaluation Criteria

The performance of a collaborative filtering algorithm is computed using the test
data matrix $A$ mentioned above. The ground-truth scores in $A$ must be known for
computing the error measure, but the goal is to predict unknown future scores. For
example, $A$ may be constructed by the recent scores collected in a data set. The
formation and distribution of $A$ may affect the evaluation of algorithms. For example,
if the scores in $A$ are randomly sampled from all exist scores in the database, a user
who gives more scores in the database tends to have more scores in $A$. This property
may mislead the evaluation of algorithms for the applications which care about the
users equally.

A good example of test data generation is Netflix Prize (Bennett and Lanning,
2007), which is a grand contest for collaborative filtering on how people like or dislike
movies. The database used for Netflix Prize contains over 100 million scores for 480, 189
users and 17,770 movies. The test data of Netflix Prize, which are known by the
sponsor but invisible to the competitors, are selected from the newest scores of each
user. More precisely, they select a fix number of most recent scores made by each user,
regardless of the total number of scores given by that user. Then, the collection of
these newest scores is randomly divided into three sets, named probe, quiz, and test
sets. The ground-truth scores of probe set are given to the competitors as well as other
older scores in the database. The competitors are then asked to predict the scores of
quiz and test sets, but the division between quiz and test set is also unknown to them.
Netflix uses the quiz set to compute the error measure and report it on their public leaderboard, while the test set is used to decide the final winner of the prize. The submission of predictions is restricted to be at most once per day, so the competitors usually use part of the training data to test their algorithms in an offline mode. That is, a validation dataset divided from the whole training data is needed.

These sets are generated so that the validation and test data have a similar distribution. Since the test data are selected from the newest scores in the database, this selection matches the goal of predicting future scores from previous ones. The similarity between validation and test data ensures that the performance of an algorithm is consistent in validation and testing. The random division of test data, i.e., the separation of quiz and test sets, has another purpose that prevents overfitting on test data and enhances the fairness of the contest. For each user, the number of unknown scores to be predicted is roughly the same. Hence, the error of predictions for users who gave only a few scores in the past tends to be larger due to the lack of information. This situation is challenging to collaborative filtering algorithms.

Another issue of evaluation is the error measure between prediction values and the ground-truth. A common and efficient measure is Root Mean Square Error (RMSE), which is used by the Netflix Prize. Consider the prediction matrix $P \in \mathbb{R}^{n \times m}$ and the ground-truth answer matrix $A \in \mathbb{R}^{n \times m}$. Let $J \in \{0, 1\}^{n \times m}$ be the indicator of $A$. The RMSE between the prediction $P$ and the answer $A$ is defined as

$$\text{RMSE}(P, A) = \sqrt{\frac{\sum_{i=1}^{n} \sum_{j=1}^{m} J_{ij} (A_{ij} - P_{ij})^2}{\sum_{i=1}^{n} \sum_{j=1}^{m} J_{ij}}}.$$  

(1.1)

1.3 Related Work

Several conventional methods are available for collaborative filtering, and they appear in the competition of Netflix Prize. In this section, we briefly describe these
methods as well as their strengths and weaknesses. More details of some methods are shown in later chapters.

The most intuitive way to predict the score of an object given by a user is to ask other users who have scored the object. Instead of merely using the mean score of that object among all users, we would like to consider only those users similar to the target user. This method is known as $K$-Nearest Neighbor, abbreviated as KNN. For an unknown user/object pair to be predicted, a typical KNN algorithm finds at most $K$ users with the highest similarity to the given user, and takes the average on the existing scores of the given object from those $K$ users. In contrast, one can also consider the similarity between objects rather than users. This approach is even more popular in Netflix Prize, since the distribution of scores on users is more unbalanced than that on objects. The KNN algorithm can be applied directly to the scores (Paterek, 2007), or to the residuals of another algorithm as a post-processing method (Bell et al., 2007a).

The key point of KNN methods is the definition of similarity. Usually the similarity is computed by the feature values of users and objects which represent the characteristics of them in numerical values, but these features are difficult to find.

Another type of methods finds the feature values of each user and each object, and then predicts the unknown scores by a prediction function using those features. Usually such algorithms involve a matrix factorization which constructs a feature matrix respectively for users and for objects. This kind of algorithms includes Non-Negative Matrix Factorization and Singular Value Decomposition, which have numerous variants and also play important roles in the Progress Prize of Netflix Prize in the year 2007 (Bell et al., 2007b). The matrix factorization procedure is usually done by numerical optimization, and regularization is also an important consideration against the problem of overfitting. Generally speaking, this type of methods assumes that the
score values depend on a predefined prediction function. The implementation is easy, but its performance is often limited without combining with other methods. Some variants different from conventional matrix factorization are also mentioned in Bell et al. (2007b). They do not find the user features directly by optimization, but aggregate them by the object features with some transformation. The optimization is only performed on the object features, so the number of variables is smaller and the user features are dependent on the object features.

Besides the widely-used algorithms described above, there are also other methods which give good performance in the competition of Netflix Prize. One of them uses a model called Restricted Boltzmann Machines (Salakhutdinov et al., 2007). Its formulation is similar to matrix factorization, but the structure is more complicated and involves some binary-valued variables as well as real-valued ones. The Restricted Boltzmann Machines model is proved to perform well in Netflix Prize, by either itself or a linear combination with other algorithms. Another approach is Regression, which predicts the unknown scores by taking the scores in the training data as observations. The target values of observations are the values of scores, and the features used for regression are often retrieved by other algorithms due to the lack of features in the training data. A regression algorithm usually trains a model for each user or for each object, and predicts an unknown score by the corresponding model. On the other hand, if one wants to combine the results of several algorithms for better accuracy, regression is also useful to find the weights of linear combination.

1.4 Summary

In this chapter, we describe the problem of collaborative filtering, and investigate related algorithms. One of the conventional algorithms, Singular Value Decomposition,
is explained particularly in Chapter II with some useful variants. In Chapter III, we show some post-processing methods that can further improve the performances of SVD algorithms. Chapter IV contains the experiment results of our proposed algorithms with two data sets Movielens and Netflix, as well as our best result for Netflix Prize which gets the 35th rank of the contest. The conclusions are then given in Chapter V.
Singular Value Decomposition, abbreviated as SVD, is one of the factorization algorithms for collaborative filtering (Zhang et al., 2005). This type of algorithms finds the factors, or features of users and objects, and makes the predictions based on these factors. Some collaborative filtering algorithms have additional restrictions on each single feature value, or between the feature vectors of multiple users, but Singular Value Decomposition does not impose restrictions and is easier to implement.

2.1 Formulation

Suppose $V \in \mathbb{R}^{n \times m}$ is the score matrix of $m$ objects and $n$ users, and $I \in \{0, 1\}^{n \times m}$ is its indicator. Usually the data are not only sparse but also unbalanced; i.e., some users scored a large amount of objects but others scored less. The SVD algorithm finds two matrices $U \in \mathbb{R}^{f \times n}$ and $M \in \mathbb{R}^{f \times m}$ as the feature matrix of users and objects. Here $f$ is the dimension of SVD. A prediction function $p$ is used to predict the values in $V$. The value of a score $V_{ij}$ is estimated by $p(U_i, M_j)$, where $U_i$ and $M_j$ represent the feature vector of user $i$ and object $j$, respectively. Once $U$ and $M$ are found, the missing scores in $V$ can be predicted by the prediction function.

The optimization of $U$ and $M$ is performed by minimizing the sum of squared errors between the existing scores and their prediction values. This leads to the following
objective function with regularization terms:

\[ E = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^{n} \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^{m} \|M_j\|^2, \]  

(2.1)

where \( k_u \) and \( k_m \) are the regularization coefficients and we use the Frobenius norm of \( U \) and \( M \) as the penalty. The regularization term can effectively avoid overfitting of the training data. The negative gradients on matrix columns \( U_i \) and \( M_j \) are:

\[ \frac{\partial E}{\partial U_i} = \sum_{j=1}^{m} I_{ij} \left( (V_{ij} - p(U_i, M_j)) \frac{\partial p(U_i, M_j)}{\partial U_i} \right) - k_u U_i, \]  

(2.2)

\[ \frac{\partial E}{\partial M_j} = \sum_{i=1}^{n} I_{ij} \left( (V_{ij} - p(U_i, M_j)) \frac{\partial p(U_i, M_j)}{\partial M_j} \right) - k_m M_j. \]  

(2.3)

The most common prediction function is the dot product of feature vectors. That is, \( p(U_i, M_j) = U_i^T M_j \). The optimization of \( U \) and \( M \) thus becomes a matrix factorization problem that \( V \approx U^T M \). But in most applications, scores in \( V \) are confined to be in an interval \([a, b]\), where \( a \) and \( b \) are the minimal and maximal score values defined in the domain of data. For example, if the users rate the objects as 1−5 stars, then the scores are bounded in the interval \([1, 5]\). One way to restrict the predicted values to be within the interval is to scale the scores to \([0, 1]\), and use a sigmoid function \( g \) in prediction:

\[ p(U_i, M_j) = g(U_i^T M_j), \]  

(2.4)

where

\[ g(x) = \frac{1}{1 + e^{-x}}. \]  

(2.5)

Another method is to clip the values of dot products. For example, we can bound the values of \( U_i^T M_j \) in the interval \([a, b]\) as the prediction values. The other way is to bound them in another interval \([0, b - a]\) and the prediction function becomes \( a \) plus the bounded dot product. The latter is more general as \([a, b]\) may be far away from 0.
Hence, the prediction function is:

\[
p(U_i, M_j) = \begin{cases} 
  a & \text{if } U_i^T M_j < 0, \\
  a + U_i^T M_j & \text{if } 0 \leq U_i^T M_j \leq b - a, \\
  b & \text{if } U_i^T M_j > b - a.
\end{cases}
\] (2.6)

This prediction function is easier to implement and usually leads to better performances than using the sigmoid function, but it is non-differentiable when \(U_i^T M_j = 0\) or \(b - a\). If we would like to keep the differentiability, the prediction values are not clipped during training. That is,

\[
p(U_i, M_j) = a + U_i^T M_j
\] (2.7)

with the gradients

\[
\frac{\partial p(U_i, M_j)}{\partial U_i} = M_j, \\
\frac{\partial p(U_i, M_j)}{\partial M_j} = U_i.
\] (2.8)

(2.9)

But values are clipped in validation and testing. We find that if the true scores are in \([a, b]\), clipping usually leads to a better performance. A variant is that we still clip the values in training but use (2.8) and (2.9) as gradients regardless of the value of \(U_i^T M_j\).

In most cases, the prediction function is differentiable if its value is close to the true score in \(V\). However, this variant does not lead to significantly different performances.

When using the prediction function (2.7), the objective function and its negative gradients have the following forms:

\[
E = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^{n} ||U_i||^2 + \frac{k_m}{2} \sum_{j=1}^{m} ||M_j||^2, (2.10)
\]

\[
-\frac{\partial E}{\partial U_i} = \sum_{j=1}^{m} I_{ij} ((V_{ij} - p(U_i, M_j))M_j) - k_u U_i, \\
-\frac{\partial E}{\partial M_j} = \sum_{i=1}^{n} I_{ij} ((V_{ij} - p(U_i, M_j))U_i) - k_m M_j.
\] (2.11)

(2.12)
One can then perform the optimization of $U$ and $M$ by gradient descent.

The optimization using gradients (2.11) and (2.12) is called batch learning, which modifies $U$ and $M$ according to the whole gradients. The other kind of optimization method is incremental learning, which modifies only some feature values in $U$ and $M$ after scanning part of the training data. For example, if one user $i$ is considered at a time, the variables related to this user are:

1. $U_i$: the feature vector of user $i$,
2. each $M_j$ with $I_{ij} = 1$: the feature vectors of objects scored by user $i$.

If one considers only terms related to user $i$, the objective function becomes:

$$E_i = \frac{1}{2} \sum_{j=1}^{m} I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^{m} I_{ij} (\|M_j\|^2) ,$$  \hspace{1cm} (2.13)

with the negative gradients

$$- \frac{\partial E_i}{\partial U_i} = \sum_{j=1}^{m} I_{ij} ((V_{ij} - p(U_i, M_j)) M_j) - k_u U_i ,$$  \hspace{1cm} (2.14)

$$- \frac{\partial E_i}{\partial M_j} = I_{ij} ((V_{ij} - p(U_i, M_j)) U_i) - k_m I_{ij} (M_j) = I_{ij} [(V_{ij} - p(U_i, M_j)) U_i - k_m M_j] .$$  \hspace{1cm} (2.15)

Note that if $I_{ij} = 0$, then the gradients on $M_j$ is 0. Hence an object not scored by user $i$ is not updated. This incremental learning approach is different from batch learning, as the sum of $E_i$ over all users does not lead to the objective function $E$ given in (2.10):

$$\sum_{i=1}^{n} E_i = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^{n} \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^{m} \sum_{i=1}^{n} I_{ij} (\|M_j\|^2) .$$  \hspace{1cm} (2.16)

Each object feature vector $M_j$ has an additional regularization coefficient $\sum_{i=1}^{n} I_{ij}$, which is equal to the number of existing scores for object $j$. Therefore, an object
with more scores has a larger regularization coefficient in this incremental learning approach.

An extreme case of incremental learning is to update the features after looking at each single score. That is, we consider the following objective function and negative gradients if $V_{ij}$ is not missing:

\[
E_{ij} = \frac{1}{2} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \|U_i\|^2 + \frac{k_m}{2} \|M_j\|^2, \tag{2.17}
\]

\[-\frac{\partial E_{ij}}{\partial U_i} = (V_{ij} - p(U_i, M_j))M_j - k_u U_i, \tag{2.18}\]

\[-\frac{\partial E_{ij}}{\partial M_j} = (V_{ij} - p(U_i, M_j))U_i - k_m M_j. \tag{2.19}\]

The summation of $E_{ij}$ over all existing votes is

\[
\sum_{i=1}^n \sum_{j=1}^m E_{ij} = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^m I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^n \sum_{j=1}^m I_{ij} \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^m \sum_{i=1}^n I_{ij} \|M_j\|^2. \tag{2.20}\]

In this case, each user and object has a regularization coefficient which is proportional to the number of existing scores related to that user or object. We denote this type of learning as \textit{complete incremental learning}, and the one in (2.13), which considers multiple scores at a time as \textit{incomplete incremental learning}. These two types of learning methods have different behaviors and will be compared in Chapter IV.

The performance of an algorithm is measured by Root Mean Square Error (RMSE) as equation (1.1), where the prediction matrix $P$ is computed by the chosen prediction function. The training RMSE is proportional to the objective function without the regularization term:

\[
\text{training RMSE} = \sqrt{\frac{\sum_{i=1}^n \sum_{j=1}^m I_{ij} (V_{ij} - p(U_i, M_j))^2}{\sum_{i=1}^n \sum_{j=1}^m I_{ij}}}. \tag{2.21}\]

During the update of feature values, the test RMSE decreases at first. It then increases at some point while the training RMSE continues to decrease. At this time,
the optimization of matrices $U$ and $M$ should be stopped since it starts overfitting the training data. By using good regularization coefficients, the algorithm can slow down the overfitting and reach a lower test RMSE.

### 2.2 Optimization

The values of feature matrices $U$ and $M$ are optimized by minimizing the objective function using gradient descent. Usually the objection function as well as training RMSE are easily decreased, but the actual performance of the algorithm, measured by the test RMSE, may suffer from overfitting. To decide when to stop the learning procedure, a validation set is often used to prevent overfitting.

The following two algorithms describe the optimization of $U$ and $M$ in the SVD framework. Algorithm 1 performs batch learning, and Algorithm 2 does incremental learning, which updates the feature values after examining each single score.

**Algorithm 1 (Batch learning of Singular Value Decomposition)**

Select a learning rate $\mu$, and regularization coefficients $k_u, k_m$.

1. Set the starting values of matrices $U, M$.

2. Repeat

   (a) Compute gradients $\nabla_U$ and $\nabla_M$ by (2.11) and (2.12).

   (b) Set $U \leftarrow U - \mu \nabla_U, M \leftarrow M - \mu \nabla_M$.

until the validation RMSE starts to increase.

**Algorithm 2 (Incremental learning of Singular Value Decomposition)**

Select a learning rate $\mu$, and regularization coefficients $k_u, k_m$.

1. Set the starting values of matrices $U, M$.

2. Repeat
(a) For each existing score $V_{ij}$ in training data

i. Compute gradients $\nabla U_i$ and $\nabla M_j$ by (2.18) and (2.19).

ii. Set $U_i \leftarrow U_i - \mu \nabla U_i$, $M_j \leftarrow M_j - \mu \nabla M_j$.

until the validation RMSE starts to increase.

In Algorithm 2, the scores in training data are scanned in a given order. Instead of using the full matrices in Algorithm 1, only the feature vectors of one user and one object are updated at a time. The choice of orders does not lead to significant differences in performances. In general, these two algorithms cause different optimization results and performance. However, if $k_u$ and $k_m$ are 0, both algorithms have similar performances, and Algorithm 1 needs an extremely small learning rate to prevent divergence. Hence, the learning time of Algorithm 1 may be much longer. When regularization is involved, Algorithm 2 is significantly better than Algorithm 1 no matter what values of $k_u$ and $k_m$ are used. Recall that the objective function (2.10) of batch learning is not equal to the summation of objective functions (2.20) of incremental learning. If Algorithm 1 is modified so that (2.20) is used as its objective function then it can achieve a similar performance to Algorithm 2 under the same regularization coefficients. However, a smaller learning rate and longer learning time are required. Although the learning speed can be improved by adding a momentum term in the gradient descent procedure, in general batch learning is not suitable for SVD algorithms on a sparse and unbalanced data. The numerical range of gradients in batch learning is so large that the setting of parameters can not work well on all variables.

The tuneable parameters are the learning rate $\mu$, and the regularization coefficients $k_u$ and $k_m$ for user and object features respectively. The learning rate affects the learning time, and a too large value may lead to divergence. In general, a smaller learning rate gives better performance, but the learning time is also longer. The regularization
coefficients prevent the feature value from being extreme, which is the main cause of overfitting. Small regularization coefficients may not have a significant effect, but a large value may decrease the learning speed and even give inferior performance.

Another setting that affects the performance is the starting point of feature values. One way is to use random values in a specific range, but unstable performances may occur if these values are too random. In Algorithms 1 and 2, the starting point can be the average of all existing scores $\bar{V}$:

$$U_{ij}, M_{ij} = \sqrt{\bar{V} - a_f + n(r)} \text{ for each } i, j,$$

(2.22)

where $a$ is the lower bound of scores, $f$ is the dimension of SVD algorithms, and $n(r)$ is a random noise with uniform distribution in $[-r, r]$. According to the prediction function (2.7), it is like to predict all values to the global average $\bar{V}$ with some noises at the start of the algorithm. Without the random noise in (2.22), the features of a user or an object will be the same as they always have the same gradients during optimization. A small value of $r$ is usually enough. This kind of starting points is very useful for Algorithm 2, as it can give the lowest test RMSE compared with other settings. For example, the starting point can also be set by the average score of each user and object. This setting has a lower training RMSE at the beginning of the optimization procedure, but it overfits faster and leads to a higher test RMSE.

Besides Algorithms 1 and 2, there are other methods for the optimization of $U$ and $M$. Rather than considering the whole feature matrix, the optimization can be performed on one row of $U$ and $M$ at a time. The procedure is described in the following algorithm. During the update of feature $k$, a single vote $V_{ij}$ only changes the $k$-th feature value of user $i$ and object $j$. i.e., $U_{ki}$ and $M_{kj}$.

**Algorithm 3 (Feature-wised learning of Singular Value Decomposition)**

Select a learning rate $\mu$, and regularization coefficients $k_u, k_m$. 

1. Set the starting values of matrices $U, M$.

2. For $k = 1$ to $f$
   (a) Repeat
      i. For each existing score $V_{ij}$ in the training data
         A. Compute gradients $\nabla U_{ki}$ and $\nabla M_{kj}$ by (2.18) and (2.19).
         B. Set $U_{ki} \leftarrow U_{ki} - \mu \nabla U_{ki}$, $M_{kj} \leftarrow M_{kj} - \mu \nabla M_{kj}$.

   until no significant decrease of the objective function.

Algorithm 3 is similar to Algorithm 2, but there are several differences. First, the stopping condition of Algorithm 3 is not from using the validation RMSE, but is decided by the decrease of the objective function. The validation RMSE may not be suitable here as it may increase during the first several iterations for a feature. But it will go down after more iterations on that feature. Therefore, an additional parameter $\text{miniter}$ is usually used so at least $\text{miniter}$ iterations are conducted for each feature. After that, the optimization on this feature is performed until the decrease of objective function is less than a threshold. Since the validation RMSE is not required, the validation data are also used as training data in Algorithm 3. Moreover, the initial $U$ and $M$ should be different from (2.22). A small value like 0.1 for each $U_{ij}, M_{ij}$ is enough to give good performance. In addition, the random noise in (2.22) is not required.

If the prediction function is like (2.7), one can speed up Algorithm 3 by caching. When 3 is processing feature $k$, only the values of feature $k$ are updated by the algorithm and the values of other $f - 1$ features are kept fixed. Hence, the dot product on
those \( f - 1 \) features for each training score \( V_{ij} \) can be recorded in a cache matrix \( C \):

\[
C_{ij} = \begin{cases} 
U_i^T M_j - U_{ki} M_{kj} & \text{if } V_{ij} \text{ exists}, \\
0 & \text{if } V_{ij} \text{ is missing},
\end{cases}
\]  

(2.23)

where \( U_{ki}, M_{kj} \) are the \( k \)-th features of user \( i \) and object \( j \). The cache matrix is computed when the procedure of feature \( k \) starts, and its values are fixed when feature \( k \) is updated by iterations. Hence, the value of dot product \( U_i^T M_j \) can be efficiently computed by the cache while the values of \( U_{ki} \) and \( M_{kj} \) are changed:

\[
U_i^T M_j = C_{ij} + U_{ki} M_{kj},
\]  

(2.24)

without computing the dot product actually. However, it is still slower than the simple incremental learning like Algorithm 2 and also has an inferior performance.

In summary, from our experiments Algorithm 2 is the best in both efficiency and performance. If we would like to use both training and validation data to predict the unknown scores, we can record the number of iterations in Algorithm 2 when using the training/validation split. After that, the training and validation data are merged as the final training data and Algorithm 2 is run again with that number of iterations. In the Netflix Prize contest, this approach can give a significantly lower RMSE than using only the training data.

### 2.3 Variants

With proper optimization settings, the SVD algorithm is able to give a good performance. However, variants of the algorithm have the potential for making more accurate predictions. Here we discuss some of them. The simplest one is to add per-user bias \( \alpha \in \mathbb{R}^{n \times 1} \) and per-object bias \( \beta \in \mathbb{R}^{m \times 1} \) on the prediction function (Paterek, 2007).
That is, the prediction function is modified to

\[
p(U_i, M_j, \alpha_i, \beta_j) = a + U_i^T M_j + \alpha_i + \beta_j,
\]  

(2.25)

where \(\alpha_i\) is the bias of user \(i\) and \(\beta_j\) is the bias of object \(j\). Similar to the previous setting, the predicted values are only clipped in validation and testing.

The biases are updated like the feature values. For the “complete incremental learning”, using a single score \(V_{ij}\) at a time gives the following negative gradients of biases \(\alpha_i\) and \(\beta_j\):

\[
E_{ij} = \frac{1}{2} (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))^2 + \frac{k_u}{2} \|U_i\|^2 + \frac{k_m}{2} \|M_j\|^2 + \frac{k_b}{2} (\alpha_i^2 + \beta_j^2),
\]  

(2.26)

\[
-\frac{\partial E_{ij}}{\partial \alpha_i} = (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) - k_b \alpha_i,
\]  

(2.27)

\[
-\frac{\partial E_{ij}}{\partial \beta_j} = (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) - k_b \beta_j,
\]  

(2.28)

where \(k_b\), the regularization coefficient of biases, is similar to \(k_m\) and \(k_u\). Of course, the learning rate of biases can be different from others.

Another variant of SVD algorithms, called constrained SVD, adds additional constraints to each user feature vector \(U_i\) (Salakhutdinov and Mnih, 2008). In this algorithm, the user feature matrix \(U \in \mathbb{R}^f \times n\) is replaced by a matrix \(Y \in \mathbb{R}^f \times n\) of the same size, and an object constraint matrix \(W \in \mathbb{R}^f \times m\) shifts the values of user features:

\[
U_i = Y_i + \sum_{k=1}^{m} \frac{I_{ik} W_k}{\sum_{k=1}^{m} I_{ik}},
\]  

(2.29)

where \(W_k\) is the \(k\)-th column of the matrix \(W\). In other words, the feature vector \(U_i\) is the user-dependent feature vector \(Y_i\) plus an offset \(\sum_{k=1}^{m} \frac{I_{ik} W_k}{\sum_{k=1}^{m} I_{ik}}\), which is the mean of object constraint feature vectors on the objects scored by user \(i\). Each \(W_k\) contributes to a user feature \(U_i\) if user \(i\) scores object \(k\).
Since our experiments show that batch learning is less suitable for collaborative filtering, we consider the error function of a single score for constrained SVD:

\[
E_{ij} = \frac{1}{2}(V_{ij} - p(U_i, M_j))^2 + \frac{k_y}{2}\|Y_i\|^2 + \frac{k_m}{2}\|M_j\|^2 + \frac{k_w}{2}\sum_{k=1}^{m} I_{ik} (\|W_k\|^2),
\]

\[
-\frac{\partial E_{ij}}{\partial Y_i} = (V_{ij} - p(U_i, M_j))M_j - k_y Y_i, \tag{2.31}
\]

\[
-\frac{\partial E_{ij}}{\partial M_j} = (V_{ij} - p(U_i, M_j))U_i - k_m M_j, \tag{2.32}
\]

\[
-\frac{\partial E_{ij}}{\partial W_k} = (V_{ij} - p(U_i, M_j)) \frac{I_{ik} M_j}{\sum_{k=1}^{m} I_{ik}} - k_w I_{ik} (W_k) = I_{ik} \left( (V_{ij} - p(U_i, M_j)) \frac{M_j}{\sum_{k=1}^{m} I_{ik}} - k_w W_k \right), \tag{2.33}
\]

\[k = 1, \ldots, m.\]  \tag{2.34}

Note that \(U_i\) follows the relation given in (2.29), and \(p(U_i, M_j)\) is the prediction function described in (2.7). Each single score \(V_{ij}\) appears in the gradients on the user feature \(Y_i\), the object feature \(M_j\), and each object constraint feature \(W_k\) with \(I_{ik} = 1\) (i.e., each \(W_k\) that object \(k\) is scored by user \(i\)). The variable \(I_{ik}\) in (2.33) shows that if an object is not scored by user \(i\), then the corresponding feature vectors in \(W\) are not updated during the learning procedure about user \(i\).

For the original SVD algorithm, complete incremental learning which updates the feature values at each score like Algorithm 2 can give the best performance with the fastest speed. However, in constrained SVD, this incremental learning setting has a problem in updating \(W\). Since each single score \(V_{ij}\) affects all \(W_k\) that user \(i\) scores object \(k\), the update for all scores of user \(i\) causes enormous computation on \(W\). This operation cannot be simplified since the gradient of \(W\) depends on the current values of \(W\) when regularization is used. In order to solve this problem, a compound algorithm is proposed in the next section. Compared with the SVD algorithm using complete
incremental learning like Algorithm 2, the compound SVD algorithm gives a better performance with the same time complexity.

2.4 Compound SVD

Consider the combination of constrained SVD and user/object biases. The error function of a single score $V_{ij}$ becomes:

$$E_{ij} = \frac{1}{2}(V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))^2 + \frac{k_y}{2}\|Y_i\|^2 + \frac{k_m}{2}\|M_j\|^2 + \frac{k_w}{2} \sum_{k=1}^m I_{ik} (\|W_k\|^2) + \frac{k_b}{2}(\alpha_i^2 + \beta_j^2),$$

where $U_i$ is computed by (2.29) and the prediction function $p(U_i, M_j, \alpha_i, \beta_j)$ follows (2.25). The user feature vector $Y_i$ and object feature vector $M_j$ can be updated using the negative gradients of (2.35):

$$-\frac{\partial E_{ij}}{\partial Y_i} = (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))M_j - k_yY_i,$$

$$-\frac{\partial E_{ij}}{\partial M_j} = (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))U_i - k_mM_j,$$

and the biases $\alpha_i$ and $\beta_j$ are updated by (2.27) and (2.28). In other words, the values in $Y$ and $M$ as well as all biases are updated as the complete incremental learning like Algorithm 2.

However, the values in the object constraint matrix $W$ could not be updated at each single score due to the high computational cost. We propose a feasible way under the condition that the scores in training data are ordered by the user ID. That is, during the scanning of scores, the algorithm examines all scores made by the first user, and the corresponding variables are updated. Then it proceeds to the second user, and the third, and so on. During the update procedure of user $i$, the values in $W$ are temporarily fixed while other variables are updated as described above. By (2.29), the computationally expensive term $\frac{\sum_{k=1}^m I_{ik}W_k}{\sum_{k=1}^m I_{ik}}$ is also fixed and can be precomputed.
Although the update of $Y_i$ also changes the value of $U_i$, its time requirement is much smaller than computing (2.29). After that, each $W_k$ with $I_{ik} = 1$ is updated as the final optimization step for user $i$.

The following algorithm, called Compound SVD, uses the learning method described above to perform the optimization. We explain why it can effectively reduce the computational time. Note that the gradients of (2.35) on $Y_i$ and $W_k$ are proportional to each other when the regularization coefficient $k_w$ is 0:

$$\frac{-\partial E_{ij}}{\partial Y_i} = (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) M_j,$$

$$\frac{-\partial E_{ij}}{\partial W_k} = I_k \left( (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) \frac{M_j}{\sum_{k=1}^m I_{ik}} - 0 \right)$$

$$= \frac{I_k}{\sum_{k=1}^m I_{ik}} (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) M_j.$$  (2.39)

Since we update $W$ only once in the update procedure of user $i$, we consider the error function of all scores made by user $i$ and its gradient on $W_k$:

$$E_i = \frac{1}{2} \sum_{j=1}^m I_{ij} (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))^2 + \frac{k_y}{2} \|Y_i\|^2 + \frac{k_m}{2} \sum_{j=1}^m I_{ij} \left( \|M_j\|^2 \right) + \frac{k_w}{2} \sum_{k=1}^m I_{ik} \|W_k\|^2,$$

$$- \frac{\partial E_i}{\partial W_k} = \sum_{j=1}^m I_{ij} \left( (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) \frac{I_k M_j}{\sum_{k=1}^m I_{ik}} \right) - k_w I_{ik} W_k$$

$$= I_k \left[ \sum_{j=1}^m I_{ij} \left( (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) \frac{M_j}{\sum_{k=1}^m I_{ik}} \right) - k_w W_k \right].$$  (2.41)

The summation of (2.39) over all $j$ that $I_{ij} = 1$ gives the gradient on $W_k$ for user $i$ as (2.41) except the regularization term:

$$\sum_{j=1}^m I_{ij} \left( \frac{I_k}{\sum_{k=1}^m I_{ik}} (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) M_j \right)$$

$$= I_k \left[ \sum_{j=1}^m I_{ij} \left( (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) \frac{M_j}{\sum_{k=1}^m I_{ik}} \right) \right].$$  (2.42)
Hence, by ordering the training scores according to the user ID, we can save the term $(V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))M_j$ for each $j$ when computing the gradient on $Y_i$ for each score $V_{ij}$ made by user $i$. Then we use them to compute the gradient of each $W_k$ later. Moreover, the term (2.42) is identical for each $k$ with $I_{ik} = 1$ since $\sum_{k=1}^{m} I_{ik}$ is the number of objects scored by user $i$. The regularization coefficient $k_w$ in (2.41) is still needed to prevent overfitting. The whole procedure is listed in Algorithm 4.

**Algorithm 4 (Compound Singular Value Decomposition)**

Select learning rates $\mu_v, \mu_w, \mu_b$, and regularization coefficients $k_{yi}, k_{mi}, k_{wi}, k_{bi}$.

1. Set the starting values of matrices $Y, M, W$.

2. Repeat

   (a) For user $i = 1$ to $n$

      i. Compute user feature vector $U_i$ from $Y_i$ and $W$ by (2.29).

      ii. Initialize the gradient buffer $\mathbf{g}$ to $0^{f \times 1}$.

      iii. For each existing score $V_{ij}$ made by user $i$

          A. Compute gradient $\nabla Y_i$ by (2.36), and also set $\mathbf{g} \leftarrow \mathbf{g} + (V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))M_j$.

          B. Compute gradient $\nabla M_j$ by (2.37).

          C. Compute gradients $\nabla \alpha_i$ and $\nabla \beta_j$ by (2.27) and (2.28).

          D. Set $Y_i \leftarrow Y_i - \mu_v \nabla Y_i$, $U_i \leftarrow U_i - \mu_v \nabla Y_i$, $M_j \leftarrow M_j - \mu_v \nabla M_j$.

          E. Set $\alpha_i \leftarrow \alpha_i - \mu_b \nabla \alpha_i$, $\beta_j \leftarrow \beta_j - \mu_b \nabla \beta_j$.

      iv. For each $k$ that object $k$ is scored by user $i$

          A. Compute gradient $\nabla W_k$ from $\mathbf{g}$ and $k_w$ by (2.41).

          B. Set $W_k \leftarrow W_k - \mu_w \nabla W_k$.
until the validation RMSE starts to increase.

There are two special settings in Algorithm 4. First, the values of $U_i$ are precomputed for calculating the gradients, but they are not temporarily fixed like $W$. When the feature values are updated, $U_i$ is shifted with the same movement as $Y_i$ to preserve the relation (2.29) without recomputing it. If we set all values in $W$ and the learning rate $\mu_w$ to 0, Algorithm 4 is identical to Algorithm 2. Second, the gradient of each $W_k$ is computed by $g$ since

$$g = \sum_{j=1}^{m} I_{ij} ((V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))M_j)$$

(2.43)

when the gradient $\nabla W_k$ is computed, and

$$\frac{g}{\sum_{k=1}^{m} I_{ik}} - k_w W_k = \frac{\sum_{j=1}^{m} I_{ij} ((V_{ij} - p(U_i, M_j, \alpha_i, \beta_j))M_j)}{\sum_{k=1}^{m} I_{ik}} - k_w W_k$$

(2.44)

$$= \sum_{j \neq i}^{m} I_{ij} ((V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)) \frac{M_j}{\sum_{k=1}^{m} I_{ik}})$$

(2.45)

which is exactly (2.41) for those $W_k$ with $I_{ik} = 1$. Hence, Algorithm 4 updates $Y$, $M$ and the biases $\alpha, \beta$ like Algorithm 2, but updates $W$ as if it considers one user at a time. Since the gradient of $W$ can be computed efficiently by (2.44), Algorithm 4 avoids the time complexity caused by updating the object constraint matrix $W$.

Besides the original parameters in Algorithm 2, Algorithm 4 has some additional parameters. The learning rates $\mu_v$ for matrix $Y$, $\mu_w$ for matrix $W$, and $\mu_b$ for biases $\alpha, \beta$ can all be different. Practically the same value of $\mu_v$ and $\mu_w$ works well while $\mu_b$ is usually smaller. The regularization coefficients can be determined by the validation data, and the starting point of $W$ is easy to be set: if $Y$ and $M$ are initialized by (2.22), all values in $W$ can just be initialized to 0. Algorithm 4 is able to give a better performance than Algorithm 2 under appropriate parameter values.
As a comparison of time requirements, we show the time complexity for an iteration of Algorithms 2 and 4 respectively. Algorithm 2 computes the gradients $\nabla U_i$ and $\nabla M_j$ for each training score $V_{ij}$, but the gradients (2.18) and (2.19) have a common term $V_{ij} - p(U_i, M_j)$, which is computed only once for a given $V_{ij}$. Hence, for an SVD algorithm with dimension $f$, it takes $O(f)$ time to update the values of feature vectors $U_i$ and $M_j$ by $\nabla U_i$ and $\nabla M_j$. A whole iteration in Algorithm 2 has the time complexity $O(ft)$, where $t$ is the number of existing scores in the training data.

The compound SVD algorithm, Algorithm 4, has a loop over users, where each step contains an inner loop for the scores given by that user. Similar to Algorithm 2, there is a common term $V_{ij} - p(U_i, M_j, \alpha_i, \beta_j)$ used for the gradients. Hence the operations in the inner loop require only $O(f)$ time for a score $V_{ij}$ and accumulate to $O(ft)$ for the whole loop over users. Other operations of the algorithm include the computation of $U_i$ before the inner loop and the update of $W$ after it. Both of them involve the constraint matrix $W$ and are performed for each user during the loop over users. Let $t_i$ be the number of scores given by user $i$ as the numbers of scores given by users are different. Obviously, $\sum_{i=1}^{n} t_i = t$. The computation of $U_i$ includes the average of $t_i$ columns in $W$, and they are exactly the columns which should be updated for user $i$. The time complexity of these two operations are $O(ft_i)$ for user $i$, and the summation over all $n$ users becomes $O(ft)$. Hence, an iteration of Algorithm 4 also requires $O(ft)$ time, the same as Algorithm 2 though it has more operations and a larger overhead. The training time of these algorithms in experiments is shown in Chapter IV.
CHAPTER III

Post-Processing

While an algorithm based on Singular Value Decomposition could give good performances, better results can usually be achieved by combining various methods which have different strengths and weaknesses. For example, the final prediction of an unknown score can be made by the linear combination of prediction values from two or more algorithms. In this chapter, we consider an approach that uses one algorithm as a post-processor of another algorithm.

3.1 Residuals

Let us consider Algorithm 4 in Section 2.4. It optimizes the user and object feature matrices $U$ and $M$ and the biases $\alpha$ and $\beta$ using the training data $V$. The residuals of training data can be computed as a matrix $R \in \mathbb{R}^{n \times m}$:

$$R_{ij} = \begin{cases} V_{ij} - p(U_i, M_j, \alpha_i, \beta_j) & \text{if } V_{ij} \text{ exists}, \\ 0 & \text{if } V_{ij} \text{ is missing.} \end{cases}$$

(3.1)

If $V_{ij} \neq 0$, in general $R_{ij} \neq 0$ as we avoid overfitting training data. Occasionally existing $V_{ij}$’s corresponding $R_{ij}$ is zero. In this case, we set it to a small value $10^{-10}$.

After computing the residuals, we can use them to update the biases $\alpha$ or $\beta$. Consider $R_j$, the $j$th column of $R$, which represents the residuals of object $j$. The
average residual of object $j$ is

$$\bar{R}_j = \frac{\sum_{i=1}^{n} I_{ij} R_{ij}}{\sum_{i=1}^{n} I_{ij}},$$

(3.2)

where $I$ is the existing score indicator of both $V$ and $R$. Then, we may update the residuals and per-object biases according to the average residuals:

$$\beta_j \leftarrow \beta_j - \bar{R}_j,$$

(3.3)

$$R_{ij} \leftarrow \begin{cases} R_{ij} - \bar{R}_j & \text{if } I_{ij} = 1, \\ 0 & \text{if } I_{ij} = 0, \end{cases}$$

(3.4)

for $j = 1$ to $m$. This update can adjust the biases using training data, and improve the testing accuracy. Of course, one can also update the per-user biases rather than per-object ones by the same way. However, the update of per-user biases is more possible to cause overfitting for unbalanced data that some users have only a few number of scores.

In order to further improve the performance of our algorithms, we can apply additional post-processing by exploiting the residual matrix $R$. By observing the residuals in training data, we estimate the residuals of predicting test data. If we use another SVD algorithm to estimate the residuals, the result is not significant as this approach is analogous to simply increase the dimension of a feature-wised SVD algorithm like Algorithm 3. Hence, in the rest of this chapter, we consider other conventional methods used for collaborative filtering, like $K$-nearest neighbors and regression.

3.2 $K$-Nearest Neighbors

One of the most popular ways to predict an unknown score is to check other scores that have similar properties. This method is known as $K$-Nearest Neighbors (KNN).
between two scores, since the only features of a score in the training data are its corresponding user and object ID, and the distribution of scores may also be sparse and unbalanced.

To begin, we define the distance function between two users or two objects. The distance can be obtained from the training matrix $V$. For example, one can compute the distance between objects $i$ and $j$ from two columns $V_i$ and $V_j$. However, the sparseness of data will cause some problems. For example, the training data of Netflix Prize has 480,189 customers and 17,770 users. Due to the sparseness of data, two columns of objects only expect to have 67 scores, and two rows of users even have only two overlaps on average. Hence, the definition of the distance function is difficult for a sparse data set. A possible way is to use the SVD feature matrices $U$ and $M$. That is, the distance between objects $i$ and $j$ is defined as:

\[ d(u_i, u_j) = \|U_i - U_j\|, \]  
\[ d(o_i, o_j) = \|M_i - M_j\|, \]  

where $u_i, u_j$ and $o_i, o_j$ represent the $i$th, $j$th users and objects, respectively.

The basic assumption of this method is that if two users or objects are similar, then their features computed by an SVD algorithm should be close. However, algorithms in Chapter II may give different distances. For example, if two objects are similar in character but different in quality (e.g., two movies of the same kind but one is obviously better than the other), then they tend to have similar feature vectors if per-object biases are used.

The computation of distance from training matrix $V$ is time-consuming. For example, to compute the distance between two object columns in $V$, we have to check the overlap of two $n \times 1$ vectors which requires $O(n)$ time in the worst case. If the distance is computed by the feature matrices, it is much more efficient to precompute
the $k$ nearest neighbors of each user or object since the length of a feature vector is the dimension of SVD. After that, a KNN algorithm is used to predict the residuals of predictions in the test data. Usually a KNN algorithm is based on only the $K$-nearest neighbors of each user or each object, but not both because of the time complexity. The following algorithm runs KNN based on the neighbors of each object, and predicts the residual of a score by other residuals of scores made by the same user.

**Algorithm 5 (K-nearest neighbors on the residuals of SVD)**

Given the feature matrices $U$ and $M$ computed by an SVD algorithm.

1. Compute neighbor matrix $N \in \mathbb{N}^{K \times m}$ from the distance function (3.6), where the column $N_j$ records the $K$ nearest neighboring objects of object $j$.

2. For each user/object pair $(i, j)$ in the test data
   (a) Set $S = \{N_{1j}, N_{2j}, ..., N_{Kj}\}$.
   (b) Find $S' \in S$ with only the objects scored by user $i$ in the training data. Assume $l$ is the number of objects in $S'$.
   (c) Let $r_1, r_2, ..., r_l$ be the residuals of scores corresponding to user $i$ and objects in $S'$.
   (d) Predict the residual $\hat{r}$ of the test pair $(i, j)$ as

   $$\hat{r} = \frac{\sum_{a=1}^{l} r_a}{l + k_r},$$

   where $k_r \geq 0$ is a regularization coefficient.

The parameter $k_r$ in Algorithm 5 works against the sparseness of data. For a user who gives only few scores, $S'$ may contain only one or two objects. If we use the average of residuals directly, overfitting may occur for such users. This problem is serious in real-case collaborative filtering, because the algorithm should predict the preference of
each user equally regardless of the number of scores they give. With an appropriate $k_r$, the predicted residual avoids being extreme when the size of $S'$ is small. On the contrary, if $S'$ contains more elements, the average of residuals is more reliable and the effect of $k_r$ is negligible.

### 3.3 Regression

Regression is a popular method to predict scores (Paterek, 2007) or residuals. Consider the existing scores made by user $i$. A feature matrix $X \in \mathbb{R}^{t \times f}$ is constructed by an SVD algorithm with dimension $f$. Each row $x_i$ in $X$ is the feature vector of an object scored by user $i$, and $t$ is the number of scores for this user. Another column vector $Y \in \mathbb{R}^{t \times 1}$ contains the target values of the regression problem, which are the true scores or residuals of those objects. One can then perform linear or nonlinear regression to predict the unknown scores or residuals in validation or test data. In this section, we use the compound SVD algorithm. The matrix $X$ contains the feature vectors of objects in rows, which are computed by Algorithm 4 and then normalized by their lengths.

One typical linear regression method is ridge regression, which computes a weight vector $w \in \mathbb{R}^{f \times 1}$ by:

$$w = X^T(XX^T + \lambda I_t)^{-1}Y;$$

where $I_t$ is a $t \times t$ identity matrix and $\lambda$ is the regularization coefficient. An unknown residual of user $i$ is then predicted by:

$$\hat{y} = xw = xX^T(XX^T + \lambda I_t)^{-1}Y;$$

where $x$ is the normalized feature vector of the corresponding object. The linear regression is similar to an SVD algorithm which fixes the values of object features, and
tries to find the feature values of each user. A variant of ridge regression, called kernel ridge regression, involves a kernel function $K(x_i, x_j)$ on two row vectors. The predict function of target values becomes:

$$\hat{y} = K(x, X)(K(X, X) + \lambda I)^{-1}Y.$$  \hspace{1cm} (3.10)

If $K(x_i, x_j) = x_i x_j^T$, (3.10) is identical to the linear case (3.9). The following algorithm performs kernel ridge regression on each user to predict the residuals of scores in test data.

**Algorithm 6 (Kernel ridge regression on the residuals of SVD)**

Given the feature matrices $U$ and $M$ computed by an SVD algorithm.

1. For user $i = 1$ to $n$

   (a) Construct normalized feature matrix $X^{t \times f}$ and target vector $Y^{t \times 1}$ for user $i$, where $t$ is the number of scores made by user $i$ in the training data. Each row of $X$ is the normalized feature vector of an object scored by user $i$.

   (b) Compute $Z = (K(X, X) + \lambda I)^{-1}Y$.

   (c) For each (user, object) pair $(i, j)$ in the test data

      i. Set $x \leftarrow \frac{M_j}{\|M_j\|}$, the normalized feature vector of object $j$, where $M_j$ is the $j$-th column of matrix $M$.

      ii. Predict the residual $\hat{y}$ of the test pair $(i, j)$ as $\hat{y} = K(x, X)Z$.

Algorithm 6 is able to reach a better performance than Algorithm 5, but it requires long time to compute the matrix $Z$, which involves the kernel calculation and matrix inversion. Usually the algorithm has a restriction on $t$, the number of objects scored by a user. If $t$ is larger than a threshold, like 1,000, then we only use the $t$ most frequently scored objects among all objects scored by the user to construct $X$ and $Y$. However, its time requirement is still unacceptable.
Another issue is the choice of the kernel function. Since we normalize the object features in $X$, the matrix multiplication $XX^T$ becomes the dot products of every pair of normalized features. The values of dot products have the range $[-1, 1]$, and stand for the similarity of two unit vectors. An appropriate choice of the kernel function may vary in different problems. In our experiment, we find that a polynomial kernel is suitable for the regression on residuals. That is, we use the kernel function:

$$K(x_i, x_j) = \begin{cases} (x_i x_j^T)^p & \text{if } x_i x_j^T \geq 0, \\ 0 & \text{if } x_i x_j^T < 0, \end{cases} \quad (3.11)$$

where $p$ is usually a number larger than 1. For example, on the regression for residuals of a compound SVD algorithm with dimension 10, the best $p$ is 20. This setting implies that only the feature pairs with high similarity have effects in the algorithm. Since $XX^T$ contains the dot products of normalized feature pairs, $K(X, X)$ is almost an identical matrix under such a kernel function. Hence, we can speed up the algorithm by replacing $K(X, X)$ with $I_t$, but still use the kernel function in prediction. The simplified prediction function becomes:

$$\hat{y} = K(x, X)(I_t + \lambda I_t)^{-1}Y$$

$$= K(x, X)(\frac{1}{1+\lambda})I_t Y \quad (3.13)$$

$$= \sum_{a=1}^{t} K(x, x_a) \frac{1}{1+\lambda} y_a, \quad (3.14)$$

where $x_a$ is the $a$-th row of $X$ and $y_a$ is the $a$-th value of $Y$. Equation (3.14) shows that the prediction is a weighted sum of residuals, and the weight is proportional to the kernel function value. The matrix $Z$ in Algorithm 6 is now $\frac{1}{1+\lambda} Y$ and is faster to be computed. Under appropriate parameters $p$ and $\lambda$, the modified algorithm can still raise the performance of the base algorithm, though it is not as good as Algorithm 6.
One drawback of equation (3.14) is that it is only the weighted sum of the residuals, but not the weighted average. When the number of similar features is large, the sum may be out of the bound and give more errors in prediction. Hence, we modify equation (3.14) again so that it becomes the weighted average of residuals:

\[
\hat{y} = \frac{\sum_{a=1}^{t} K(x, x_a)y_a}{\sum_{a=1}^{t} K(x, x_a) + \lambda}.
\]  

(3.15)

Algorithm 6 now becomes the following procedure:

**Algorithm 7 (Weighted average on the residuals of SVD)**

Given the feature matrices \(U\) and \(M\) computed by an SVD algorithm.

1. For user \(i = 1\) to \(n\)
   
   (a) Construct normalized feature matrix \(X\) and target vector \(Y\) for user \(i\).
   
   (b) For each (user, object) pair \((i, j)\) in the test data
      
      i. Set \(x \leftarrow \frac{M_j}{\|M_j\|}\), the normalized feature vector of object \(j\).
      
      ii. Predict the residual \(\hat{y}\) of the test pair \((i, j)\) by (3.15).

Algorithm 7 runs much faster than Algorithm 6 even without a threshold on the value of \(t\). Moreover, its performance is the best among all post-processing algorithms described above.
CHAPTER IV

Experiments

In this chapter, we show the performances of our algorithms via experiments. Two data sets *Movielens* (Miller et al., 2003) and *Netflix* (Bennett and Lanning, 2007) are used for evaluation. The performances are measured by RMSE (1.1). Both data sets are divided into two disjointed training and test sets. We present the test RMSEs along with training time.

4.1 Data Sets

The smaller data set *Movielens* is provided by GroupLens (Konstan et al., 1997), a research group which collects and supplies several collaborative filtering data in different domains. The data set contains 1,000,209 scores for 6,040 users and 3,706 movies. Each user gives at least 20 scores. The density of the data matrix is 4.61%. Since the data set does not come with a training and test split, we randomly select three scores of each user as test data. Hence the ratio of training and test sets is similar to the *Netflix* data set.

The other data set is provided by Netflix Prize as mentioned in Chapter I. There are 100,480,507 scores for 480,189 users and 17,770 movies. Each user has at least 20 scores in the beginning, but the competition organizers intentionally perturb the data set so some users may have less than 20 scores. The density of the data matrix
is 1.18%, which is sparser than \textit{Movielens}. The test data used for experiments are exactly the probe set provided by Netflix Prize, which includes 1,408,395 scores. The training data are the remaining 99,072,112 scores.

4.2 Experimental Results

We implement the SVD algorithms in C, since they are the most time-consuming in computation. We use MATLAB for data preprocessing and postprocessing as it is convenient for matrix manipulation. For different SVD algorithms, the test RMSEs are plotted against the training time. Algorithms used for comparison are listed below:

\textbf{AVGB}: A simple baseline predictor that predicts the score \( P_{ij} \) for user \( i \) and object \( j \) as \( \mu_j + b_i \), where \( \mu_j \) is the average score of object \( j \) in training data, and \( b_i \) is the average bias of user \( i \) computed as:

\[
b_i = \frac{\sum_{j=1}^{m} I_{ij}(V_{ij} - \mu_j)}{\sum_{j=1}^{m} I_{ij}}.
\] (4.1)

\textbf{SVDNR}: The SVD algorithm without regularization terms. This algorithm works as another baseline.

\textbf{SVD}: Algorithm 2, which uses incremental learning with regularization terms.

\textbf{SVDUSER}: The SVD algorithm which has the same gradients as \textbf{SVD}, but uses incomplete incremental learning which updates the feature value after scanning all scores of a user instead of each single score.

\textbf{CSVD}: Algorithm 4, the compound SVD algorithm, which incorporates per-user and per-object biases and the constraints on feature vectors.

For each data set, the matrices \( M, U \) in \textbf{SVDNR}, \textbf{SVD} and \textbf{SVDUSER} as well as \( M, Y \) in \textbf{CSVD} are initialized by (2.22). The values of constraint matrix \( W \) and biases \( \alpha, \beta \) in \textbf{CSVD} are all initialized as 0. About the parameters, the learning rate \( \mu_v \) in \textbf{SVDNR} and \textbf{SVD} is 0.003 for \textit{Movielens} data set and 0.001 for \textit{Netflix} data
Table 4.1: Best RMSE achieved by each algorithm

<table>
<thead>
<tr>
<th>Dataset</th>
<th>AVGB</th>
<th>SVDNR</th>
<th>SVD</th>
<th>CSVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movielens</td>
<td>0.9313</td>
<td>0.8796</td>
<td>0.8713</td>
<td>0.8706</td>
</tr>
<tr>
<td>Netflix</td>
<td>0.9879</td>
<td>0.9280</td>
<td>0.9229</td>
<td>0.9178</td>
</tr>
</tbody>
</table>

set, but SVDUSER uses various learning rates for comparison. The regularization coefficients $k_u, k_m$ are 0 in SVDNR, while SVD and SVDUSER use regularization coefficients leading to the best performances, which are both 0.05 for Movielens and 0.005 for Netflix. The algorithm CSVD has more parameters than other algorithms, but its learning rates $\mu_v$ and $\mu_w$ are set equal to $\mu_v$ in SVDNR and SVD, and its first two regularization coefficients $k_y, k_m$ are also equal to $k_u, k_m$ in SVD. The other regularization coefficient $k_w$ for the constraint matrix $W$ is 0.02 for Movielens and 0.001 for Netflix. For the biases in CSVD, the learning rate and regularization coefficient $(\mu_b, k_b)$ are (0.0005, 0.05) for Movielens and (0.0002, 0.005) for Netflix. All these SVD-like algorithms have the dimension $f = 10$.

Table 4.1 shows the best RMSEs achieved by algorithms AVGB, SVDNR, SVD, and CSVD. That is, the lowest test RMSEs given by those algorithms before overfitting. We show the RMSEs versus training time in Figure 4.1, including algorithm SVDUSER with different learning rates. SVDNR suffers from overfitting quickly as it does not have a regularization term, while SVD gives a better performance with the same learning rate. CSVD is the best algorithm for both data sets, but it does not outperform SVD too much in Movielens as its assumption on user features is not effective in a denser data set. The behavior of SVDUSER shows that incomplete incremental learning is a bad choice in optimization. It can still lead to similar performances if the same gradients are used, but a smaller learning rate and more training time are needed. If the learning rate is not small enough, the decrease of RMSE is even
unstable, so it is hard to decide the stopping condition of an algorithm. Batch learning is even worse than incomplete incremental learning. In our experiments, batch learning requires an extremely small learning rate to reach the same performance as complete incremental learning, and it even takes several days to complete the optimization for Netflix. In conclusion, complete incremental learning is better in the optimization of SVD algorithms for collaborative filtering, and the compound SVD algorithm (Algorithm 4) can further boost the performances of conventional SVD algorithms for a sparse data set like Netflix.

For the post-processing algorithms described in Chapter III, we apply them to the best algorithm in the above experiments. That is, the compound SVD algorithm CSVD. The algorithms compared are:

CSVD: The model (feature matrices and biases) obtained from the compound SVD
Table 4.2: RMSE for each post-processing algorithm

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CSVD</th>
<th>SHIFT</th>
<th>KNN</th>
<th>KRR</th>
<th>WAVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movielens</td>
<td>0.8706</td>
<td>0.8677</td>
<td>0.8644</td>
<td>0.8635</td>
<td>0.8621</td>
</tr>
<tr>
<td>Netflix</td>
<td>0.9178</td>
<td>0.9175</td>
<td>0.9139</td>
<td>0.9101</td>
<td>0.9097</td>
</tr>
</tbody>
</table>

algorithm which gives the lowest RMSE. No post-processing method is applied.

**SHIFT:** We update the per-object biases and residuals by (3.3) and (3.4). This approach modifies the model of CSVD and gives better performances in experiments, so all following post-processing methods use the residuals updated by this approach.

**KNN:** Algorithm 5, which applies $K$-nearest neighbors to the residuals.

**KRR:** Algorithm 6, which applies kernel ridge regression with the polynomial kernel (3.11) to the residuals.

**WAVG:** Algorithm 7, which uses the weighted average to predict the residuals.

The parameters of algorithms KNN, KRR, and WAVG are set as follows. The number of neighbors and regularization coefficient $(k, k_r)$ in KNN are (15, 10) for Movielens and (50, 10) for Netflix, respectively. Algorithms KRR and WAVG are different in formulation, but they have the same parameters: the power of polynomial kernel $p$ and the regularization coefficient $\lambda$. We set them to (25.0, 1.0) for Movielens and (20.0, 1.0) for Netflix. The parameters are able to give the best test result for each algorithm. Another parameter of KRR and WAVG is the threshold on $t$, which is the number of objects considered for each user in Algorithms 6 and 7. A large value of $t$ leads to a better performance but requires more computational time. In order to show the best performances of algorithms, we do not use a threshold to restrict the maximal number of $t$.

The resulting RMSEs of these post-processing algorithms for Movielens and Netflix data sets are shown in Table 4.2. We can see that SHIFT slightly decreases the
RMSEs by updating the biases, and all three algorithms KNN, KRR and WAVG have significant improvements in both data sets. The weighted average algorithm, WAVG, is considered as the best post-processing method on the residuals of SVD-like algorithms. It is also efficient as it takes only several minutes to process the Netflix data set. In contrast, the kernel ridge regression algorithm KRR requires hours of computational time for the same data set.

Next we describe our best submission so far to the Netflix competition. In general, an SVD algorithm with larger dimension $f$ gives a better performance, but the time and memory requirements are also proportional to $f$. Consider to the resources we have, we use Algorithm 4 (CSVD) with dimension $f = 256$ to train a compound SVD model. The training data are the whole 100,480,507 scores in Netflix data set, and the parameters as well as the number of iterations are decided by the experiments with a training/validation split. The per-object biases in the model are then updated as SHIFT, and Algorithm 7 (WAVG) is applied for post-processing. The baseline approach provided by Netflix gives an RMSE of 0.9514, and ordinary SVD algorithms usually give results in the range of 0.91 to 0.93. Our procedure results in RMSE = 0.8868, which is 6.79% better than the baseline approach and is ranked the 35th rank at the time of submission.
CHAPTER V

Conclusions

Singular Value Decomposition is an efficient algorithm to predict the preferences of users for collaborative filtering. It uses gradient descent to optimize the feature values of users and objects, and has a time complexity linear in the number of training instances. We find that incremental learning, especially completely incremental learning which updates values after looking at a single training score, performs well in both accuracy and efficiency. The performance of algorithms can be improved by using per-user/object biases and imposing additional constraints on the feature values. As a combination, we propose a compound SVD algorithm which gives a significantly better performance with the same time complexity of traditional SVD algorithms.

In order to improve the accuracy of predictions, we apply several post-processing algorithms for estimating the residuals of scores from an SVD-like algorithm. That is, to predict the errors between the scores given by the algorithm and the ground-truth. Some different algorithms like K-nearest neighbors and regression are used, but all of them depend on the feature values obtained by the SVD algorithms. The best post-processing algorithm predicts the residuals by a weighted average, where the weights are polynomial to the dot products of normalized feature vector. For experiments we use two data sets Movielens and Netflix, which collect lots of scores for movie ratings. Our approaches show consistent results on these two data sets.


