Parallelization of Matrix Factorization for Recommender Systems

Julia Baum[1], Cynthia Cook[2], Michael Curtis[3], Joshua Edgerton[4], and Scott Rabidoux[5]
Graduate assistant: Andrew M. Raim[3] Faculty mentor: Nagaraj K. Neerchal[3]
Client: Robert M. Bell[6]

Worcester Polytechnic Institute[1] Catawba College[2] University of Maryland, Baltimore County[3],
Cornell University[4] Wake Forest University[5] AT&T Labs - Research[6]

Abstract

Recommender systems are emerging as important tools for improving customer satisfaction by mathematically predicting user preferences. Several major corporations including Amazon.com and Pandora use these types of systems to suggest additional options based on current or recent purchases. Netflix uses a recommender system to provide its customers with suggestions for movies that they may like, which are based on their previous ratings. In 2006, Netflix released a large data set to the public and offered one million dollars for significant improvements on their system. In 2009, BellKor's Pragmatic Chaos, a team of seven, won the prize by combining individual methods. Dr. Robert Bell, with whom we collaborated, was a member of the winning team and provided us with the data set used in this project, consisting of a sparse matrix with rows of users and columns of movies. The entries of the matrix are ratings given to movies by certain users. The objective is to obtain a model that predicts future ratings a user might give for a specific movie. This model is known as a collaborative filtering model, which encompasses the average movie rating (mu), the rating bias of the user (b), the overall popularity of a movie (a), and the interaction between user preferences (p) and movie characteristics (q). Two methods, Alternating Least Squares and Stochastic Gradient Descent, were used to estimate each parameter in this non-linear regression model. Each method fits characteristic vectors for movies and users from the existing data. The overall focus of this project is to explore the two methods, and to investigate the suitability of parallel computing utilizing the cluster Tara in the UMBC High Performance Computing Facility.

1 Introduction

Collaborative Filtering systems have had growing interest recently due to the Netflix million dollar challenge to improve its existing algorithm for recommending movies. Through the contest, Netflix hoped to enhance the method with which they recommend movies to their users. This method would be based on prior movies the users have rated. The goal of the contest was to predict, as accurately as possible, what any particular user would rate a movie. For instance, this could be interpreted as a standard regression problem; fitting a model which relates the characteristics of movies with those preferred by users to develop an estimated rating of the user. However, a closer look into the problem reveals that it is more challenging. The Netflix data does not explicitly contain any information about movie or user characteristics, except for ratings that users have given to movies. The ratings data are also sparse, i.e. some users have rated many movies, others hardly any, and no user has rated all of the movies, let alone a significant fraction. Collaborative filtering is a technique that makes use of the sparse data that we do have in order to give good estimates for the parameters of a regression model. The actual Netflix competition data is to the order of billions of entries and sparse, so modeling the problem is computationally expensive. Although the Netflix challenge has already been conquered, we believe that some of the iterative methods for Collaborative Filtering can be modified and parallelized to become significantly more efficient.

We present two popular Collaborative Filtering algorithms used to solve the Netflix problem: alternating least squares (ALS) and stochastic gradient descent (SGD). Our eventual goal is to parallelize these algorithms so that they can take advantage of a high performance computing cluster, but first we take a close look at the methods conceptually. In this report we focus on producing correct, efficient serial versions of the algorithms, and finding opportunities to parallelize them in subsequent work. The two algorithms themselves are also compared, in terms of their serial performance.

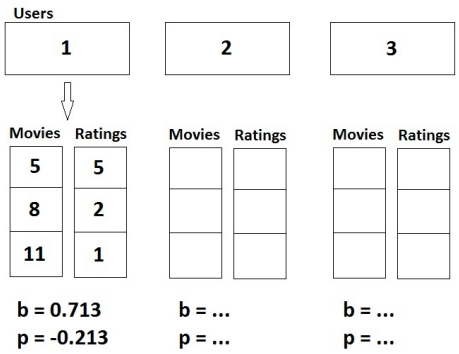
In section 2 we describe the Netflix data, and a smaller testing dataset which we used to develop our implementation. We also describe the data structures we used to manipulate the data in code. We also

describe the computing cluster available at UMBC which we used to run our experiments. In section 3 we describe the regression model for users' movie preferences. In section 4 we describe the algorithms we developed and in section 5, we discuss our conclusions and opportunities for future work.

2 The Data Organization and Hardware

For the million-dollar competition, Netflix provided participants with two main datasets: a training data set and a test data set. The training data set is used to fit the model's parameters, while the test data set is used to evaluate the quality of our estimates. The split between these two sets is used to help prevent overfitting of our parameters. Dr. Bell provided our team with two original data sets to begin development and testing of the models. Each data set contains a list of movie ratings, each grouped with a corresponding user ID and movie ID. These data sets are similar to the data sets used in the actual competition but are significantly smaller to allow for ease of testing. The training data set used in the competition contains around 99 million entries, whereas our training set for testing consists of about 180,000 entries. Our test data set is a subset of the training data and holds 20,000 entries. We have also used various sample sizes of the real data as well as generated data. The smaller size allows us to more quickly run through iterations of data, while also providing a backdrop to double test for overfitting of the parameters.

One of the most significant challenges for the project was to organize the data in a efficient manner that would provide quick setup, ease of access, and match users with movies and movies with users, without repeating entries. We also needed to be able to associate particular parameters for each user and movie, and be able to continuously update them. Initially we programmed in C, but we switched to R and C++ to have more memory management options. While C++ used object-oriented programming methods, our R implementation used mostly matrices but was not set up to handle sparse matrices well. As a result, main testing was done in C++. C++ used a User and Item class which included a vector of its ratings, its attribute vector, its bias. Each item and user class also contained a map data structure of ID and item/user pairs, as well as a vector of the IDs of the user that rated it or item that was rated by it. For example, say User 1 has rated Movies 5, 8, and 11, and has given them ratings of 5, 2, and 1 respectively. In our code, the data would be organized as follows:



Note that the values for b and p have been fabricated and only serve the purpose of explaining the data structure.

Within the High Performance Computing Center at UMBC, we have available a system called Tara to run our parallelized code. Completed and installed in 2009, Tara is a 86-node distributed-memory cluster with two quad-core Intel Nehalem processors: one front end node, one management node, two development nodes, and 82 compute nodes. We communicate directly with the front end node, which then sends our task to available processors on the compute nodes. Each node has 24 GB of memory, an InfiniBand interconnect, and 160 terabytes of central storage [2].

3 The Model

The model is a variation of a linear regression model using several parameters to predict each users rating for any particular movie, $\hat{r}_{ui}$ that is close to $r_{u,i}$. r_{ui} comes from the R matrix of ratings, where user u gives a rating to movie i for u in $\{1,\ldots,U\}$ and i in $\{1,\ldots,I\}$. Each given entry of the matrix is a floating point number. The aim of the model is to have the smallest mean squared error between r and $\hat{r}$. The full model comes from Bell and Koren's 2009 paper [1] on their prize-winning methods for the Netflix Problem. The original base model is a latent factor model breaking the large matrix of ratings into three smaller ones, one of users, one of movies, and one of averages. The parameter a_i is the difference between a movie's rating and the average movie rating, for $i = 1,\ldots,I$. Similarly, b_u is the difference between a user's rating and the average user's rating for a particular movie for $u = 1,\ldots,U$. The global average rating of all movies by all users is μ. A more advanced model takes into account the characteristic vectors p_u and q_i each of length d. Each element of p_u gives a rating of user's affinity for a certain characteristic, and the corresponding element in q_i quantifies the amount of this characteristic in the movie. Consider a simple case scenario where a recommender system is to determine whether a given user u_1 will like movie i_1. The user has rated several movies in the past generating a characteristic vector p_1, where $d = 3$. Suppose the vector measures comedy, action, and horror on a scale from -5 to 5. If u_1 likes comedy, loves action, but hates horror, his characteristic vector may be in the form $p_1 = (3.897, 4.875, -2.456)$. The characteristic vector for i_1 is also of size $d = 3$ and is measured on comedy, action, and horror; suppose it is $q_1 = (4.132, 3.978, -4.978)$. By taking the dot product of the two characteristic vectors, the recommender system can measure if the movie would be a good fit. In our example, $p_1' q_1 = 47.61$, which means that i_1 may be a good movie to recommend to u_1. Note that it is up to the recommender system to determine the d characteristics through the data, and that they may or may not have a simple interpretation as in this example. The full model cited in [1] is as follows:

$$\hat{r}_{ui} = \mu + a_i + b_u + p_u' q_i \tag{3.1}$$

The model appears fairly straightforward, but fitting it requires the special algorithms discussed later. Our first task is to find an expression to minimize, with respect to the unknown parameters, which will result in reasonable parameter estimates. Let

$$\kappa = \{(u,i) : \text{there is a rating } r_{ui} \text{ in the dataset}\},$$

for whichever dataset is being considered for training. To fit the parameters of the model (3.1) we define the objective function as

$$\sum_{(u,i)\in\kappa} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(\sum_{u=1}^{U} \|p_u\|^2 + \sum_{i=1}^{I} \|q_i\|^2 + \sum_{u=1}^{U} b_u^2 + \sum_{i=1}^{I} a_i^2 \right). \tag{3.2}$$

The left side of the objective represents the sum-squared error between the true ratings and the fitted ones. The second half of the objective with coefficient λ, penalizes parameters with large magnitude.

4 Methods of Fitting the Model

The basis of developing a robust and accurate model involves a matrix factorization breaking apart our large sparse matrix of the user ratings, the r_{ui}, into two or three smaller more manageable matrices while having the least error and keeping the number of parameters used reasonable to avoid the problem of over fitting the data.

We can see that (3.1) and (3.2) lead to something a little bit beyond linear regression problem. On one hand, the model in (3.1) is not a linear function of unknown parameters. Also, the objective function is not a simple sum of squared errors, but also includes a penalty component. A further complication is that we have a minute fraction of the possible user/movie combinations available in the dataset, perhaps on the order of one percent. Hence we now describe two techniques for matrix factorization, along with modifications.

Alternating Least Squares

The Alternating Least Squares Method (ALS) is a learning algorithm that finds a line to best fit given data by estimating unknown parameters for a collaborative filtering model of the form

$$y = X\beta + \epsilon. \tag{4.1}$$

In particular, the ALS method is used with non-linear regression models when solving for two dependent variables. The method fixes one of the dependent parameters, while solving for the other, creating a linear least squares problem. The method alternates solving for one parameter than the other. The least squares method is to obtain the best linear unbiased $\hat{\beta}$ such that:

$$\hat{\beta} = \arg\min_{\beta} ||y - X\beta||^2 \tag{4.2}$$

where the solution minimizes the sum of squared residuals [4]. The formulas for linear least squares fitting were independently derived by Gauss and Legendre [7]. In order to solve for an unknown parameter, the Gauss-Markov Theorem (GMT) states that to obtain the best linear unbiased estimate of β such that equation (4.1) holds true, we can solve for the estimate using the equation [6]:

$$\hat{\beta} = (X'X + \lambda I)^{-1} X'y.$$

Our collaborative filtering model is in the form

$$r_{ui} = \mu + a_i + b_u + p_u q_i,$$

where y is the rating; β is the parameter we are solving for; X is the fixed parameter; and $\epsilon = \mu + a_i + b_u$ [5]. The equation estimates the rating r_{ui} that user u will give movie i. Note that the model is linear in μ, a_i, b_u, and p_u for a fixed q_i. Thus, for a fixed q_i the best linear unbiased estimate of μ, a_i, b_u, and p_u can be determined by the GMT. Similarly, the model is linear in μ, a_i, b_u, and q_i for a fixed p_u. Thus, for a fixed p_u the best linear unbiased estimate of μ, a_i, b_u, and q_i can be determined by the GMT. The ALS method is an iterative algorithm that alternates between solving for p_u and q_i and updates the parameters until MSE converges within a given tolerance or the iteration count reaches the maximum iterations. We introduce some notation needed for the ALS algorithm:

- U = total number of users
- I= total number of movies
- K = total number of ratings
- N_u = number of ratings given by user u
- M_i = number of ratings given by item i

We begin the ALS algorithm by filling the characteristic vectors q_i with randomly generated real numbers ranging from -10 to 10. The parameters μ, a_i, b_u, and y_{ui} are initialized by the following equations using the ratings given in the test data set

$$\hat{\mu} = \frac{1}{K} \sum_{U} \sum_{I} r_{ui}$$

$$\hat{a}_i = \frac{1}{M_i} \sum_{u \in M_i} r_{ui} - \hat{\mu}$$

$$\hat{b}_u = \frac{1}{N_u} \sum_{i \in N_u} r_{ui} - \hat{\mu} - a_i$$

$$y_{ui} = r_{ui} - \hat{\mu} - \hat{a_i} - \hat{b_u}.$$

As stated above, the variable y_{ui} is created by subtracting a, b, and μ from the ratings. This is done as an attempt to adjust these parameters out of the data in order to simplify our set of regression equations. We then iterate between fixing q_i's and solving for p_u's, and fixing p_u's and solving for q_i's. In regression model (4.1), the X variable is the known parameter. For the ALS method, the known parameters depend on which alteration the algorithm is on.

First suppose we are solving for the p_u vectors. Then the q_i vectors are treated as known and fixed using the current estimates. The regression equation (4.1) becomes

$$y_{ui} = X_q \beta_p + \epsilon$$

where

$$X_q \beta_p = \begin{bmatrix} q_1' & 0 & 0 & \cdots & 0 \\ q_2' & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ q_I' & 0 & 0 & 0 & 0 \\ 0 & q_1' & 0 & \cdots & 0 \\ 0 & q_2' & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & q_I' & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & q_1' \\ 0 & 0 & 0 & \cdots & q_2' \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & q_I' \end{bmatrix} \begin{bmatrix} p_1 \\ \vdots \\ p_U \end{bmatrix} = \begin{bmatrix} q_1' p_1 \\ \vdots \\ q_I' p_1 \\ \vdots \\ q_1' p_U \\ \vdots \\ q_I' p_U \end{bmatrix},$$

where X_q is a $UI \times U$ matrix of $1 \times d$ vectors, and β_p is a $U \times 1$ vector of $d \times 1$ vectors.

Similarly, now suppose we are solving for the q_i vectors. Then the p_u vectors are treated as known and fixed using the current estimates, and the regression equation (4.1) becomes

$$y_{ui} = X_p \beta_q + \epsilon$$

where

$$X_p \beta_q = \begin{bmatrix} p_1' & 0 & 0 & \cdots & 0 \\ p_2' & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ p_U' & 0 & 0 & 0 & 0 \\ 0 & p_1' & 0 & \cdots & 0 \\ 0 & p_2' & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & p_U' & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & p_1' \\ 0 & 0 & 0 & \cdots & p_2' \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & p_U' \end{bmatrix} \begin{bmatrix} q_1 \\ \vdots \\ q_I \end{bmatrix} = \begin{bmatrix} p_1' q_1 \\ \vdots \\ p_U' q_1 \\ \vdots \\ p_1' q_I \\ \vdots \\ p_U' q_I \end{bmatrix},$$

where X_p is a $UI \times I$ matrix of $1 \times d$ vectors, and β_q is a $I \times 1$ vector of $d \times 1$ vectors.

The organization of each matrix is not arbitrary. Grouping the users together in the X_p vector and the movies in the X_q vector allows for easy parallelization. When using the GMT to solve for either p_u or q_i, we are able to separate the columns of the matrices X_q and X_p into vectors of length I and U respectively and

disregard the 0's. By doing this, we will have $2U \times I$ collaborative filtering equations:

$$\begin{bmatrix} y_{11} \\ y_{12} \\ \vdots \\ y_{1I} \end{bmatrix} = \begin{bmatrix} q_1^{'} \\ q_2^{'} \\ \vdots \\ q_I^{'} \end{bmatrix} p_1, \qquad \begin{bmatrix} y_{21} \\ y_{22} \\ \vdots \\ y_{2I} \end{bmatrix} = \begin{bmatrix} q_1^{'} \\ q_2^{'} \\ \vdots \\ q_I^{'} \end{bmatrix} p_2, \qquad \cdots, \qquad \begin{bmatrix} y_{U1} \\ y_{U2} \\ \vdots \\ y_{UI} \end{bmatrix} = \begin{bmatrix} q_1^{'} \\ q_2^{'} \\ \vdots \\ q_I^{'} \end{bmatrix} p_U.$$

When p_u is fixed, the separated Collaborative Filtering equations are

$$\begin{bmatrix} y_{11} \\ y_{21} \\ \vdots \\ y_{U1} \end{bmatrix} = \begin{bmatrix} p_1^{'} \\ p_2^{'} \\ \vdots \\ p_U^{'} \end{bmatrix} q_1, \qquad \begin{bmatrix} y_{12} \\ y_{22} \\ \vdots \\ y_{U2} \end{bmatrix} = \begin{bmatrix} p_1^{'} \\ p_2^{'} \\ \vdots \\ p_U^{'} \end{bmatrix} q_2, \qquad \cdots, \qquad \begin{bmatrix} y_{1I} \\ y_{2I} \\ \vdots \\ y_{UI} \end{bmatrix} = \begin{bmatrix} p_1^{'} \\ p_2^{'} \\ \vdots \\ p_U^{'} \end{bmatrix} q_I.$$

We solve for each p_u and q_i with the GMT. To solve for p_u we use the equation

$$p_u = (X'_{q_u} X_{q_u} + \lambda I)^{-1} X'_{q_u} r_{ui}.$$

Similarly, to solve for q_i, we use

$$q_i = (X'_{p_i} X_{p_i} + \lambda I)^{-1} X'_{p_i} r_{ui}.$$

After solving p_u for all u, we can use the estimates to solve for q_i and continue alternating. By rescaling both the p_u and q_i vectors, we are able to increase the stability of our converging parameters, by not allowing them to drift off to very large or very small numbers. For $r_{ui} = p'_u q_i$, we multiply all the p_u's by a constant c and divide all the q_i's by the same c. By doing so, all the r_{ui}'s remain unchanged. Given current p_u's and q_i's, it is best to solve for a value of c that minimizes $||P||^2 + ||Q||^2$, which turns out to be the c that makes $||P||^2 = ||Q||^2$. We find c for each dimension d to be:

$$c = \left(\frac{||q||}{||p||} \right)^{\frac{1}{4}}$$

We rescale the p_u and q_i vectors after every pair of alternating regressions, along with updating the parameters μ, a_i, b_u, and y_{ui} using the following equations:

$$\mu = \frac{1}{K} \sum_{i}^{I} \sum_{u}^{U} r_{ui} - a_i - b_u - p_u q_i$$

$$a_i = \frac{1}{M_i} \sum_{u}^{U} r_{ui} - \mu - b_u - p_u q_i$$

$$b_u = \frac{1}{N_u} \sum_{i}^{I} r_{ui} - \mu - a_i - p_u q_i$$

$$y_{ui} = r_{ui} - \mu - a_i - b_u.$$

The MSE is also calculated after each iteration using the equation:

$$\text{MSE} = \frac{\sum (r_{ui} - \mu - a_i - b_u - p_u q_i)^2}{K}.$$

Our algorithm continues to iterate solving for p_u and q_i on the training data set until each parameter converges or the tolerance and/or iteration maximum is met. We then take the best λ's and compute the MSE for the test data set.

We analyze the MSE and watch how the parameters are converging with each iteration. We began by testing the algorithm when $d = 0$ and $\lambda = 0$. With these input values, the problem reduces to its simplest form since the p_u's and q_i's drop out, and the algorithm should converge in one step. We continued by varying λ on the training data to minimizes the MSE on the test data.

Algorithm 1 : ALS

Set up vectors yUsers, yItems, xq, and xp
Initialize μ, a_i, b_u, y_{ui}
  while MSE has not converged and current iteration ¡ max iteration limit
    Create q_i vectors
     Solve for p_u vectors
    Create p_u vectors
     Solve for q_i vectors
    Rescale each p_u and q_i vector
    Update μ, a_i, b_u, y_{ui}
    Find MSE
  End

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a variation of the standard batch gradient for unconstrained optimization problems. In the standard batch gradient descent, the gradient is calculated over all observed r_{ui} with respect to the unknown parameters, at some current point in the space. The gradient gives the direction to adjust the parameters for the next step. For large-scale problems such as the Netflix problem, calculating the gradient is computationally expensive. The stochastic gradient descent algorithm approximates the gradient by taking the gradient at a single observation [3]. The idea is that although more iterations will be performed since we are not moving exactly in the direction of the true gradient. However the time, number of iterations, and all of the updates saved from not computing the full gradient over all of the observations may be more efficient and the algorithm may converge faster. The algebra for the SGD is derived from our objective function (note that we are using a simplified model here)

$$\sum_{(u,i)\in\kappa} (r_{ui} - \mu - a_i - b_u)^2 + \lambda \left(\sum_{u}^{U} b_u^2 + \sum_{i}^{I} a_i^2 \right). \tag{4.3}$$

To minimize this equation, we use SGD described above rather than the standard gradient descent method. Using an approximation of the gradient by taking the derivative of our parameters, each parameter will be updated by the gradient of the formula inside of the summation. The derivative is taken with respect to any particular parameter, either a specific b_v from the vector b_u or a a_l from the vector a_i. Let N_u be the set of ratings given by user u and M_i be the set of ratings for a given item i. This notation is helpful for bringing (4.3) under one summation, to aid in the derivation of SGD,

$$\begin{aligned}
&\sum_{u=1}^{U} \sum_{i\in N_u} (r_{ui} - \mu - a_i - b_u)^2 + \lambda \left(\sum_{u=1}^{U} b_u^2 + \sum_{i=1}^{I} a_i^2 \right) \\
&= \sum_{u=1}^{U} \sum_{i\in N_u} (r_{ui} - \mu - a_i - b_u)^2 + \lambda \left(\sum_{u=1}^{U} \sum_{i\in N_u} \frac{b_u^2}{|N_u|} + \sum_{u=1}^{U} \sum_{i\in N_u} \frac{a_i^2}{|M_i|} \right) \\
&= \sum_{u=1}^{U} \sum_{i\in N_u} \left[(r_{ui} - \mu - a_i - b_u)^2 + \frac{\lambda}{|N_u|} b_u^2 + \frac{\lambda}{|M_i|} a_i^2 \right] \\
&= \sum_{(u,i)\in\kappa} \left[(r_{ui} - \mu - a_i - b_u)^2 + \frac{\lambda}{|N_u|} b_u^2 + \frac{\lambda}{|M_i|} a_i^2 \right].
\end{aligned}$$

We can proceed with the stochastic gradient approximation for the update equations by taking the derivative of a single observation, call it r_{vl}. We want to minimize the original equation with the two variables pertaining

to our particular r_{vl}: a_l and b_v, and hence solve the equation

$$0 = \frac{\partial}{\partial a_l}\left[\sum_{i=1}^{I}\sum_{u\in M_l}(r_{ul}-\mu-a_l-b_v)^2+\frac{\lambda}{|N_u|}b_u^2+\frac{\lambda}{|M_i|}a_l^2\right]$$

Since we want to take the derivative with respect to a particular a_l, we can focus on only the observations that have an a_l. Recall that $\frac{d}{dx}\|x\|^2 = 2x$.

$$\begin{aligned}
&= \frac{\partial}{\partial a_l}\left[\sum_{u\in M_l}(r_{u,l}-\mu-a_l-b_u)^2+\frac{\lambda}{|N_u|}b_u^2+\frac{\lambda}{|M_i|}a_l^2\right]\\
&= \sum_{u\in M_l} 2\left(r_{u,l}-\mu-a_l-b_u\right)(-1)+2\frac{\lambda}{|M_l|}a_l\\
&= \sum_{u\in M_l} -2\left(r_{u,l}-\hat{r}_{u,l}\right)+2\frac{\lambda}{|M_l|}a_l\\
&= 2\sum_{u\in M_l}\left(\frac{\lambda}{|M_i|}a_l-e_{u,l}\right), \qquad \text{where } e_{u,l}=r_{u,l}-\hat{r}_{u,l}.
\end{aligned}$$

With stochastic gradient descent, we split up the summation above and consider each observation r_{vl} at a time. For the parameter a_l, this gives the iterative update formula

$$a_l^* = a_l - \gamma_l\left(\frac{\lambda}{|M_l|}a_l - e_{v,l}\right).$$

And following similar algebra, the updates of the b_v parameter are given by

$$b_v^* = b_v - \gamma_v\left(\frac{\lambda}{|N_v|}b_v - e_{v,l}\right).$$

The γ parameters arise from incorporating the constant terms being multiplied to each expression. They act as a step size, controlling how quickly the parameters can move. Larger γ values will allow larger steps to be taken, while smaller values are more appropriate when the given parameter is close to converging. We can fine tune the γ's by setting them differently for different parameters, or allowing them to adapt as the algorithm progresses. Currently however, we use a single γ for all parameters and leave it fixed for the duration of the algorithm.

We began testing this base model on the training and test sets. We work within the gamma and lambda parameter space to fit our parameters (a,b) and test the model on the test set to find a smallest MSE. Although we subscript the gammas in our update equation to have a unique gamma for each parameter; for our tests, we worked with a fixed gamma that we found through testing to be best suited. We explored many different parameters through numerous tests, searching to find a good MSE. For the simple two parameter model we found that the lambda parameter was by far the more important parameter and a lambda of 10 resulted in a low MSE. We tried a wide variety of values larger and smaller to explore any anomalies. Although the lambda's value was critical, a much more interesting question was adjusting gamma. Gamma was critical in how quickly our algorithm converged. We had to find a good trade off between a low MSE and the number iterations taken to find that MSE. For example, we also looked at a gamma value of .001. However, the iterations taken to put the error within our specified tolerance with fixed lambda of 10 was 2397 compared to 683 with a gamma of .005. These both resulted of MSEs within one thousandth of each other. Our gamma and lambda parameters are very sensitive! The window of 'good' parameters is small, and a too large of a gamma blows up the error. Besides looking at MSE, we would also consider different methods to add a learning rate, simply a dynamic gamma variable adjusting itself each pass. Our experiments with this provided interesting results, but were probably too specific to the individual data set to warrant serious recognition.

We start with the simplest model in C for testing, using just the a_i and b_u to fit the model. The general outline of the model is simple. Starting with initialized values, we run through the data set and compute the errors at each pass and update the parameters. We continue doing these until the stopping criterion is met, where our MSE does not improve enough by a certain threshold.

The algorithm itself works well with few problems. In the simple two parameter model solely with the a's and b's, the number of iterations the algorithm requires till convergence is immense. However, as we progressed from testing with two parameters and gradually tried to incorporate the ps and qs into the algorithm, the runtime and iteration count significantly decreased. We saw the iterations drop as low as 20 in some cases. We explored our gamma and lambda parameters in the search for the best combination, a pair that resulted in a low MSE when cross tested on the test set but yet also provided a low number of iterations in fitting all of the parameters (a's, b's p's q's).

After thorough testing of the base model, we move into the bigger model with the vectors p_u and q_i and work within the gamma, lambda, and d parameter space where d = the length of p and q, for finding optimal MSE over our data. With similar algebra as shown above for any a_l and b_v, the vectors are updated similarly in the model.

$$q_l^* = q_l - \gamma_l \left(e_{v,l}\, p_v - \frac{\lambda}{M_l} q_l \right)$$

$$p_v^* = p_v - \gamma_v \left(e_{v,l}\, q_l - \frac{\lambda}{N_v} p_v \right)$$

Algorithm 2 : SGD

Initialize Parameters
while MSE has not converged and current iteration ¡ max iteration limit
    For a user u in the list of users
        For a movie i corresponding to the particular user u
          Compute the $\hat{r}_{ui}$
          Compute the error of the specific observation
          Update parameters a_i, b_u, p_u, & q_i
          Update the sum squared errors;
        End
    End
Shuffle Data Set
End

5 Conclusions

Through close work with Dr. Bell, a member of the winning team and the guidance of our mentors, we accomplished our objectives. Since we built our model from the basics, and did all our coding, we have gained a firm understanding of the underlying framework. On the smaller data sets we worked with, the ALS algorithm was superior in terms of our metrics, namely, accuracy with respect to time, number or iterations. However, with very large data, the matrix algebra in ALS might become computationally expensive and make SGD may become more appealing. With both models, we did fully explore adaptive learning rate and parametric gamma and lambda values. Preliminary ideas, showed that even the simplest implementations decreased the number of iterations significantly. Further testing would allow us a better means of comparing the two methods, to more thoroughly evaluate the models strengths and weaknesses, and to have a backdrop to compare our results not only to each other, but to actual results from past participants of the Netflix Competition. For example, although both methods included a gamma parameter, the relationship between

the two gammas is unknown. For future work, we wish to explore several modifications of the ALS and SGD algorithms such as parallel implementations, piecewise algorithms for huge data sets and parallel data input. The actual parallelized serial algorithm for SGD is cumbersome and seems inefficient. However, with these modifications it would be more appealing and, as a result, warrant serious consideration and testing.

References

[1] R. Bell and Y. Koren, "Matrix Factorization Techniques for Recommender Systems," *IEEE International Conference on Data Mining Workshops*, 2009.

[2] `www.umbc.edu/hpcf`, 2010.

[3] `www.en.wikipedia.org/wiki/Stochastic_gradient_descent`, July 2010.

[4] Yehuda Koren, Robert Bell and Chris Volinsky, "Matrix Factorization Techniques for Recommender Systems," IEEE Vol. 0018-9162, pp.42-49, 2009.

[5] Robert Bell, "Matrix Factorization for Recommender Systems," presentation at UMBC, 2010.

[6] Jia Li, "Linear, Ridge Regression, and Principal Component Analysis," `www.stat.psu.edu/jiali`.

[7] Eric W. Weisstein, "Least Squares Fitting," From MathWorld–A Wolfram Web Resource. `www.mathworld.wolfram.com/LeastSquaresFitting.html`, 2010.