MATRIX PROJECT: RECOMMENDER SYSTEM

JADEN WANG (001, 109706844)
ALEXEY YERMAKOV (002, 109488187)

29 APRIL 2020

Abstract.

Matrix factorization was the method used by the team that won the Netflix Prize competition more than a decade ago. However, a formal mathematical description of this method is absent from winning team's paper and is hard to find elsewhere. More specifically, how is matrix factorization mathematically related to singular value decomposition (SVD)? Throughout this paper we explored, developed, and implemented linear algebra concepts by following in the footsteps of the winning team and others who tackled the problem of improving upon Netflix's recommender system. First, we delved into the mathematical formulas and derivations needed to understand the method of matrix factorization via gradient descent. We walked through various proofs and analyzed the relationship between matrix factorization and SVD before formally deriving the equations for gradient descent. Then, we provided and explained the algorithm and parameters of our implementation. After that, we presented our result: we achieved a root-mean-square-error of 0.9301 and beat Neflix's own system by 1.28%. Finally, we compared our result to the winning team, recounted challenges we encountered such as the trade-offs between minimizing root-mean-square-error and program run-time, and discussed future directions.

1. Introduction.

Recommender systems are ubiquitous in today's digital world. Streaming services recommend our favorite types of entertainment to keep us hooked to their platforms, online shopping services recommend quality products that we are more likely to spend money on, and social platforms recommend potential jobs, friends, or even romantic partners to us so we are connected with more people and opportunities. Regardless of the ethical merits of these modern phenomena, it is no doubt that recommender systems can generate massive profits for the businesses and play a powerful role in shaping our daily lives. Thus, it might be hard to imagine that just a little over a decade ago, we didn't quite know how to build an accurate recommender system.

The breakthrough only came after 2006 when Netflix announced a worldwide competition with a one-million dollar prize. The training dataset Netflix released was enormous: it contained sparse ratings of 17,770 movies from 480,189 users [4]. The goal of the competition was to construct a recommender system that reduced the root-mean-square-error (RMSE)[1], a measure of prediction accuracy used by Netflix, of movie rating predictions by more than 10% from that of Netflix's own system, which yielded an RMSE of 0.9474. Not surprisingly, mathematics played an essential role in constructing such a system. What was surprising was that the mathematics used by the winning team was nothing more than one linear algebra concept and some calculus [2].

How can linear algebra be applied to this problem? Readers might immediately recognized that the dataset provided by Netflix is just a sparse rectangular matrix with movie ID as the row index, user ID as the column index, and the corresponding ratings as entries of the matrix. To analyze such a rectangular matrix, a common but powerful method might come to mind: singular value decomposition, or SVD. For readers unfamiliar with this method, we have supplied a formal description of SVD in the next section. Intuitively, SVD extracts the most important components from a rectangular matrix and therefore preserves the greatest amount of information with the least amount of data. It is widely used in data compression and information extraction.

Still, it might not be immediately obvious to the perceptive readers why SVD is relevant to this project. This skepticism is well-founded since SVD is only useful if one has a complete matrix. Evidently, none of the users rated all of the 17,770 movies (hereafter referred to as *items*), so the dataset has lots of missing entries represented by 0. Yet SVD would incorrectly assume that 0 is the actual rating given by the users and yield the wrong matrices. Although we cannot directly use SVD to factor this matrix, SVD provides the theoretical justification that such decomposition would exist had the information been complete. The modified SVD used by the winning team aims to overcome missing data and approximate this theoretical decomposition.

Therefore, the goal of this paper is to dissect the mathematics described by the winning team, implement their matrix factorization algorithm, and present our results over the same Netflix dataset.

[1]RMSE is calculated by the square root of mean of the sum of all squared prediction errors.

2. Mathematical Formulation.

2.1. Singular Value Decomposition.

First, we would like to provide a formal description of singular value decomposition (SVD). The following theorems and definition were excerpts from [5]. We assume that the readers have sufficient knowledge about eigenvalues and eigenvectors and are comfortable with the following proposition:

Proposition 2.1. *Let $A = A^T$ be a $n \times n$ symmetrical matrix. Let $\mathbf{v}_1, \dots, \mathbf{v}_n$ be an orthogonal eigenvector basis such that $\mathbf{v}_1, \dots, \mathbf{v}_k$ correspond to non-null eigenvalues, while $\mathbf{v}_{k+1}, \dots, \mathbf{v}_n$ are null eigenvectors corresponding to the zero eigenvalue if it exists. Then $k = \operatorname{rank} A$; the non-null eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_k$ form an orthogonal basis for* $\operatorname{img} A = \operatorname{coimg} A$, *while the null eigenvectors $\mathbf{v}_{k+1}, \dots, \mathbf{v}_n$ form an orthogonal basis for* $\ker A = \operatorname{coker} A$.

We need to define singular values before discussing SVD.

Definition 2.2. *The singular values $\sigma_1 \dots \sigma_k$ of an $m \times n$ matrix A are the positive square roots, $\sigma_i = \sqrt{\lambda_i} > 0$, of the nonzero eigenvalues of the associated Gram matrix $K = A^T A$.*

This definition allows us to state the following theorem (proof available in Appendix A.1):

Theorem 2.3 (Singular Value Decomposition). *A nonzero real $m \times n$ matrix A of rank $k > 0$ can be factored as below:*

$$A = U \Sigma V^T$$

where U is an $m \times k$ matrix with orthonormal columns, Σ is a $k \times k$ diagonal matrix with the singular values of A as its diagonal entries, and V is a $n \times k$ matrix with orthonormal columns.

2.2. Relating SVD to the Neflix Competition.

The theory can also be understood intuitively. Suppose there are k orthogonal characteristics of any item that affect the ratings, such as the amount of action present, or the creativity of the plot, *etc.* Let's refer to these characteristics as *features.* Each item has different weights of those features, which can be represented by a vector $\mathbf{q}_i$ where i is the item index representing the ith unique item. Each user also has different preference for those features, which can be represented by a vector $\mathbf{p}_u$, where u is the user index representing the uth unique user. If we assume that users' ratings are predominantly influenced by their preference for the particular composition of those features present in the item, then the true rating of ith item by the uth user, r_{iu}, can be approximated by the Euclidean dot product between the two vectors, *i.e.*

$$r_{iu} \approx \langle \mathbf{q}_i, \mathbf{p}_u \rangle \tag{2.1}$$

where i, u denote the indices of the item and the user. Then we denote the hypothetical matrix with complete information on all of the r_{iu} as R, an item-by-user matrix. It follows that R would have an approximated factorization of

$$R \approx Q^T P \tag{2.2}$$

where P is a feature-by-user matrix with $\mathbf{p}_u$ as columns, and Q is a feature-by-item matrix with $\mathbf{q}_i$ as columns. Note that the jth row of Q and P representing entries associated with the jth feature are denoted as row vectors[2] $\mathbf{q}_j^T$ and $\mathbf{p}_j^T$, respectively.

However, the approximated factorization seems to be missing the diagonal matrix consisting of singular values. It turns out that the diagonal matrix can be extracted as below. Recall that since the matrices Q^T and P^T aren't normalized, we can express $Q^T = \tilde{Q}^T C$ and $P^T = \tilde{P^T} D$ where $\tilde{Q}^T, \tilde{P}^T$ have orthonormal columns representing distinct features and C, D are diagonal matrices with the scaling factors of those columns as entries. Let $U = \tilde{Q}^T$, $V = \tilde{P}^T$, and $\Sigma = CD$. Then equation (2.2) would morph from a quasi-SVD into a true SVD:

$$\begin{aligned} R &\approx Q^T P \\ &= \tilde{Q}^T C(\tilde{P}^T D)^T \\ &= \tilde{Q}^T C D \tilde{P} \\ &= U \Sigma V^T \end{aligned}$$

2.3. Gradient Descent. [3]

To compute the approximate rank-k factorization of a large sparse matrix, where k is the predetermined number of features, we can follow the gradient of the sum of squared error function and arrive at global minima numerically via gradient descent, popularized by Simon Funk as a factorization method for the Netflix data in 2006 [1]. Based on equation (2.1), we can establish the error term as

$$\mathcal{E}_{iu} = r_{iu} - \langle \mathbf{q}_i, \mathbf{p}_u \rangle \tag{2.3}$$

However, while training $\mathbf{q}_i$ and $\mathbf{p}_u$ iteratively by feature, we cannot use the entries associated with features that we haven't trained, since we want the current feature to be orthogonal to previous fully-trained features and only minimize the error unexplained by them. For training, let's modify the equation above as the following:

$$\mathcal{E}_{iu} = r_{iu} - \sum_{n=1}^{j} q_{ni} p_{nu} \tag{2.4}$$

where j is the rank of the feature that we are training at the moment and $1 \leq j \leq k$.

We use the sum of squared error function as our loss function because it is convex with well-defined gradient, which guarantees that gradient descent would reach the global minimum. After adding the L^2 regularization term, $\mathcal{R}_{iu} = \sum_{n=1}^{j} \left(q_{ni}^2 + p_{nu}^2\right)$, with parameter λ to penalize model complexity and prevent overfitting, we obtain the final formula to find $\mathbf{q}_j^T$ and $\mathbf{p}_j^T$ that minimize the loss function:

$$\underset{\mathbf{q}_j^T,\ \mathbf{p}_j^T}{\operatorname{argmin}} \sum_{i,u} (\mathcal{E}_{iu}^2 + \lambda \mathcal{R}_{iu}) \tag{2.5}$$

When training the jth feature, for each item-user pair (i, u) with existing rating, we aim to reduce $\mathcal{L}_{iu}(q_{ji}, p_{ju}) = \mathcal{E}_{iu}^2 + \lambda \mathcal{R}_{iu}$ by computing its gradient with respect

[2]In this paper, we use bold lower case letters to denote column vectors and the transpose of column vectors to denote row vectors.

[3]The paper named the method "stochastic gradient descent", but it was actually a simple gradient descent. Since each vector size equals to the number of features k, unmodified gradient descent is tractable under limited features and therefore preferred.

to q_{ji} and p_{ju}. With details described in Appendix A.2, we obtain

$$\nabla \mathcal{L}_{iu} = \begin{pmatrix} \frac{\partial \mathcal{L}_{iu}}{\partial q_{ji}} \\ \frac{\partial \mathcal{L}_{iu}}{\partial p_{ju}} \end{pmatrix} = \begin{pmatrix} -2\mathcal{E}_{iu} p_{ju} + 2\lambda q_{ji} \\ -2\mathcal{E}_{iu} q_{ji} + 2\lambda p_{ju} \end{pmatrix} \tag{2.6}$$

Where for each (i, u) pair and each training step γ, the regularized gradient descent updates the variables q_{ji} and p_{ju} as:

$$\begin{pmatrix} q^*_{ji} \\ p^*_{ju} \end{pmatrix} = \begin{pmatrix} q_{ji} - \gamma(-2\mathcal{E}_{iu} p_{ju} + 2\lambda q_{ji}) \\ p_{ju} - \gamma(-2\mathcal{E}_{iu} q_{ji} + 2\lambda p_{ju}) \end{pmatrix} \tag{2.7}$$

Notice how gradient descent eventually trains all $\mathbf{q}_i$ and $\mathbf{p}_u$ and thus allows us to predict rating of any item-user pair despite that most of such data is missing. Now, we are ready to translate math into algorithms and code.[4]

3. Examples and Numerical Results.

3.1. Algorithms.

The first aspect of performing gradient descent is processing the data. In the code below, it is assumed that the data was processed into a data structure from which a rating for a user/item ID combination can be extracted, and a chosen value that represents the value doesn't exist (for example, 0 or -1). From that data structure, item_f and user_f can be created, which are the two smaller matrices that result from gradient descent.

The algorithm for one step of gradient descent is as follows:

Algorithm 3.1 Gradient_Descent

```
Require: $1 \le uid \le USERS,\ 1 \le item \le ITEMS,\ 1 \le n \le FTRS$
 1: function GRADIENT_DESCENT(user_f[USERS][FTRS], item_f[ITEMS][FTRS], n, uid, item)
 2:     $dot\_product = 0$
 3:     for i = 1 to n do
 4:         $dot\_product \mathrel{+}= user\_f[uid][i] \cdot item\_f[item][i]$
 5:     end for
 6:     $err = rating[uid][item] - dot\_product$
 7:     $user\_tmp = user\_f[uid][n]$
 8:     $user\_f[uid][n] \mathrel{+}= LRATE \cdot (err \cdot item\_f[item][n] - K \cdot user\_f[uid][n])$
 9:     $item\_f[item][n] \mathrel{+}= LRATE \cdot (err \cdot user\_tmp - K \cdot item\_f[item][n])$
10: end function
```

[4]Furthermore, additional methods for model improvement are extensively used in the paper, such as correcting for user and item biases. These techniques could reduce the error further by accounting for the idiosyncratic nature of each user or movie that influences the ratings beyond features. But since these methods are outside the scope of linear algebra, we omitted the details here.

The function named `Gradient_Descent` performs one step of gradient descent for a single q_{ij} and p_{ju}, as described in equation (2.7). Here, the arrays `user_f` and `item_f` represent 2-dimensional matrices where the row index indicates the unique user or item respectively and the column index indicates the feature number (ranging from 1 to `FTRS` inclusive, where `FTRS` is the number of features present in a single user or item vector). These two 2-dimensional matrices can be seen as a column vectors where each entry in the column vector is a corresponding user/item vector of length `FTRS`. `LRATE` is equivalent to γ and `K` is equivalent to 2λ. `USERS` and `ITEMS` are the total number of users and items given by the data used for the gradient descent. `err` is the current error, and it is calculated by subtracting the existing rating for the user/item pair from the dot product of the user and item vectors for elements 1 to `n` inclusive (where `n` is the feature being trained).

`Gradient_Descent` saves the value `user_tmp` before updating `user_f[uid][n]` because for each step of gradient descent, the formula updates both `user_f[uid][n]` and `item_f[item][n]` simultaneously. Without the temporary variable, `item_f[item][n]` will use the updated value of `user_f[uid][n]` when training the feature.

The following algorithm presents itself as the complete training function. When this function exits, the entire set of user/item vectors are trained for `FTRS` features.

Algorithm 3.2 Feature Training

```
Require: $0 \leq FTRS,\ 0 \leq USERS,\ 0 \leq ITEMS,\ 0 \leq EPOCHS$
 1: function Train
 2:     for n from 1 to FTRS do
 3:         for i = 1 to EPOCHS do
 4:             for item = 1 to ITEMS do
 5:                 for uid = 1 to USERS do
 6:                     if RatingExists(uid, item) then
 7:                         Gradient_Descent(user_f, item_f, n, uid, item)
 8:                     end if
 9:                 end for
10:             end for
11:         end for
12:     end for
13: end function
```

The function `Train` has multiple loops: the outermost loop (line 2) shows that each feature is trained before moving onto the next, until all `FTRS` features are trained; the next loop (line 3) manages how many steps of gradient descent is performed per feature (in this case, `EPOCHS` is the number of steps of gradient descent per feature); the next two loops (line 4) and (line 5) go through all existing item/user pairs, and if a rating exists (by a call to `RatingExists`) for that pair, then one step of gradient descent is performed by calling `Gradient_Descent`.

3.2. Numerical considerations.

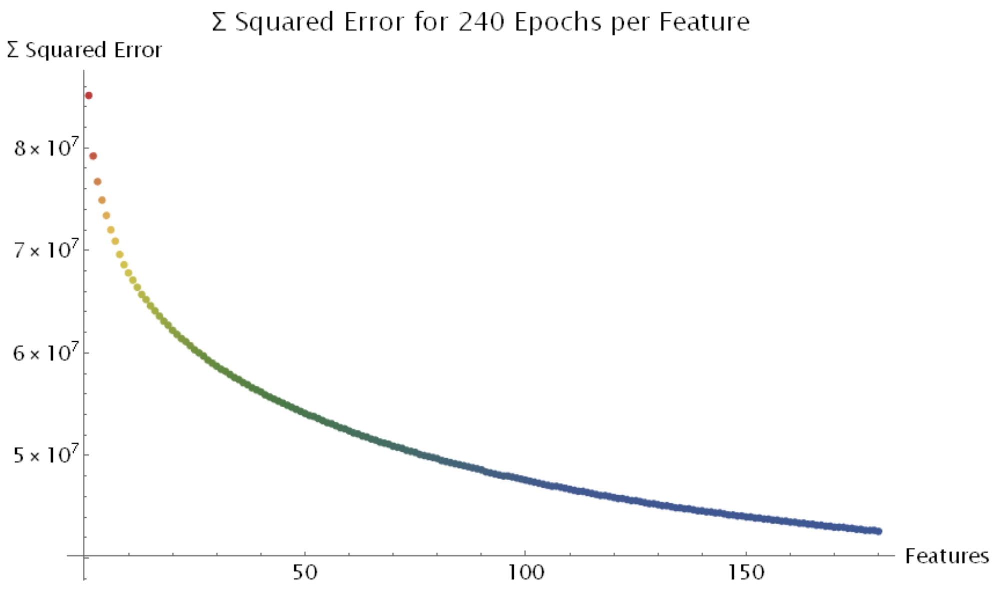

FIG. 3.1. Σ *Squared Error for 240 Epochs per Feature*

The graph above, titled Σ *Squared Error for 240 Epochs per Feature*, is the result of running the code created off the algorithms explained earlier. The code ran 240 epochs per feature for 180 features for the training dataset. After each feature was trained[5], the sum of the squared error was calculated by summing up $\mathcal{E}_{iu}$ described in equation (2.4) for all existing ratings. To this effect the elements of vectors $\mathbf{q}_i$ and $\mathbf{p}_u$ that are untrained were not used in calculating the squared error.

Although the total squared error may seem high, the context of these numbers must be taken into account. After the first feature was trained, the total squared error was 85,062,700. This same value means that the average squared error for each rating was approximately 0.8465 for the first feature (since 100,480,507 ratings were given by the data set). Comparing this to the total squared error after training the 180th feature - which was a total of 42,624,800 with an average of 0.4242 for each rating - it is clear that gradient descent did indeed minimize the total squared error. This conclusion is further supported by the figure above by displaying that each consecutive trained feature resulted in a decrease in the sum of squared error for the entire data set.

Another set of interesting data to discuss is elapsed time and applicability of this method.

[5]Trained is used here as a way of describing that gradient descent ran to completion before moving onto the next feature. In this case, trained means performing 240 steps of gradient descent.

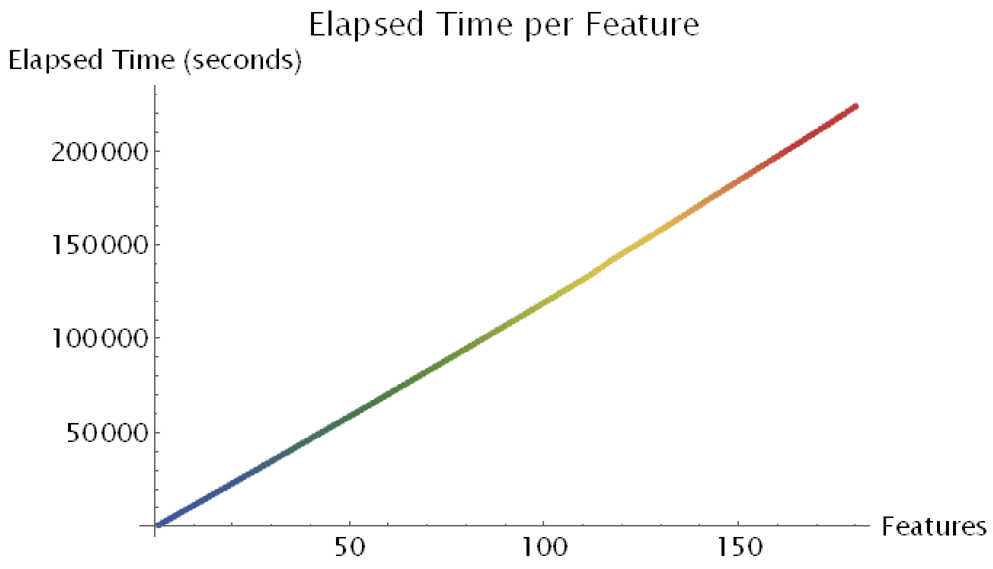

FIG. 3.2. *Elapsed Time for 240 Epochs per Feature*

This graph shows that the time necessary to run the code is linear. This makes sense, as approximately 2.75 seconds were needed to perform a single step of gradient descent after various code optimizations, and an additional 10 minutes were needed to save the data after each feature (just in case the program stopped execution before completion). The end result is that training one feature took about 20 minutes, resulting in a total run-time of the program of 62.13 hours (just over 2.5 days).

Total Squared Error			
Feature	120 Epochs	240 Epochs	480 Epochs
1	8.50854e+07	8.50627e+07	8.50584e+07
40	5.88484e+07	5.61716e+07	5.5025e+07

FIG. 3.3. Σ *Squared Error for Varying Epochs for Features 1 and 40*

Figures 3.3 to 3.6 show that if we increase the number of epochs exponentially, our reduction in total squared error is not exponential, especially if we use a smaller number of features. Regardless of how many epochs per feature we use, the time necessary to train a feature increases linearly with the amount of features, with the number of epochs per feature scaling the total amount of time needed to train a feature. Training 40 features took 1.7x the amount of time when using 240 epochs compared to 120 epochs and 3.1x the amount of time when using 480 epochs compared to 120 epochs (since the code saved the matrices after every feature, there wasn't a 2x increase in

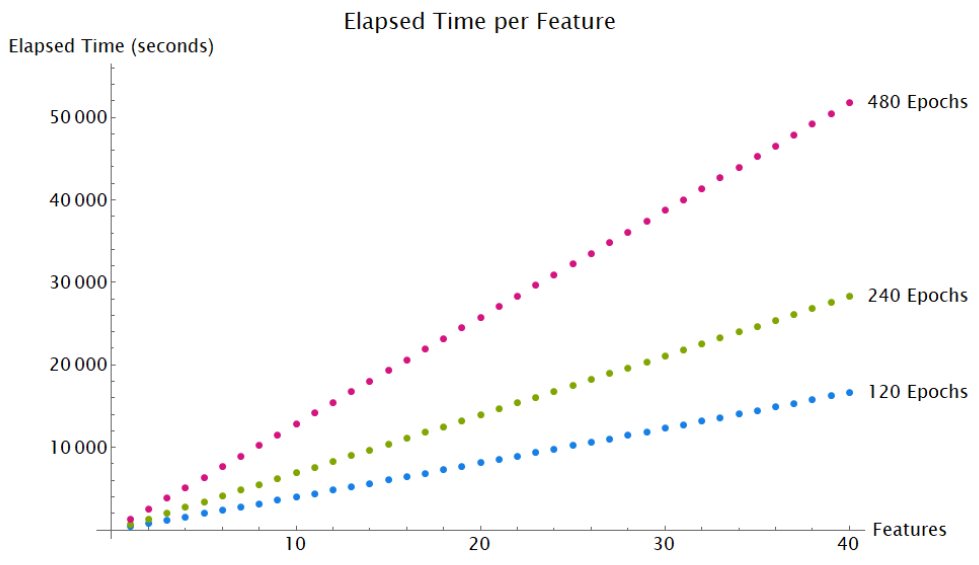

FIG. 3.4. *Elapsed Time for Varying Epochs per Feature*

Elapsed Run Time in Seconds			
Feature	120 Epochs	240 Epochs	480 Epochs
1	369.842	657.359	1244.78
40	16664.4	28343.9	51805.9

FIG. 3.5. *Elapsed Run Time for Varying Epochs for Features 1 and 40*

run time for a 2x increase in epochs); however, the total square error improved by 4.549% for 40 features from 120 epochs to 240 epochs and 6.497% from 120 epochs to 480 epochs. As a result, a 3.1x increase in run time does not translate in a 3.1x decrease in total squared error. This makes sense since gradient descent uses error as a step size, so the initial epochs will result in a larger decrease in total squared error than later epochs which have a smaller error, and as a result, a smaller step size.

3.3. Results.

Our implementation resulted in an RMSE of 0.9301 on the same probe data that Netflix and the winning team used to assess their performance. This is a 1.82% improvement over Netflix's own system.

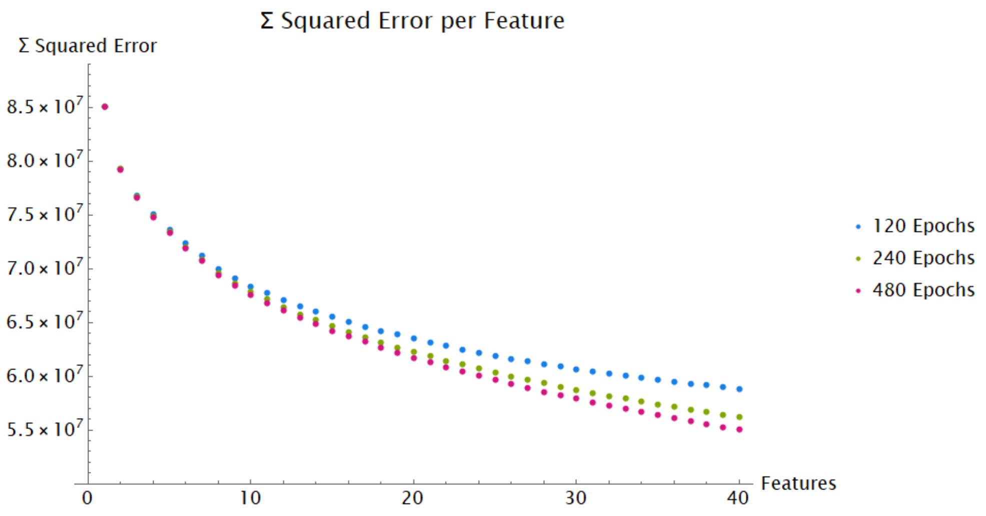

FIG. 3.6. Σ *Squared Error for Varying Epochs per Feature*

4. Discussion and Conclusions.

4.1. Discussion of results.

It was quite impressive that our recommender system achieved better performance than Netflix's 2006 system simply by implementing one linear algebra concept via gradient descent. Besides adding regularization to avoid overfitting, we did not use any modeling techniques to fine-tune the performance, such as considering user/item biases. This result demonstrated just how powerful the method of matrix factorization is.

On the other hand, the winning team achieved a 0.903 RMSE performance using 180 features and regularization only, beating us by a large margin despite using similar method. The main difference between our implementation and theirs is that we used gradient descent as the optimization technique and thus had to limit training to 240 epochs to achieve reasonable runtime on a personal desktop, sacrificing accuracy for speed; whereas the winning team used alternating least square that allows massive parallelization, which translated into more efficient error reduction and better accuracy that goes way beyond what 240 epochs can achieve. Moreover, since ratings cannot exceed the range between 1 and 5, during prediction we implemented *clipping* which clipped the score to the extrema if the score exceeds the extrema during each feature summation. This is an easy way to safeguard the fidelity of the results, but it might not be the most accurate way. The winning team didn't describe their implementation during prediction at all, but we would not be surprised if they used a more sophisticated technique that overcame spurious extreme ratings due to lack of data during training. Therefore, we see that our "bare-minimum" implementation still has room for improvement.

Overall, the lesson we learned is that although mathematical theories play a vital role in solving complicated real-world problems like recommender systems, the specific implementation of theories are highly non-trivial as well. Especially considering that a small gain in the performance of the system might lead to massive profits for the company, understanding the theories is just the first step to become a world-class problem-solver.

4.2. Learning Stretches.

There were multiple different obstacles and resulting learning stretches encountered during the project. The first of which was understanding the mathematical structure of gradient descent and how it relates to the quasi-SVD matrix factorization method used by the winning team. As a result, we all have a deeper understanding of matrix factorization and the real world applications of such structures. A particular skill we picked up from the project is how to formulate a real-world problem into a mathematical one. Furthermore, an understanding of the mathematical principles behind gradient descent gained during this project allows us to tackle more difficult tasks in the future, such as training neural networks. What we realized was that the process of reducing RMSE is a form of machine learning: we were given a training set of data and used that to make useful predictions (in this case, predicted ratings). This was our first time approximating a solution indirectly via minimizing a loss function as opposed to finding the solution analytically, and as a result we have gained a deep appreciation for numerical methods, especially how it easily overcame the missing data problem!

Another set of obstacles encountered was associated with writing the software. The project GitHub [6] (https://github.com/yyexela/MatrixProject) containing all of the relevant software required for the project has a combination of different languages associated with different tasks needed to fulfill the project's goals. We had to preprocess the Netflix data set using MATLAB and Python, import that data into a C++ data structure, perform gradient descent on the data, save the resulting matrices whilst also obtaining relevant run-time data, and finally compute the RMSE against the probe data.

One particularly interesting challenge was program efficiency. The first obstacle was converting one hundred million raw ratings and their user/movie IDs into a sparse matrix. It turned out that this was unnecessary since although a sparse matrix was the way to conceptualize the problem, during implementation we simply stored existing ratings and their user/item indices in a three-column compact dataframe and completely circumvented working with sparse matrix. The second obstacle was loading the Netflix data into C++ to perform gradient descent. Creating a 2D-Array was out of the question since (1) C++ didn't allow us to make a 17,770 x 480,189 array and (2) a 17,770 x 480,189 array would have over 98% of its entries wasted (since we are given only 100,480,507 ratings). So, as a result we implemented an array of vectors[6] in order to increase both memory and run time efficiency[7]. The third obstacle was computational efficiency. It would be very inefficient to calculate the dot product and magnitudes of a pair of vectors for every feature `EPOCHS` number of times. This was solved by creating an array of vectors that stored the predicted error minus the dot product of a pair of vectors and two arrays for the magnitudes of

[6]A vector in C++ is a dynamically sized array that grows in size only when elements are added to it.

[7]Run time efficiency would go up since we are only parsing through non-zero ratings, whereas if we had a 17770 x 480189 array we would be spending a lot of computing power moving through arrays with many empty values.

user and item vectors. All three arrays accounted only for the trained features. These arrays were updated after each trained feature once, since once a feature is trained we don't change its values in the arrays anymore. This allowed for the cumulative software run time to be linear with increasing amounts of trained features despite the mathematical dot product being larger. This large increase in run time efficiency comes at the marginal cost of memory efficiency.

Another software challenge was implementing the C++ library Boost. The library allowed us to save and load the user/item feature matrices whenever we wanted. We used this to save our data during program run time just in case the software crashed during the 62.13 hour run time and to only run the factorization software once and use the matrices for other applications, such as predicting user ratings. The code itself wasn't difficult to implement, but the installation of Boost and linking in the MakeFile[8] was strenuous for someone who has never explicitly used linking or a C++ library before.

4.3. Further Application.

As we mentioned earlier, we omitted the details of model improvement because they aren't relevant to linear algebra. But as the winning team demonstrated in their paper, adding user/item biases and temporal biases significantly improved prediction accuracy. An immediate improvement we can make is to include those variables into our loss function as well. Including more variables means that the run-time would be further increased. Hence we might want to consider implementing alternative least squares used by the winning team as the optimization technique.

The immediate logical next step is to implement neural network models and compare the results. Given that the matrix is sparse and that this quasi-SVD can only do so much to compensate for missing data, it would be interesting to see how effective neural networks are at filling in the gaps, which is a job neural networks are known to be particularly good at.

Finally, we are fully aware that we have just scratched the surface of recommender systems, since the only source of our data came from one type of explicit feedback: user ratings. In reality, these companies hold massive data of customers that are implicit feedback in nature, such as browsing behavior or characteristics of one's social group. A better model would use both explicit and implicit data to gain a full picture of customers' true preferences.

4.4. Conclusion.

Throughout this project we have formally described the method of quasi-SVD matrix factorization via gradient descent, successfully implemented it on the Netflix dataset, and shown that it works according to the mathematical principles behind it. The theoretical knowledge and the numerical implementation we learned through this project pave the way for us to understand information extraction from large datasets and the fundamentals of machine learning.

[8]The standard file used for compiling software with multiple components. In our case, compiling together multiple C++ source, header, and library files together into a single executable.

REFERENCES

[1] S. Funk, *Netflix update: Try this at home*. Personal Blogs, Dec. 2006, https://sifter.org/~simon/journal/20061211.html (accessed 2020-04-24).

[2] Y. Koren, R. Bell, and C. Volinsky, *Matrix factorization techniques for recommender systems*, Computer, 42 (2009), pp. 30–37, https://doi.org/10.1109/MC.2009.263.

[3] J. Meiss, *April 20th lecture note*, 2020.

[4] Netflix, *Netflix competition training dataset*, 2006, https://archive.org/download/nf_prize_dataset.tar (accessed 2020-04-28).

[5] P. Olver and C. Shakiban, *Applied Linear Algebra*, Springer, New York, NY, 2 ed., 2018, https://doi.org/10.1007/978-3-319-91041-3.

[6] A. Yermakov and J. Wang, *Matrix project github code*, 2020, https://github.com/yyexela/MatrixProject (accessed 2020-04-28).

Appendix A. Proofs and derivations.

A.1. Proof of Theorem 2.3 based on [5] and [3].

Proof. We would like to use the eigenvectors of the Gram matrix to form one of the matrix with orthonormal columns and show that we can construct the other.

Consider the associated Gram matrix $K = A^T A$. Since it is clearly symmetrical, Proposition 2.1 yields non-null orthonormal eigenvectors $v_1, \ldots, v_k$ of K, where $k = \operatorname{rank} K$ for $1 \leq i \leq k$. $A^T A\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Recall that singular values σ_i are defined as the square roots of eigenvalues λ_i, thus the equation can be rewritten as:

$$A^T A\mathbf{v}_i = \sigma_i^2 \mathbf{v}_i$$

Let V be the matrix formed by the non-null orthonormal eigenvectors of K and let Σ be the diagonal matrix with σ_i as its entries. Notice that the inverse of this diagonal matrix, Σ^{-1}, is also a diagonal matrix with $\frac{1}{\sigma_i}$ as its entries.

I claim that $U = AV\Sigma^{-1}$ is also a matrix with orthonormal columns. Taking the transpose on both sides leads to $U^T = \Sigma^{-T} V^T A^T$. Since any diagonal matrix is also a symmetric matrix, we have $\Sigma^{-T} = \Sigma^{-1}$. Thus, $U^T = \Sigma^{-1} V^T A^T$. Let's verify this claim by examining the dot product of two arbitrary columns of U.

$$\begin{aligned}
\langle u_i, u_j \rangle = u_i^T u_j &= \frac{1}{\sigma_i} \mathbf{v}_i^T A^T A \mathbf{v}_j \frac{1}{\sigma_j} \\
&= \frac{1}{\sigma_i} \mathbf{v}_i^T \sigma_j^2 \mathbf{v}_j \frac{1}{\sigma_j} \\
&= \frac{1}{\sigma_i} \sigma_j \mathbf{v}_i^T \mathbf{v}_j \\
&= \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}
\end{aligned}$$

By definition, we verified that U has orthonormal columns. Now we have

$$\begin{aligned}
AV\Sigma^{-1} &= U \\
AV &= U\Sigma \\
AVV^T &= U\Sigma V^T
\end{aligned}$$

It remains to show that $A = AVV^T$. Let's examine the columns of U and V, $\mathbf{u}_i$ and $\mathbf{v}_i$. Recall that $A^T A\mathbf{v}_i = \sigma_i^2 \mathbf{v}_i$ and $AV = U\Sigma$, we can write

$$\begin{aligned}
A\mathbf{v}_i &= \mathbf{u}_i \sigma_i \\
A^T A\mathbf{v}_i &= A^T \mathbf{u}_i \sigma_i \\
A^T \mathbf{u}_i \sigma_i &= \mathbf{v}_i \sigma_i^2 \\
A^T \mathbf{u}_i &= \mathbf{v}_i \sigma_i
\end{aligned}$$

This means that $\mathbf{v}_i \in \operatorname{coimg} A$. Since $\dim(\operatorname{coimg} A) = \operatorname{rank} A = k$, and V is a matrix with orthonormal columns, we know that $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ forms an orthonormal basis for $\operatorname{coimg} A$. That is, given a vector $\mathbf{x} \in \operatorname{coimg} A$, it can be written as $\mathbf{x} =$

$\sum_{i=1}^{k} c_i\mathbf{v}_i$ where $c_i \in \mathbb{R}$. Given a vector $\mathbf{y} \in \mathbb{R}^n$, since $\operatorname{coimg} A \perp \ker A$, we can decompose it as $\mathbf{y} = \mathbf{x} + \mathbf{z}$ where $\mathbf{z} \in \ker A$. Thus,

$$A\mathbf{y} = A(\mathbf{x} + \mathbf{z}) = A\mathbf{x} + A\mathbf{z} = A\mathbf{x}$$

Recall that columns of V are orthonormal, and let $\mathbf{c} = (c_1, \ldots, c_k)^T$. Now consider

$$\begin{aligned} AVV^T\mathbf{y} &= AV(V^T(\mathbf{x} + \mathbf{z})) \\ &= AV\left(V^T\sum_{i=1}^{k} c_i\mathbf{v}_i\right) \\ &= AV\mathbf{c} \\ &= A\sum_{i=1}^{k} c_i\mathbf{v}_i \\ &= A\mathbf{x} \end{aligned}$$

It follows that $AVV^T\mathbf{y} - A\mathbf{y} = (AVV^T - A)\mathbf{y} = 0$. Since $\mathbf{y}$ is abitrary, it must be that $AVV^T = A$. Hence, we obtain

$$AVV^T = A = U\Sigma V^T$$

as required. □

A.2. Gradient derivation from equation (2.7).

Given an arbitrary index $j \in [1, k]$, where k is the number of features, we can compute the partial derivative with respect to q_{ji} directly:

$$\begin{aligned}\frac{\partial \mathcal{E}_{iu}}{\partial q_{ji}} &= \frac{\partial}{\partial q_{ji}} \left(r_{iu} - \sum_{n=1}^{j} q_{ni} p_{nu} \right) \\ &= -p_{ju}\end{aligned}$$

where all the terms without q_{ji} are treated as constants. Similarly,

$$\begin{aligned}\frac{\partial \mathcal{R}_{iu}}{\partial q_{ji}} &= \frac{\partial}{\partial q_{ji}} \sum_{n=1}^{j} \left(q_{ni}^2 + p_{nu}^2 \right) \\ &= 2q_{ji}\end{aligned}$$

Then, we can use the Chain Rule on the squared error function and obtain:

$$\begin{aligned}\frac{\partial \mathcal{E}_{iu}^2}{\partial q_{ji}} &= 2\mathcal{E}_{iu} \frac{\partial \mathcal{E}_{iu}}{\partial q_{ji}} \\ &= -2\mathcal{E}_{iu} p_{ju}\end{aligned}$$

Combining the above results,

$$\begin{aligned}\frac{\partial \mathcal{L}_{iu}}{\partial q_{ji}} &= \frac{\partial \mathcal{E}_{iu}^2}{\partial q_{ji}} + \lambda \frac{\partial \mathcal{R}_{iu}}{\partial q_{ji}} \\ &= -2\mathcal{E}_{iu} p_{ju} + 2\lambda q_{ji}\end{aligned}$$

Without loss of generality, we can compute the partial derivative of $\mathcal{L}_{iu}$ with respect to p_{ju} the same way and obtain:

$$\frac{\partial \mathcal{L}_{iu}}{\partial p_{ju}} = -2\mathcal{E}_{iu} q_{ji} + 2\lambda p_{ju}$$