Matrix Factorization Techniques in Collaborative Filtering

Joey Dylan Bringley *

*DePaul University

CSC 575: Information Retrieval Final

This paper discusses two popular matix factorization techniques, Singular Value Decomposition and Gradient Descent, and their application in collaborative filtering. The theory and mathematics behind each algorithm will be introduced and discussed before implementing the algorithms and studying their performance on real data.

Matrix Factorization | Collaborative Filtering | Recommender Systems

Introduction

With the rise of companies like Amazon, Netflix, and Facebook collaborative filtering has become a very significant part of a users online experience. Everyday when we visit websites we are contantly recommend new books to read, movies to watch, or friends to encounter. One method companies use to make these recommendations is known as collaborative filtering.

In the most simple context, collaborative filtering is the process of making predicitions about a users tastes or preferences, based on data from data from many users. Think about it like this, if you and you're friend both like *Pulp Fiction*, you're probably likely to recommend another movie you like to your friend, seeing as how you both have a similar taste in film. On the other hand, if our friend doesn't like *Pulp Fiction*, you may be hestitant to recommend another movie you like. Collaborative filtering tries to do the same thing using various algorithms.

Matrix Factorization is a mathematical operation from linear algebra in which you well... factorize a matrix. This simply means breaking down a matrix into two or more matrices such that when they are multiplied together, they equal the original matrix. In this paper we will look into two matrix factorization techniques, Singular Value Decomposition and Gradient Descent both of which are commonly used for collobaritive filtering. In fact, these two techniques helped Simon Funk capture the Netflix Prize. For those unfamiliar, the Netflix Prize was a one million dollar competition, in which the challenge was to improve Netflix current recommendation engine's prediction accuracy by 10%. The mathematics behind each algorithm will be discussed, and the accuracy for each will be evaluated using data from MovieLens.org.

Singular Value Decomposition

Singular Value Decomposition comes from a theorem in linear algebra which states that any rectanglular m x n matrix can be broken down into the form:

$$A_{mn} = U_{mr} S_{rr} V_{nr}^T \quad [1]$$

where r is the rank of A.

On the right side of the equation we have orthoganol matrix, U consisiting of the eigen$-$vectors of AA^T, a diagonal matrix S representing the eigen-values of matrices U and V, and another orthogonal matrix V, conisiting of the eigen-vectors of $A^T A$.

SVD is particularly useful with regards to dimensionality reduction. The reason being is that we can approximate the original matrix through singular value decompoisition by choosing a value k such that $k < r$. Here, the value of k corresponds to how many eigen-values we choose to keep.The eigen-values are sorted in decreasing order such that if we choose a k where $k = r - 2$, we drop the two lowest eigen values. For any k the decomposed matrix then becomes:

$$A_{mn} \approx U_{mk} S_{kk} V_{nk}^T \quad [2]$$

Multiplying the three matrices will no longer yield a matrix identical to A but something similar to the original. In fact, what makes this method so popular is its garuntee concerning the low rank decomposition of A. If we considered the Frobenius Norm of a matrix defined as:

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} \quad [3]$$

Then it can be proven that SVD provides the best rank-k approximation for a matrix in terms of the frobenius norm. This is what is known as the optimal property, and was proved by mathematicians Eckart and Young in 1936. Specifically, the theorem states::

$$||A - A_k||_F \leq ||A - B||_F \quad [4]$$

where B is any matrix with rank k.

In simplest terms, this tells us that the difference between the original matrix A and our SVD approximated matrix A_k is *always* smaller than the difference between A and *any* and *all* other rank k matrices, B. It is impossible to find or create a matrix of rank k that is closer to A than A_k. This property is so useful because it garuntees us the minimal amount of information is lost when decomposing a matrix.

This feature makes SVD a very popular algorithm for tasks such as data compression and image analysis, but it is also widely used in recommender systems. In collabortive filtering, if we think of our data as a user-item matrix A_{mn} where m is the number of users and n is the number of items, then the entry a_{ij} represents the rating given by user i for movie j. SVD allows us to decompose this matrix while preserving valuable information about user preferences. Once decomposed this low-dimensional representation of the original matrix can be used to predict user preferences. Simply by multiply our 3 decomposed matrices back together, we generate an approximation of A where the element corresponds to our predicted rating for a user about item.

Reserved for Publication Footnotes

Choosing the Optimal k. An issue that arises with SVD is what value of k to choose. As one might untuitively expect, a higher k relates to a better representation of the original matrix, usually resulting in better accuracy, but a lower k means less work. It is a accuracy-time tradeoff that one must address when implementing the algorithm.

There is no concrete way of determining a proper value of k for SVD, and for that reason a couple different methods are used in this paper. The first being a simple trial and error approach. Setting different values of k and observing the errors. Also implemented is what as know as the Scree-Test. This is a simple graph where the eigen-values are plotted in a successive fashion, the general idea being to choose k where the curve starts to flatten. These graphs will be discussed later on.

Gradient Descent

In its most simple form, gradient descent is an algoirthm that minimizes a function. Starting with an objective function and a set of parameters, gradient descent initializes values for the parameters and then iteratively converges to the set of parameters that minimize the function. The parameters are updated in the opposite direction of the gradient. Why in the opposite direction? The gradient always points in the direction of the greatest ascent. Since we are trying to minimize a function (usually a loss function) we want to follow away from the gradient to find the lowest value.

A general objective function, E which gradient descent tries to minimize can be defined as:

$$E = \frac{1}{N}\sum_{n=1}^{N} e(h(x_i), y_i) \qquad [5]$$

Here the function inside the summatiom, e is determined by the evaluation metric being used. Our hypothesis function is h, whose values wre are trying to determine.

A step size (or learning rate) is also defined such that at each iteration the "step" you take along with the gradient relates to how large the step size is. The learning rate must be set such that the steps are not too large and the local minimum is missed, but not too small that it requires many iterations to find.

Stochastic Gradient Descent. One issue with gradient descent is it only garuntees to find the *local* minimum, an issue if the local minimum is not the global minimum. The fact that every training example is used to compute the gradient, while accurate, is very time and computationally expensive operation as well. But alongs comes *Stochastic Gradient Descent*! Here to help us out (although it does not completely erradicate the problem).

Stochatic Gradient Descent and gradient descent work the same way, only stochastic gradient descent uses one random sample point from the data to compute the gradient. For each iteration, you sample one point from the data, apply gradient descent, and calculate the error. Thus making the time complexity at every pass simply $O(1)$ versus $O(n)$ for gradient descent (where n is the number of data points).

To examine why this works we will define some random point, (x_k, y_k). If we use our objective function from equation 5, the gradient is:

$$-\nabla e(h(x_k), y_k) \qquad [6]$$

Taking the expected value of the gradient for this one point we can see:

$$E[-\nabla e(h(x_k), y_k)] = \frac{1}{N}\sum_{n=1}^{N} -\nabla e(h(x_k), y_k) \qquad [7]$$

Which is simply the gradient of the objective function in equation 5. So even though we are only using 1 point to calculate the gradient we are still going in the "average direction" at each iteration (in addition to some noise). The idea is that the noise will eventually average out, and we will wind up going in the direction we want.

Another advantage of stochastic gradient descent is the stochastic aspect it contains. We know the expected value at each iteration, but there is a random element that comes from samping one point. This random element has shown to be helpful when escaping local minimum. As it turns out, being too deterministic in optimazation can be a bad thing.

Stochastic Gradient Descent in Collaborative Filtering

The troubling aspect of SVD for collaborative filtering is that it requires one to decompose the user-item matrix which is often times extremely spare. To overcome this, one must impute the missing values in order for SVD to be performed. These imputations required to perform SVD are a cost to the algorithms performance, and when we encounter sparse datasets, it can be shown that SVD often doesn't perform well.

The idea of using gradient descent as a matrx facotorization technique in collaborative filtering was (as mentioned earlier) popularized by Simon Funk. In examining how to deal with such a large amount of data for the Netflix Prize, Funk realized that the original user-tem matrix could be factored into two matrices, a user preference matrix and a movie aspect matrix.

The intuition behind this is that there must be some underlying features that indicate how a user rates an item. If two users rate an item similary, that probably means they have similar preferences. Maybe the are both fans of sci-fi movies, or really like a certain actor. If these features were able to be identified, it could be used to predict a users rating for a certain item.

Start by taking a user-item matrix, R having size m x n. The SVD of this matrix then becomes:

$$R_{mn} = U_{mr} S_{rr} V^T_{nr}$$

where r is the rank of matirx, R. Additionally, we will choose some arbitrary k less than r so the matrix can now be approximated as:

$$R_{mn} \approx U_{mk} S_{kk} V^T_{nk}$$

If we blend the S_{kk} into both U and V we can let: $P = U_{mk} S_{kk}$ and $Q = S_{kk} V^T_{nk}$ our decomposed matrix now becomes:

$$R_{mn} \approx P_{mk} Q^T_{nk} \qquad [8]$$

Remember our original user item matrix R is of size m x n, corresponding to m user and n items in the data. The factorized matrices of R can now be thought of as a user preference matrix, P and a movie aspect matrix Q, each containing k features regarding user/movie preference. As an example, the first column in matrix P might indicate a users preference to comedies, while the first column in matrix Q might represent how relevant the movie is to the comedy genre.

To predict the rating for a movie j by user i we can simply calculate the inner product of the two vectors.

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj} \qquad [9]$$

Thus our predicition error, e we define as:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - p_i^T q_j)^2 \quad [10]$$

Now that we have our error function and the parameters we wish to optimize (P and Q) we can perform gradient descent. We start by taking the partial derivatives (computing the gradient) of our error function with respect to parameters p and q.

$$\frac{\partial}{\partial p_i} e_{ij}^2 = 2q_j(r_{ij} - p_i^T q_j) \quad [11]$$

$$\frac{\partial}{\partial q_j} e_{ij}^2 = 2p_i^T(r_{ij} - p_i^T q_j) \quad [12]$$

After calulating the gradient, we can then update the columns of P and Q at every iteration following the formulas:

$$p_i' = p_i + \eta \frac{\partial}{\partial p_i} e_{ij}^2 \quad [13]$$

$$q_j' = q_j + \eta \frac{\partial}{\partial q_j} e_{ij}^2 \quad [14]$$

where η is the learning rate. This process continues on for a set number of iterations.

Regularization. The only issue with the above algorithm is that it becomes susceptible to overfitting. A common technique to avoid this is known as Tikhonov Regulariation. This technique introduces a new parameter λ and penalizes the model for complexity. The new parameter λ aids in controling the weights of the features vectors and restricts the features from being updated too much at one time. With regularization our error function we seek to minimize now becomes:

$$e_{ij}^2 = (r_{ij} - p_i^T q_j)^2 + \frac{\lambda}{2} \sum_{k=1}^{k} (||P||^2 + ||Q||^2) \quad [15]$$

Taking the gradient we arrive at our new update rules for the feature vectors:

$$p_i' = p_i + \eta(2e_{ij}q_j - \lambda p_i) \quad [16]$$

$$q_j' = q_j + \eta(2e_{ij}p_j - \lambda q_j) \quad [17]$$

Data

To study these algorithms against real world data, each were implemented in Python and the results will be discussed below. The data used for the simulation was taken from *GroupLensResearch* who published data from the movie rating website MovieLens.org. While there are multiple versions of this dataset, the "Movie Lens Latest Dataset" (found here: https://grouplens.org/datasets/movielens/latest/) was used due to time and processing power concerns. The data consisted of 100,000 rows and 4 columns: userID, movieID, rating, and timeStamp. The ratings feature consisted of integers ranging from 1 (worst rating) to 5 (best rating), no decimal ratings were allowed. Alltogether, there were 943 unique users and 1,682 unique movies in the dataset.
Looking at the summary statistics for the ratings feature:

Table 1. Summary Statistics for Movie Ratings

Mean	Median	Mode	Std Dev	Skewness
3.53	4	4	1.13	-0.51

We see a slight negative skew, an average rating of 3.53, and a mode of 4.

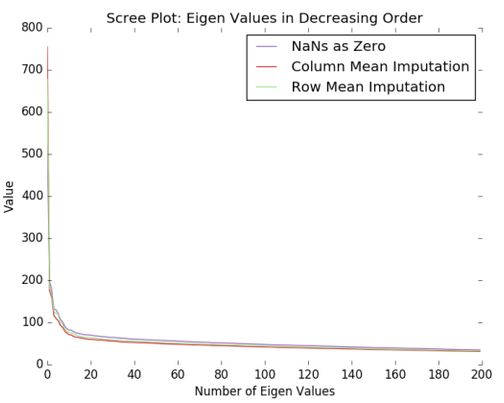
To allow for the algorithms to be implemented the dataset is modified into a *User-Item* matrix, consisting of 943 rows (the number of users) and 1,682 columns (the number of movies). Collaborative filtering datasets are very often sparse, the MovieLens data being no exception (sparsity = 93.695%). In testing the algorithms performance an 80/20 train/test split was used for the data.

Results

SVD Performance. As mentioned before, the problem with SVD in collaborative filtering tasks is it requires all elements of the matrix to be real numbers. With such a sparse matrix, imputating the miss values was an interesting taks. Three methods were implemented and evaluated for filling in these missing values, these being:

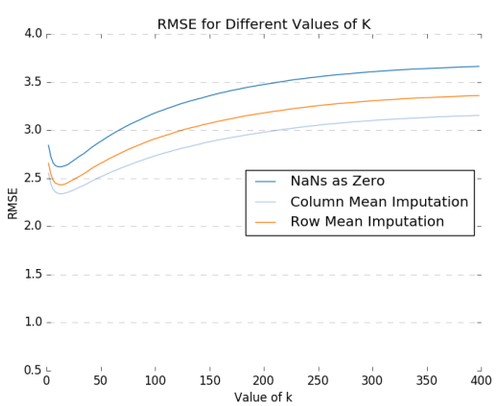
- Using zero.
- Using column means (average rating for a movie)
- Using row means (average rating given by a user)

We first examine the scree plot for the first 200 eigen values.



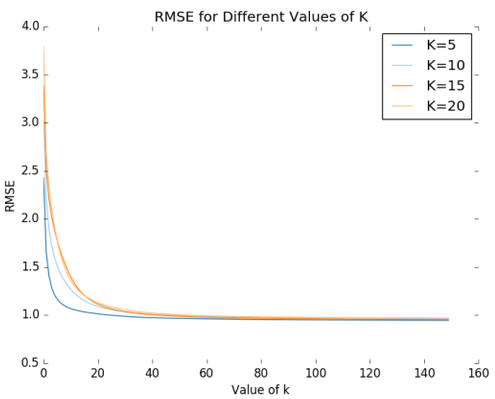
The idea behind the scree test is to choose k based on where the curve starts to level off. All three lines are in a similiar agreement here, as we see this happen around $k = 10$.

To verify the results above, the algorithm was run for all even values of k for k between 2 and 400. Again we look at the results.



The scree plot test seems to hav guided us in the right direction. For every instance, we see a low point in the $RMSE$ around $k = 10$. In a similar fashion the study done in [1] we find the column mean imputation produces the lowest error. Both column and row mean imputations are better than filling missing values with zero.

Gradient Descent Performance. Running the gradient descent algorithm which was implemented in python. We start the evaluation of gradient descent using a grid search technique searching for an optimal k. We fix our learning rate η at 0.01, the number of iterations at 150, and the regularization paramter λ at 0.1. Similar to SVD we can see that smaller values of k perform best. While, $k = 5$ converges fastest, they all have about the same error after 60 iterations or so.



The biggest surprise here is the improvement in predicition accuracy. With no optimatization regarding the parameters λ and η, we still arive at a $RMSE$ around 1. A great improvement over SVD. Adjusting the parameters η and λ however, we see little to no improvement in the algorithms accuracy. While not shown, stochastic gradient descent was implemented and while performing better than SVD, was outperformed by gradient descent.

Discussion

Part of SVD's poor performance here in relation to gradient descent can be attributed to the imputation process. With over 93% of the data missing, the algorithm relies heavily on what values are inserted into the missing entries. This idea has been discovered in many different studies [1] and [7], so using SVD with such a sparse matrix may not be the preferred route.

Gradient descent shows a large improvement over SVD and has the potential to perform faster when SGD is implemented. With more time, I look to improve the gradient descent algorithm with an introduction of bias weighting. A technique which examines which users and items are more likely to be rated higher/lower than others. In addition, I feel the problem of sparsity could be further evaluated and more advanced techniques could be implemented to further improve performance.

While not discussed here the alternating least squares approach is another popular algorithm in marix factorization. In fact, one study [3], reported better results than even bias stochastic gradent descent when the matrix is sparse.

1. B. Sarwar, G. Karypis, J. Konstan, and P. Niyogi, Application of Dimensionality Reduction in Recommender Systems, Army HPC Research Center (2000).
2. Y. Koren, Factorization Meets the Neighborhood: A Multifaceted Collaboriative Filtering Model, KDD (2008).
3. C. Aberger, An Anaylsis of Collaborative Filtering Techniques, Stanford University(2015)
4. L. Bottou, Stochastic Gradient Descent Tricks, Microsoft Research
5. M.J. Pazzani and D. Billsus, Content Based Recommendation Systems, Brusilovsky P., Kobsa A., Nejdl W. (eds) The Adaptive Web. Lecture Notes in Computer Science, vol 4321. Springer, Berlin, Heidelberg
6. S. Funk, Netflix Update: Try this at Home, http://sifter.org/ simon/journal/20061211.html (2006)
7. C. Volinksy, Matrix factorization techniques for recommender systems, IEEE Computer, vol. 42, pp. 3037, 2009