Netflix Prize and SVD

Stephen Gower

April 18th 2014

Abstract

Singular Value Decompositions (SVD) have become very popular in the field of Collaborative Filtering. The winning entry for the famed Netflix Prize had a number of SVD models including SVD++ blended with Restricted Boltzmann Machines. Using these methods they achieved a 10 percent increase in accuracy over Netflix's existing algorithm.

In this paper I explore the different facets of a successful recommender model. I also will explore a few of the more prominent SVD based models such as Iterative SVD, SVD++ and Regularized SVD. This paper is designed for a person with basic knowledge of decompositions and linear algebra and attempts to explain the workings of these algorithms in a way that most can understand.

Introduction

On October 2nd, 2006 Netflix began their contest to find a more accurate movie recommendation system to replace their current system, Cinematch. They promised a prize of one million dollars to anyone who could improve over the Cinematch system by at least 10% Root Mean Squared error (RSME). After three years of the contest, in 2009 the grand prize was awarded to team "Bellkor's Pragmatic Chaos", much of this paper draws from papers written by one of the members of the team, Yehuda Koren.

But before one can examine the algorithm developed to win the prize, first it must be known what must be addressed. Recommender systems are important to services such as Amazon, Netflix and other such online providers that aim to attract users. For Amazon, they want to present their users with products they want so they are more likely to spend their money through their site. On the part of Netflix, their motivation is to help users find movies and shows that fit their tastes because they are more likely to maintain a subscription.

Recommender Systems

Recommender systems fall into different groups depending on their strategy and these strategies fall largely into two groups. One group is the content filtering approach[1] and this approach essentially creates a profile for each user or product in order to classify each. This

relies upon the ability to gather external information. On the other hand there is the collaborative filtering approach[1], where the focus is analyzing relations between users and products and interdependencies between these groups. These relations are used to establish new connections. Each method comes with its own strengths and weaknesses. For instance while collaborative filtering is more accurate than content filtering, it suffers from a "cold start problem" [1]. This is because collaborative filtering relies on knowing more information about the user in relation to the products and for new accounts, this information is not available yet.

This division of recommender systems can be further subdivided into two models, neighborhood methods[1] and latent factor models[1]. The neighborhood method "neighbors" items rated by the same user in an effort to establish a taste. Then it compares these already rated items with ratings from other users on the same items to find users that exhibit the same taste[1]. Based upon these similarities, the system then considers related items that these other users rated highly and recommends them to the original user.

Latent factor models on the other hand attempt to analyze the user-item interaction through factors either inferred or gathered explicitly. Consider the following figure from the article "Matrix Factorization Techniques for Recommender Systems"[1]:

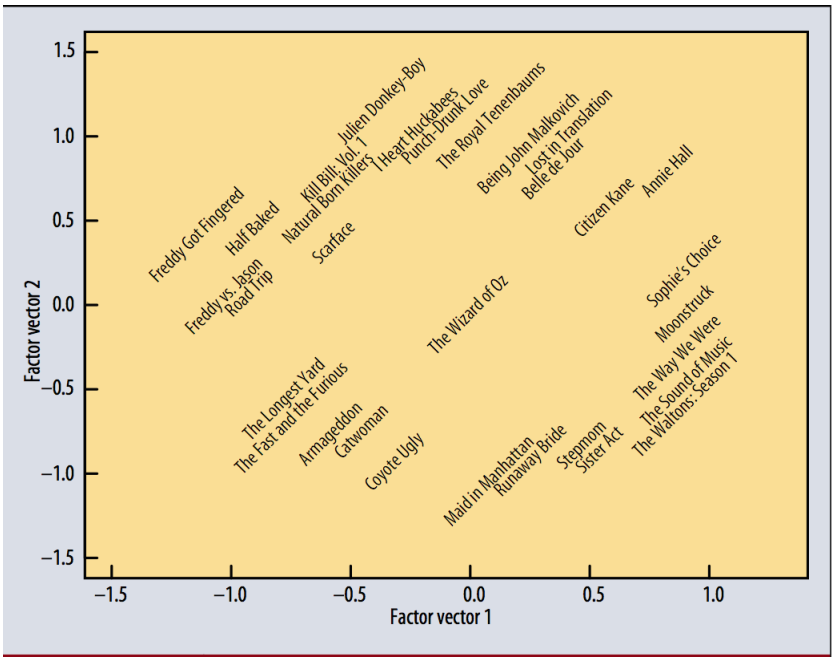

Figure 1: A mapping of movies based on two latent factors

The above is a mapping of items based upon two related latent factors. The factor on the horizontal axis groups movies on the left as horror / lowbrow comedies while movies on the right contain movies such as comedies with serious tones and dramas. On the vertical axis, the factor is characterized as "independent, critically acclaimed, quirky films on the top, and on the bottom, mainstream formulaic films" [1].

Data

These models are reliant upon a wealth of information gathered from the users of the service. The most obvious examples of this data is found in explicit feedback. This is the data that users provide the service. In the case of Netflix, users provide ratings for the movies and television shows they view. This takes on a value from one to five. Further examples of this data form are statistics provided by the user about themselves; such as gender, age, the shows and movies they decide to put into their queue, etc.

The problem involved with explicit feedback is that it assumes that the user participates actively in the service by sending feedback. Not only that but even active users are only really able to provide ratings for a small fraction of the total library provided by Netflix. This is where we introduce the other data that comes into consideration: implicit feedback. This is data collected on the user that may shed light on their opinions / preferences through behaviors such as search history, purchase history and even tracking the movement of the users mouse as they explore for the movies they might be interested in. This form of data is more plentiful for the system, but is less reliable on establishing connections between users and items. Both sets of data have their advantages and disadvantages, while explicit data is more obviously applicable, it is less available and provides a sparse rating matrix (since most users have not rated a good fraction of the total number of items). The exact opposite is true for implicit data, it provides a dense matrix but it is based on less powerful assumptions on the user-item interactions.

Matrix Factorization

Matrix factorizations (also known as matrix decompositions) are powerful tools in the mapping of users and items in factor-spaces. User-item interactions are modeled as inner products in the latent factor space with dimensionality f[1]. Each item is associated with a vector $\mathbf{q}_i \in \mathbb{R}^f$. Each user is associated with a vector $\mathbf{p}_u \in \mathbb{R}^f$ and the elements of the vector $\mathbf{p}_u$ involve characterizations of the level of interest that the user has in items that highly correspond, whether this interest is positive or negative[1]. Thus the inner product of the two vectors approximates the rating of the users rating of any specific item i, denoted as $\mathbf{r}_{ui}$, in the estimate[1]

$$r_{ui} \approx \mathbf{q}_i^* \mathbf{p}_u.$$

Once the system maps the vectors that describe the user-item interaction, it is a relatively simple task for it to then use the above equation to estimate the rating a user would give to a given item.

Adding Biases

It is the case that two items of the same rating are not equal in quality. One might be a film that exhibits higher quality either in its general regard by critics, the cost and effort that goes into creating the film, etc. Thus ratings alone cannot determine what movies should be recommended and that is when biases are added to the equation. This is essentially a

characterization of the base quality of an item, something that is independent of the rating analysis done and serves to better promote movies that are considered higher quality than others. The first order approximation of this bias is [1]

$$b_{ui} = \mu + b_i + b_u.$$

Thus b_{ui} is the bias involved in the rating r_{ui} and accounts for this shift based upon quality. In the equation, μ denotes the average rating by users, b_i is the bias from the item, such as the relative quality of the item. Finally, the b_u term is the user bias that is determined by how their ratings compare to the average. If they consistently rate above the average, this term is positive and the term is negative if they usually rate below the average. All terms are constants and so is the final resulting term b_{ui} If you wanted to find a first-order approximation of the expected rating for an item i by user u you would use[1]

$$r_{ui} = \mu + b_i + b_u + \mathbf{q}_i^* \mathbf{p}_u$$

Other Inputs

In addition to the ratings and biases discussed prior, we must take into account many of the other forms of inputs that affect the outcome of the recommendation. These inputs are generally made up of the implicit data discussed in the earlier section. Denote $N(u)$ as a set of items where we consider boolean values of interest by the user, either interest or no interest, based upon implicit data. In order to incorporate this data, the item i must be associated with a new vector $\mathbf{x}_i \in \mathbb{R}^f$. Then the user who showed preferences for the items in $N(u)$ is characterized by the normalized sum[1]

$$|N(u)|^{-0.5} \sum_{i \in N(u)} \mathbf{x}_i.$$

Another form of input that is available is known user attributes such as income, gender, address, etc. Taking in these inputs and forming the set of attributes $A(u)$ which is a set of Boolean attributes of user u. These inputs are associated with a factor vector $\mathbf{y}_a \in \mathbb{R}^f$ and this time, the sum of these factor vectors is not normalized. If we integrate these into the overall model we find a new approximation for the expected rating of user u of item i as[1]

$$r_{ui} = \mu + b_i + b_u + \mathbf{q}_i^* \left[\mathbf{p}_u + |N(u)|^{-0.5} \sum_{i \in N(u)} \mathbf{x}_i + \sum_{a \in A(u)} \mathbf{y}_a \right] \tag{1}$$

Time-Based Factors

Thus far, the model has only been dealing with inputs that exist at this current moment and generate recommendations based on a model that is time-independent. However, accuracy of the model can be improved by taking into consideration changes in user preference as a function of time. Of the terms that exist in our model, the following are time dependent: item biases, user biases and user preferences[1].

These should be obvious, but to explain, the relative quality of an item changes with time. So too does the way in which a user is likely to rate (more or less critical relative to the established average) and the preferences of the user changes with time. All of these time dependencies means that a user that once rated a movie at four stars for instance might now, years later, rate it at 3 stars instead. In order to capture this time dependence, the addition of many million parameters is necessary, however it can be simplified to the following model: [1]

$$r_{ui} = \mu + b_i(t) + b_u(t) + \mathbf{q}_i^* \mathbf{p}_u(t)$$

Here we have simply reflected the addition of time dependence in our model by modifying the aforementioned terms to be functions of time.

Learning Algorithm

A good model can adjust its approximations of the inter-dependencies between user and item and "learn" over time. This in mind, the regularized squared error on the set of known ratings is [1]

$$\min_{q*,p*} \sum_{(u,i)\in\kappa} (r_{ui} - \mathbf{q}_i^* \mathbf{p}_u)^2 + \lambda(||\mathbf{q}_i||^2 + ||\mathbf{p}_u||^2).$$

This function minimizes the sum which is the difference between the approximated ratings and the actual ratings squared. The second term in the sum is important primarily in the factor λ, which "controls the extent of regularization and is usually determined by cross-validation"[1]. Thus learning algorithms attempt to further help minimize this error and two of these forms include Stochastic gradient descent(SGD) and Alternating least squares(ALS). First, SGD computes the prediction error[1]

$$e_{ui} = r_{ui} - \mathbf{q}_i^* \mathbf{p}_u$$

then it modifies the item and user vectors using

$$\mathbf{q}_i \leftarrow \mathbf{q}_i + \gamma(e_{ui}\mathbf{p}_u - \lambda \mathbf{q}_i)$$

and

$$\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma(e_{ui}\mathbf{q}_i - \lambda \mathbf{p}_u).$$

While SGD is more popular, alternating least squares is also very applicable. Since the minimum function above relies on two unknowns, q_i and p_u, it is not convex and can not be solved for a minimum. If, however, we fix one of the values, we can solve for the other quadratically. Alternating least squares alternates between fixing one value and solving for the other. ALS is best used in systems that are centered around implicit data.

Solutions to the Netflix Prize

SVD and popular models

So far we have explored the ingredients of a good recommender system and even the winning model, SVD++. But it is not obvious how these models connect to SVD especially since

the estimations of ratings deal with vectors of user and item interactions. The reason these models are *based* on SVD is that it is used to form the vectors $\mathbf{p}_u$ and $\mathbf{q}_i$. When the Netflix prize competition began, they released the ratings of over 400,000 anonymous members, equal to over 100 million ratings. This huge matrix formed is simply a compilation of users as either rows or columns and items on the other. Ratings are stored at the intersection of the user and the item, with each value in the matrix ranging from 1-5 for rated material and 0 for movies that were not rated by the user. Denoting the rating matrix as A, the next step involves forming A^*A and performing the singular value decomposition on the matrix.

Once the SVD is applied to A, there is a lot of information extracted from this matrix that involves user-item interactions. The vectors that are formed from this process are the user taste vector $\mathbf{p}_u$ which is a vector that characterizes the level of interest the user has in items that are high on the corresponding factors. The other vector formed is $\mathbf{q}_i$ which assesses whether or not an item has certain factors. These are formed through the application of SVD in conjunction with Principal Component Analysis (PCA).

SVD

Throughout the paper I have referenced the user and item vectors $\mathbf{p}_u$ and $\mathbf{q}_i$ respectively. However it is not all too clear how these vectors come into existence and that is through SVD. Singular Value Decomposition or SVD approximates a single matrix A by the product of three matrices

$$A = USV^*$$

In [10] they outline a good way in which SVD can be used to form these user and item vectors. If we assume our rating matrix A is an m x n matrix with m users and n items then the vectors are defined as [10]

$$\mathbf{p}_u = U_k\sqrt{S}^T$$

and

$$\mathbf{q}_i = \sqrt{S}V^T$$

While I am not sure whether or not this is the exact process employed by the "BellKor's Pragmatic Chaos" group, it makes sense as a general idea. The vectors in the columns of the U matrix span the column space of the matrix A[10] i.e. the user characterizations. The columns of the V matrix span the row space of the matrix A[10] which is the space that characterizes the items. The S matrix contains singular values along the diagonal in decreasing order such that $[S]_{1,1} > [S]_{2,2} > \ldots > [S]_{n,n}$ which means that for approximations of these vectors, the largest of these singular values can be used to form the vectors. This characteristic is important and is why SVD is so great for these systems, because not only is it economical in terms of processes since vectors can be approximated very closely, but this also serves to emphasize strong latent factors and eliminate weak ones. The largest singular values correspond to the most related latent factors. Thus this raw SVD approach is good at analyzing latent factors and using these to create user and item vectors, but to incorporate neighborhood factors, PCA is implemented.

Principal Component Analysis

This process is described in paper [9]. Principal Component analysis works by first denoting a matrix C such that $C = A^*A$ and this can be weighted / normalized if the model calls for it. Thus we can find a matrix E and Λ such that

$$C = E^T \Lambda E$$

and equivalantly

$$ECE^T = \Lambda.$$

Form the linear transformation B such that

$$B = AE^T$$

and that the transformation allows for A to be diagonalized (transformed points are uncorrelated). This means

$$C_B = B^T B = ECE^T = \Lambda$$

Each column of B has variance λ_j and sorting by eigenvalue, we can take the first n eigenvectors as an approximation of the overall variance between the two points. This is because these first n vectors comprise of the majority of the overall variance between the points and thus a good approximation without taking all eigenvectors will focus on the vectors with the most variance. This process serves to put a number to the difference of two elements and characterize how related or unrelated they are. This is important when applying a neighborhood model that must take a set of users and find which users, based on ratings, are most related and use this to recommend movies to each user based on ratings from the other user. PCA is most connected to the neighborhood method and SVD is most connected to latent factor models and the final vectors include a blend of the two methods in order to form them.

Boltzmann Machines

There were many teams that competed in the Netflix prize and each had their own ideas of how to approach the task at hand. Generally the top teams formed their own variation of an SVD based model and some attempted to blend models to gain higher accuracy. The largest success was team "Bellkor's Pragmatic Chaos" that blended what they termed SVD++ which is essentially the model from (1). They blended SVD++ with a Restricted Boltzmann Machine algorithm. The following is the list produced by Pragmatic Theory on the list of RBM's used[3]:

Variant	Description
rbm100	Ordinary RBM from Section 2 of [5], with 100 softmax hidden units. The model is trained using mini-batches of 1000 users, a learning rate of $2x10^{-5}$, a weight decay of 0.02 and a momentum of 0.9.
crbm100	Conditional RBM from Section 4 of [5], with 100 softmax hidden units. The model is trained using mini-batches of 1000 users, a learning rate of $2x10^{-5}$, a weight decay of 0.02 and a momentum of 0.9.
crbm100x	Conditional RBM from Section 4 of [5], with 100 softmax hidden units. The model is trained using mini-batches of 1000 users, a learning rate of $2x10^{-5}$, a weight decay of 0.02 and a momentum of 0.9. The learning rate is progressively reduced by a factor of 8 over the last 6 iterations.
crbm200	Conditional RBM from Section 4 of [5], with 200 softmax hidden units. The model is trained using mini-batches of 1000 users, a learning rate of $2x10^{-5}$, a weight decay of 0.02 and a momentum of 0.9.
drbm100-500	Conditional Factored RBM from Section 5 of [5], with a factorization of rank 100 and 500 softmax hidden unit. The initial learning rate is set to 0.0005 for the hidden unit biases and the rank reducing matrices, 0.005 for all others. The learning rate is reduced exponentially by a factor of 1000 over 100 iterations. Update occurs by batches of 1 user, with no weight decay or momentum.
drbm160-640	Conditional Factored RBM from Section 5 of [5], with a factorization of rank 160 and 640 softmax hidden unit. The same training parameters were used as *drbm100-500*.
urbm20-1000	Conditional Factored RBM from Section 5 of [5], with a factorization of rank 20 and 1000 softmax hidden unit. In this variant, the role of the users and the movies is reversed: each movie is modeled by a number of units, and weights connect the hidden units to the users. The initial learning rate is set to $4.51005x10^{-7}$ for the hidden unit biases and the rank reducing matrices, $6.21418x10^{-5}$ for all others. The learning rate is reduced by a factor of 0.0315713 per iteration. Update occurs by batches of 1 user, with a weight decay of 0.0946444 on all parameters except the hidden unit biases, and no momentum. The meta-parameters were selected using the Nelder-Mead Simplex Method. During the selection of the meta-parameters, the maximum number of iterations is limited to 100.
integ0-0-0TZ-grbm200	RBM with 200 Gaussian visible units as described in [7], running on residuals of *integ0-0-0TZ*. The model is trained using mini-batches of 1000 users, an initial learning rate of $8.18119x10^{-5}$ which decreased by a factor 0.000390623 at each iteration. A weight decay of 0.00012058 + 0.0569244/N is used where N is the support of the respective parameter. Momentum is set to zero. The meta-parameters were selected using the Nelder-Mead Simplex Method. During the selection of the meta-parameters, the maximum number of iterations is limited to 100.
mfw31-10-grbm200	RBM with 200 Gaussian visible units as described in [7], running on residuals of *mfw31-10*. The same training parameters were used as *integ0-0-0TZ-grbm200*.

For the references in the table, [5] accurately refers to paper [5] in the reference section of this paper, however [7] in the table refers to paper [2] in the reference section of this paper. Largely, the Boltzmann machines follow the same goals as the SVD methods, they seek to predict missing values based upon values given. Boltzmann machines are an important part of the overall algorithm. On their own, the SVD and Boltzmann approaches reach good degrees of accuracy but were unable to exceed the 10% threshold. But a blend of the two is what gave that extra accuracy bump to win the contest. The specifics of this method warrant almost an entire paper alone so the purpose of this section is to remind the reader. A good paper to look at on this subject is cited as paper [5].

Additional Methods: Regularized SVD

Another entry into the contest was something known as regularized SVD, a method outlined in [7]. The predictions for regularized start the same way as

$$\bar{y}_{ij} = \mathbf{u}_i^* \mathbf{v}_j.$$

Where $\bar{y}_{ij}$ is the prediction based upon the inner product of the two vectors of parameters $\mathbf{u}_j$ and $\mathbf{v}_j$. Again, these vectors are trained using gradient descent and regularization. Before using the gradient descent though, a baseline prediction is subtracted from the vectors

$$r_{ij} = y_{ij} - \bar{y}_{ij}$$

$$u_{ik} += lrate * (r_{ij} v_{jk} - \lambda u_{ik})$$

$$v_{jk} += lrate * (r_{ij} u_{jk} - \lambda v_{ik})$$

Where the constant parameters are $lrate = 0.001$ and $\lambda = 0.02$ and the training on the data stops when the error associated with the prediction starts to increase with further iterations.

In the same paper, an improved regularized SVD is described where the prediction equation is

$$\bar{y}_{ij} = c_i + d_j + \mathbf{u}_i^* \mathbf{v}_j.$$

Further, these new parameters (which act as biases) c_i and d_j are also trained using

$$c_i += lrate * (r_{ij} - \lambda_2(c_i + d_j - global\_mean)$$

and

$$d_j += lrate * (r_{ij} - \lambda_2(c_i + d_j - global\_mean)$$

While the regularized SVD method tests with a RMSE of 0.9094, the improved regularized SVD method tests with an RMSE of 0.9018.

Additional Methods: Iterative SVD

This approach is much more easily recognizable as an SVD based function and is outlined in [6]. This model is a piece-wise function that finds the predicted values $R_{iu}^{(t)}$ as

$$R_{iu}^{(t)} = \begin{cases} R_{iu}, & \text{if } iu \in D \\ [\sum_k U_k S_k V_k^*]_{iu}^{(t-1)}, & \text{otherwise} \end{cases}$$

Where D is the original rating matrix. Essentially what this iterative function does is, it takes the rating matrix and if a rating exists in the original rating matrix, then the function leaves the variable at that. If however that space in the original rating matrix is a zero value, then it attempts to approximate it using SVD. This function is combined with an Expected Maximization and the equation

$$\sum_{iu} (R_{iu}^{(t)} - \mu_{iu})^2$$

where $\mu_{iu} = [\sum_k U_k S_k V_k^*]_{iu}$. The method repeats these iterations until the sum converges.

Conclusion

It is clear that the applications of SVD in recommender systems are very powerful and extensive. Conceptually it makes sense that an SVD based model is very well fit to a recommender system since a large strength of SVD is its ability to establish the relatedness of items based on a set of factors. So powerful in fact that SVD is featured in almost all of the top entries for the Netflix prize.

References

[1] Y. Koren, R. Bell, C. Volinsky, *Matrix Factorization Techniques for Recommender Systems*. IEE Computer Society, 2009.

[2] Y. Koren, *The BellKor Solution to the Netflix Grand Prize*. 2009.

[3] M. Piotte, M. Chabbert, *The Pragmatic Theory Solution to the Netflix Grand Prize*. Pragmatic Theory inc, Canada, 2009.

[4] A. Toscher, M. Jahrer, *The BigChaos Solution to the Netflix Grand Prize*. AT&T Labs, New Jersey, 2009.

[5] R. Salakutdinov, A. Mnih, G. Hinton, *Restricted Boltzmann Machines for Collaborative Filtering*. U. of Toronto, Canada, 2007.

[6] M. Ali Ghanzanfar, A. Prugel-Bennett, *The Advantage of Careful Imputation Sources in Sparse Data-Environment of Recommender Systems: Generating Improved SVD-based Recommendations*. School of Electronics and Computer Science, UK, 2013.

[7] A. Paterek, *Improving regularized singular value decomposition for collaborative filtering*. Institute of Informatics, Poland, 2007.

[8] Y. Koren, *Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model*. AT&T Labs, New Jersey, 2008.

[9] K. Goldberg, T. Roeder, D. Gupta, C. Perkins, *Eigentaste: A Constant Time Collaborative Filtering Algorithm*. IOER and EECS, California, 2000.

[10] M. Vozalis, K. Margaritis, *Applying SVD on Generalized Item-based Filtering*. International Journal of Computer Science & Applications, 2006