Introduction to Matrix Factorization for Recommender Systems

Shalin Shah
Applied and Computational Mathematics, Johns Hopkins University
shah.shalin@gmail.com

Abstract

Recommender systems aim to personalize the experience of user by suggesting items to the user based on the preferences of a user. The preferences are learned from the user's interaction history or through explicit ratings that the user has given to the items. The system could be part of a retail website, an online bookstore, a movie rental service or an online education portal and so on. In this paper, I will focus on matrix factorization algorithms as applied to recommender systems and discuss the singular value decomposition, gradient descent-based matrix factorization and parallelizing matrix factorization for large scale applications.

1. The Singular Value Decomposition

Matrix factorization algorithms decompose a matrix into components which can then be used for various purposes. Eigen-decomposition decomposes a square matrix into a matrix P and a diagonal matrix D.

$$A = PDP^{-1}$$

Where P is a matrix that contains the eigenvectors of A and D is a diagonal matrix that contains the eigenvalues of A on its diagonal. This decomposition is particularly useful for computing large matrix powers, in applications such as Markov processes. Principal components analysis (PCA) uses eigen-decomposition to compute the principal components which are eigenvectors of the covariance matrix A^TA. A very related decomposition is the Singular Value Decomposition (SVD) which can factorize non-square matrices with a similar interpretation. SVD factorizes a rectangular m x n matrix A into three matrices, U, Σ and V^T. U is an m x m matrix, Σ is an m x n diagonal matrix and V is an n x n matrix. U captures information about the rows of A and V captures information about the columns of A. The columns of U and V are called the left and right singular vectors and the diagonal values of Σ are called the singular values of A (square roots of the eigenvalues). U and V are orthogonal matrices which says that all columns are orthogonal to the other columns and are unit vectors (also, $U^T = U^{-1}$ and $V^T = V^{-1}$). The SVD takes the following form:

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

SVD is the same as PCA by noting the following:

$$A^TA = (U\Sigma V^T)^T\left(U\Sigma \mathrm{V}^{\mathrm{T}}\right) = V\Sigma^{\mathrm{T}}(U^TU)\Sigma\mathrm{V}^{\mathrm{T}} = V\left(\Sigma^{\mathrm{T}}\Sigma\right)\mathrm{V}^{\mathrm{T}}$$
$$AA^T = \left(U\Sigma\mathrm{V}^{\mathrm{T}}\right)(U\Sigma V^T)^T = U\Sigma(V^T\mathrm{V})\Sigma^TU^T = U\left(\Sigma\Sigma^{\mathrm{T}}\right)\mathrm{U}^{\mathrm{T}}$$

U contains eigenvectors of AA^T and V contains the eigenvectors of A^TA. The diagonal entries of Σ contain the square roots of the eigenvalues of either (they are equal). Both AA^T and A^TA are square symmetric matrices whose eigen-decomposition exists, and the eigenvectors are orthogonal. The next section shows how this decomposition can be used in recommender systems.

2. SVD and Recommender Systems

$$\begin{bmatrix} UserItemMatrix & Item\ 1 & \cdots & Item\ n \\ User\ 1 & 1 & \cdots & 0 \\ User\ 2 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ User\ m & 0 & \cdots & 1 \end{bmatrix}$$

If A is an m x n matrix, and noting the similarity between SVD and PCA, it is clear that U contains latent information about the rows of A (of dimension m) and V contains latent information about the columns of A (of dimension n). The rank of a matrix is the number of non-zero rows (or columns) in its reduced form (which can be found using Gaussian-Elimination).

$$The\ rank\ of\ a\ matrix\ is\ the\ number\ of\ nonzero\ singular\ values.$$

$$Proof:$$

$$U\ and\ V\ are\ square\ full\ rank\ matrices\ because\ they\ are\ orthogonal.$$

$$UU^T = I_m => \det(U) = 1 \text{ (because } \det(U) = \det(U^T)\text{)} =>\ U\ is\ nonsingular$$

$$=> U\ is\ full\ rank => rank(U) = m, rank(V) = n, rank(\Sigma) = \mathrm{s},$$

$$\text{(where s = number of nonzero singular values)}$$

$$Rank\ of\ the\ product\ of\ two\ nonsingular\ matrices, XY\ is\ \min\ (rank(X), rank(Y))$$

$$So, rank(U\Sigma) = \min(\mathrm{m}, \mathrm{s}) = \mathrm{s}$$

$$and\ rank\big((\mathrm{U\Sigma})\mathrm{V^T}\big) = \min(s, n) = s$$

$$Thus, rank\big(U\Sigma\mathrm{V^T}\big) = s = number\ of\ nonzero\ singular\ values$$

$$\blacksquare$$

By ignoring all singular values less than the kth largest (and similarly the singular vectors in U and V), a rank reduced matrix can be constructed from A. In the rank reduced form, U is now an m x k matrix, Σ is a k x k diagonal matrix and V^T is a k x n matrix (the rank reduced form using SVD is the best rank k approximation to the original matrix A, as compared using the Frobenius norm of the difference). Rank reduction to rank k constructs k-dimensional latent factors in UΣ and ΣV^T. These latent factors can be interpreted as embeddings in some space. If A originally contained users as the rows and retail products as its columns, the k-dimensional rows of UΣ contain the latent factors of the users and the k-dimensional columns of ΣV^T contain the latent factors of the retail products [12] (where there are m users and n items i.e. the original matrix A is an m x n matrix with a 1 if the user interacted with the respective item and a 0 otherwise).

Inner products between the user and item latent factors can be interpreted as an affinity of the user for a specific item. Thus, the top c non-increasing inner products of a user latent factor with all item factors gives the top c items that can be recommended to the user. The interpretation is that these rank-reduced latent factors contain latent information about the user and the item, such as inclination towards categories, colors, style, brand etc. and that a large value of the dot product between the user and item factors implies similar interests and properties of the user and the item respectively.

SVD for recommender systems was used by participants of the Netflix prize [1]. SVD can be computed in $\min(O(m^2n), O(n^2m))$ time i.e. roughly cubic for a square matrix. MATLAB might have more efficient ways of computing the SVD. The singular values are, in some form, the relative importance of the latent information present in U and V. There are other ways of computing embeddings or latent factors using neural networks (e.g. word2vec [2]) which are not based on matrix factorization, though, there have been attempts to relate matrix factorization with neural embeddings [3]. All algorithms use some form of a context to construct the rows and columns of A (i.e. items bought by one user are somehow similar, and items viewed in the same web session are similar and so on).

3. Matrix Factorization using Gradient Descent

The disadvantage of SVD is that it cannot handle matrices with missing values. In most recommender systems applications, a user interacts with a very small subset of products and it is not necessarily true that the user is not interested in all of the remaining majority of products. And SVD, in its naïve form requires too many computing resources and is not easily parallelizable for very large-scale systems.

It is possible to learn the latent factors of the users and items using gradient descent. By choosing an appropriate cost function (e.g. sum of squared errors) and then doing updates of the latent factors iteratively, the algorithm can learn more effectively, and it is a lot more scalable.

Say v_u = *user latent factor and* v_i = *item latent factor, and*
r_{ui} = *rating that the user gave to item i*
(*this could take on values of* 0 *and* 1, *and a*
logistic squashing function could be used for the dot product)

$$\textit{Cost Function} = f = (r_{ui} - v_u . v_i)^2$$

And its derivatives:

$$\frac{\partial f}{\partial v_u} = 2(v_u v_i^2 - r_{ui} . v_i)$$

$$\frac{\partial f}{\partial v_i} = 2(v_u^2 v_i - r_{ui} . v_u)$$

In the algorithm to update the latent factors of users and items, it is sufficient to use just those items that a user interacted with, and negatively sampling from the items that the user did not interact with (i.e. putting equal emphasis on items viewed and not viewed). Then stochastic gradient descent could be used to update the user and item latent factors:

```
Stochastic Gradient Descent:
Initialize all user and item factors to $\mathbb{N}(0,1)$
Sample a user u, a viewed item i and a not viewed item j
Update the user u, item i and item j factors using the derivatives just computed
$v_u = v_u - \eta \frac{\partial f}{\partial v_u}$
$v_i = v_i - \eta \frac{\partial f}{\partial v_i}$
Repeat for the desired number of iterations, or until convergence
```

Where, η is the learning rate or step size (usually something like 0.01 or even less). This algorithm will learn the latent factors of all users and items. To recommend items to a user, the top c items with the highest $v_u v_i$ could be used.

Geometrically, similar users will appear in similar areas and similar items will get clustered together. The value of k (dimension of the latent factors) is a parameter, and usually a value between 50 and 100 works pretty good for many cases.

4. Parallelizing Matrix Factorization using Spark

Matrix factorization using stochastic gradient descent can be easily parallelized on Spark.

Spark [5][6] is a parallel processing system which is based on a particular format of storing data in memory and disk called resilient distributed datasets (RDD). The creator of the Spark based algorithm can choose which RDD resides in memory and which RDD resides entirely on disk. Spark has a driver which starts the application and controls the compute nodes that process some parts of the data in parallel.

Spark has map, reduceByKey, groupByKey and many other operations. A groupByKey operation groups data by keys. The rows that have the same key are collected on one compute node. This is similar to the reduce operation in map-reduce [7][8]. Spark also has a broadcast mechanism in which data can be broadcasted to all nodes of the application. A map operation can transform data from one format or schema to another, similar to the map operation in map-reduce.

In the parallel implementation of matrix factorization, there are several alternatives possible. Usually, it is not unreasonable to assume that the number of users is several orders of magnitude larger than the number of products (items). It is sufficient to parallelize the users into an RDD and broadcast the latent factors of all products to all nodes.

The algorithm:

1. *Initialize all user factors to* $\mathbb{N}(0,1)$ *and create an RDD<User>*
2. *Initialize all product factors to* $\mathbb{N}(0,1)$ *and store on the driver*
3. *Broadcast all product factors using the Spark broadcast mechanism*
4. *On the compute node of the RDD<User>, sample an item that the user interacted with and a random item that the user did not interact with.*
5. *Update the user factor and return it*
6. *Update both item factors and return them*
7. *Complete the iteration by updating RDD<User> with the updated user factors and updating the item factors on the driver*
8. *Iterate from 3 till convergence*

Convergence can be detected when the metric used to measure the recommendations doesn’t change much or starts reducing (early stopping). Many metrics can be used to measure the recommendations such as NDCG, Hit Rate, and Area under the Curve (AUC).

5. Conclusion

In this paper, I gave a brief overview of SVD, argued that SVD can be used for personalized recommendations, gave an implementation of matrix factorization based on gradient descent and then showed that it can be implemented on Spark for scale. There are several variations that can be used, especially the cost function for gradient descent. Item and user biases could be added to the cost function. Bayesian Personalized Ranking [9][10] places a Gaussian prior on the latent factors. It might be interesting to compare pure feature based ranking algorithms such as Lambda Rank [11] with the matrix factorization approach. The latent factors could be compared with the ones created by neural embeddings.

References

[1] Koren, Yehuda, Robert Bell, and Chris Volinsky. "Matrix factorization techniques for recommender systems." *Computer*8 (2009): 30-37.

[2] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781* (2013)

[3] Levy, Omer, and Yoav Goldberg. "Neural word embedding as implicit matrix factorization." *Advances in neural information processing systems*. 2014.

[4] "Understanding matrix factorization for recommendation", Nicolas Hug

[5] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[6] Spark homepage on Apache.org, https://spark.apache.org/

[7] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

[8] https://en.wikipedia.org/wiki/MapReduce

[9] Rodrigo, Alfredo Láinez, and Luke de Oliveira. "Distributed Bayesian Personalized Ranking in Spark."

[10] Rendle, Steffen, et al. "BPR: Bayesian personalized ranking from implicit feedback." *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*. AUAI Press, 2009.

[11] Burges, Christopher JC. "From ranknet to lambdarank to lambdamart: An overview." *Learning* 11.23-581 (2010): 81.

[12] Sarwar, Badrul, et al. *Application of dimensionality reduction in recommender system-a case study*. No. TR-00-043. Minnesota Univ Minneapolis Dept of Computer Science, 2000.