Final Project
Data Mining
CMPT 741

Gustavo Felhberg
Brie Hoffman
Jorge Marcano

12 / 12 / 2017

1 Introduction

For this project, we were given a data set of 2,038,130 known ratings by Yelp users for various businesses and asked to build a recommender system to predict unknown ratings for a list of 158,024 user and business combinations, using algorithms and implementations of our choosing.

2 Approach

The process of finding a suitable method to predict the test ratings was hindered due to the large size of the training dataset. Most libraries and packages for recommendation systems do not work with this amount of data, and end up leading to memory errors at runtime. The main reason for this is that most of the algorithms used in recommendation systems require a dense matrix filled with all the ratings (both known and predicted), so even if the initial training data fits in memory, the matrix produced at the end of the algorithm with all ratings does not.

We first tried to implement matrix factorization in Python, following the method given in a tutorial [8]. This method ended up producing a memory error when running, in addition to taking a long time to run. After this we tried using the scikit-learn Python library and its Non-Negative Matrix Factorization class, but this also had the same memory problem. We tried both of these options using a sparse matrix from the SciPy Python package, but to no avail. We then found a Python library called Surprise, built for the implementation of recommendation systems. Surprise includes algorithms such as Single Value Decomposition (SVD), Matrix Factorization and Co-Clustering, among others. Lastly we used the Apache Spark implementation of matrix factorization which uses the alternating least squares (ALS).

After finding and testing these different algorithms, we decided that the best course of action was to use multiple algorithms to produce an **ensemble** prediction. We selected the algorithms that ran successfully with the large training set and used each one multiple times, each time with slightly different hyperparameters settings, to produce multiple prediction files. We then calculated the average of all the predictions. We chose to produced ensemble predictions because it has been shown to yield a set of predictions with lower variance than any of the originals, which produces better results [2].

In order to test the hyperparameters of each algorithm, we used a cross-validation process and did multiple submissions in the Kaggle competition for each algorithm. Our final result, and the best we obtained, came from the average of multiple submissions.

3 Algorithms applied

In this section, we will explain each algorithm used, as well as the hyperparameters available to each of them, and the best results obtained for each case.

3.1 Matrix Factorization

Matrix Factorization is the recommendation method popularized by Simon Funk during the Netflix contest, whereby users and items are mapped to a joint latent factor space, and unknown ratings are given by the cross product of the user's and item's respective latent factor vectors [5]. In our project we used five different implementations of matrix factorization: four of which were provided by the Surprise library, and one from the Apache Spark MLlib library. All are described below.

3.1.1 Singular Value Decomposition (SVD)

The SVD algorithm provided by Surprise is matrix factorization with user and item biases taken into account. The predicted rating for user u and item i is given by

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u \tag{1}$$

where μ is the average known rating for all items, b_u and b_i are the user and item biases, and q_i and p_u are the latent factor vectors. The values for b_u, b_i, q_i and p_u are found by minimizing the regularized squared error, given in equation 2, using stochastic gradient descent, iterating over all known ratings until convergence:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2) \tag{2}$$

The hyperparameters that were tuned for this algorithm were the number of latent features used, number of epochs and learning rate. The tests uploaded to Kaggle were the best out of the different cross-validation combinations. They all used 50 epochs. One was done with the default parameters of the algorithm (20 epochs, 0.005 learning rate and 100 features), another with 50 epochs, 50 features and 0.005 learning rate and another with 50 epochs, 50 features and 0.01 learning rate. The best results were obtained with the default parameters but with 50 features instead of 100 and obtained a Kaggle score of 1.31023.

3.1.2 Singular Value Decomposition Plus Plus (SVD++)

This is an extended version of SVD that takes into account implicit ratings. We decided to use this one because, once again, SVD is one of the most popular algorithms for solving this kind of problem, and we wanted to see the effect taking the implicit ratings into account would have on this data set.

We used the implementation provided by Surprise, where the predicted rating is given by

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_j \right) \tag{3}$$

where y_j are a new set of item factors that capture the fact that user u rated item j, regardless of the rating value, and I_u is the set of users who rated item j [4].

The hyperparameters tuned for this algorithm were the same as in SVD. The best results obtained and the ones that were uploaded to Kaggle, were one with default parameters (20 features, 20 epochs, 0.007 learning rate and 0.02 regularization term), another with 50 factors, 30 epochs, 0.01 learning rate and 0.02 regularization term, and another with 50 factors, 50 epochs, 0.05 learning rate and 0.02 regularization term. The one that obtained the best score was the first one, using default settings with a Kaggle score of 1.31466.

3.1.3 Probabilistic Matrix Factorization (PMF)

PMF was presented by Ruslan Salakhutdinov and Andriy Mnih also in the context of the Netflix contest. This method scales linearly with observations and, thus, can better handle large and sparse datasets [7]. To implement an algorithm equivalent to Probabilistic Matrix Factorization, we again used Surprise's SVD algorithm, but this time setting the biased parameter to False [4]. This gives a predicted rating of

$$\hat{r}_{ui} = q_i^T p_u \tag{4}$$

The hyperparameters that achieved the best results through cross-validation were 100 features, 20 epochs, 0.005 learning rate and 0.02 regularization term. With these hyperparameters we achieved a Kaggle score of 1.56279.

3.1.4 Non-negative Matrix Factorization (NMF)

This algorithm is very similar to SVD. The basic difference is that it keeps the user and item factors always positive [4]. It does this by using an adaptive learning rate that is adjusted during the update phase on each iteration to account for any negative components [6].

A cross-validation search for the best hyperparameter settings gave the best results with 15 features, 50 epochs, 0.005 learning rate and 0.06 regularization term for users and items. These settings achieved a Kaggle score of 1.52098

3.1.5 Alternating Least Squares (ALS)

The ALS technique tries to learn the factor vectors p_u and q_i by minimizing the regularized squared error on the set of known ratings:

$$min_{p^*,q^*} = \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(||q_i||^2 + ||p_u||^2)$$

Since both q_i and p_u are unknowns, this equation is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally. Thus, ALS techniques alternate between fixing the q_i and fixing the p_u. When all p_u are fixed, the system recomputes the q_i by solving a least-squares problem, and vice versa. This ensures that each step decreases the equation until convergence [5].

We used the implementation of ALS in the Apache Spark MLlib library [1]. We used maximum number of iterations as 10, regularization parameter as 0.1, and the predictions made by this model scored 1.41254 on Kaggle.

3.2 Co-Clustering

Co-clustering is a collaborative filtering technique for recommendations whereby the user-item matrix, M, is co-clustered into user-clusters and item-clusters. It is the only non-matrix factorization algorithm we had any success with.

The predicted recommendation for user u and item i is

$$\hat{r}_{ui} = \overline{C_{ui}} + (\mu_u - \overline{C_u}) + (\mu_i - \overline{C_i})$$

where $\overline{C_{ui}}$ is the average rating of the particular user-item co-cluster, μ_u and μ_i are the average ratings of user u and item i, and $\overline{C_u}$ and $\overline{C_i}$ are the average ratings of the user-cluster and item-cluster. George and Merugu showed co-clustering to be comparable to SVD and NMF in terms of accuracy, but its particular strength is its support for dynamic addition of new users and items[3]. While not relevant to this project, this is a big strength for online systems where a computationally intensive retraining of the recommendation model every time a new user or item is added would

be undesirable.
We used the implementation of Co-clustering provided by Surprise. Hyperparameters tuned were the number of user clusters, the number of item clusters, and the number of training epochs. Through a cross-validation grid-search of parameters, we determined the optimal settings to be 3 user-clusters, 3 item-clusters and 100 epochs. The predictions made by this model with these parameters scored 1.46421 on Kaggle.

3.3 Algorithms we did not use

As mentioned above, collaborative filtering techniques that rely on building a complete N x M matrix of recommendations was impossible due to memory constraints. This ruled out both item- and user-based k-nearest neighbors algorithms, as well as the SlopeOne algorithm.

3.4 Boosting

An attempt was made to improve predictions using the boosting method. Following the Adaboost algorithm for collaborative filtering proposed by Bar et. al., weights were assigned to each rating, and after each iteration the weights were updated to boost the weights of ratings that were predicted incorrectly. Unfortunately this didn't improve our results significantly.

3.5 Ensemble Average

After observing the results obtained with the different algorithms, some of which didn't produce particularly good scores in Kaggle, we decided it would be a good idea to create an ensemble of the different models. The idea is that, when averaging the results, the variance of the predictions is reduced and presumably would generate better results. We wrote a Python program to produce of an average rating from input csv files containing prediction results from individual algorithms. We used this program to create ensemble predictions from different combinations of individual algorithms.

4 Results

The table below shows the best Kaggle results for each algorithm or ensemble listed. In terms of individual algorithms, our best scores came from the Surprise implementations of SVD and SVD++. Out of the multiple ensembles we tried, the one that got the best results was combining our best results from SVD, SVD++ and Co-clustering algorithms.

Algorithm / Ensemble	**Kaggle Score**
SVD	1.31023
SVD++	1.31466
PMF	1.56279
NMF	1.52098
CoClustering	1.46421
ALS	1.41254
SVD and SVD++	1.30612
Best ensemble	**1.29742**

References

[1] Collaborative filtering - rdd-based api. https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html.

[2] Ariel Bar, Lior Rokach, Guy Shani, Bracha Shapira, and Alon Schclar. Improving simple collaborative filtering models using ensemble methods. In *International Workshop on Multiple Classifier Systems*, pages 1–12. Springer, 2013.

[3] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. In *Data Mining, Fifth IEEE international conference on*, pages 4–pp. IEEE, 2005.

[4] Nicolas Hug. Matrix factorization-based algorithms. http://surprise.readthedocs.io/en/stable/matrix_factorization.html, 2015.

[5] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer Society*, pages 42–49, 2009.

[6] Xin Luo, Mengchu Zhou, Yunni Xia, and Qingsheng Zhu. An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems. *IEEE Transactions on Industrial Informatics*, 10(2):1273–1284, 2014.

[7] Andriy Mnih and Ruslan R Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2008.

[8] Albert Au Yeung. Matrix factorization: A simple tutorial and implementation in python. http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/, 2010.