Recommendation Systems Applied to the Yelp Dataset

Shawn Anderson Maria Babaeva
Ka Hang Jacky Lok

Department of Computing Science
Simon Fraser University

INTRODUCTION

In this project, we were asked to build a Recommender System for Yelp. Yelp focuses on providing crowd-sourced business ratings and reviews for commercial establishments that we will refer to as items or businesses. We need to predict the "rating" or "preference" that a Yelp user would give to an item.

To be able to do that we needed to create a Recommender system. Recommender System is a well-known and widely used filtering system that can predict unknown user ratings based on the available historical ratings given by many users.

There has been many algorithms established for constructing Recommender systems. We were free to use any of them that would create the best recommendations for users of Yelp. The best Recommendation System is the system which most accurately predicts how users will rate businesses in cases not seen by the algorithm. We were also free to use any programming language, however we were advised and encouraged to work with Python. Python is widely-used and has many built-in libraries which are very convenient and easy to use in data science.

APPROACH

Our original approach was to use only rating data, ignoring any extra information like date and review text. We tried every algorithm from the open source recommender systems library Scikit-Surprise, written in Python. However we could not achieve a good accuracy of our Recommender System. This motivated us to try utilizing the text review data.

For each training example, there is an accompanying text review that the user has written about the business. By inspecting these reviews we discovered that there are many types of businesses, not just restaurants. We checked on Yelp.com to see the different types of businesses available. We found the predominant ones to be Restaurants, Hotels, Automotive, and Beauty/Salon. Therefore the first thing was to clean our data by clustering and separating different services from each other. After that we could create different models for every service separately to be able to predict ratings for them.

Once we had four separate train / test sets based on business types, we again began testing RS algorithms on the sets. At this point we continued to have poor results. We discovered that the whole time, we had been working with only about 1/6th of the train data. We assume a faulty download had taken place. Once we had all the data, we began to get much better results using algorithms from surprise on the entire dataset.

In order to tune faster with better result, we want to simulate the test set. We do this by finding the ratio between test set and train set, which is 7.75%. Then we split our train set into 1/13 (13 fold). Since it takes extremely long time to do all 13 folds, we will stop in 2 step (or chosen split steps) to stop the loop and average the results. We then choose multiple parameters and run parallely and score them using this testing technique. If one param get better result, we then can run more steps to ensure it is not false positive.

We then started achieving highly competitive results by just tuning the SVD matrix factorization algorithm provided by surprise. We continued to try additional techniques. We tried again splitting the datasets by business type as before, but this did not yield better results than training SVD over the entire dataset. We suppose that business type may be one of the latent

factors that is learned via the matrix factorization. We also tried ensembling models by averaging predictions scores, and also taking weighted averages of predictions, weighted by model performance. No techniques performed as well as simply tuning and training SVD over the entire dataset.

TECHNIQUES USED AND CHOICES MADE

To split the train data by business type, we first joined the train data and review data by train_id. Then, for each train example, we use the Natural Language Processing Toolkit NLTK, a Python library, to tokenize the review, and check it against a list of keywords for each business type. For example hotel keywords included ‘hotel’, ‘sleep’, ‘bed’, ‘accomodation’, ect. From this technique we found that the data was roughly 70% Restaurants, 20% Hotels, 5% Auto, and 5% Beauty.

To separate the test data by business type, we first classified each business (item) by its business type. To do this, we went through each train set, if a particular business appears more than twice in a single train set, then we assign the business to the business type matching that train set. Since there are many more train examples than businesses, businesses will always show up more than twice in at least one train set type. Other than data exploration and clustering, our work horse was the surprise library.

Surprise gives perfect control over experiments. It provides various ready-to-use predictions algorithms such as baseline algorithms, neighborhood methods (k-NN), matrix factorization-based (SVD, PMF, SVD++, NMF), and Slope-One, and Co-Clustering. Also, various similarity measures (cosine, MSD, pearson…) are built-in. Surprise makes hyper-parameter tuning easy. To do that we used GridSearch class provided by Surprise library. Given a dictionary of parameters, this class exhaustively tries all the combination of parameters and helps get the best combination for an accuracy measurement.

We found that the SVD algorithm far outperformed the others and we began pursuing the optimization of the hyper-parameters for this algorithm. We found that a key parameter setting with SVD was the number of factors ie. the dimensionality of the latent feature matrix. Setting this parameter in the

range of 4-6 gave best results. This is interesting as it suggests that there roughly 5 key factors that determine how a user will rate a business.

CONCLUSION

Today there are libraries available for most Data Mining problems. The challenge is no longer to implement complex algorithms, but rather to understand them so that you are able to use someone else's implementation to its full potential.

Matrix factorization is indeed a very powerful tool for building recommendation systems. We believe that results could still be improved by finding optimal hyper parameters for other algorithms and using an ensembling technique.

When obtaining poor results, always question your data first. We were working with 1/6th of the train data for most of the semester. We tried so many complex techniques to improve our score, when all we needed to do was investigate our data source. This was a very valuable lesson that we will remember for all time.

REFERENCES

http://surprise.readthedocs.io/en/stable/index.html

http://surpriselib.com

http://www.nltk.org/

https://www.yelp.com

THE CODE

src/ensemble.py

In this file we train every algorithm from surprise, cache the score to be compared in the future, cache the model, and cache the predictions if they produce a better cross validation score than the best cached for that model. It also contains code to average over predictions to produce an ensemble prediction.

src/ensemble_split.py

Similar to above except trains a separate model for each of the four datasets split by business type. And then produces predictions of each example in the test set, using the appropriate model for that business.

src/svd.py

In this file we make it easy to iterate through parameter choices with SVD. We found this manual approach slightly more efficient than working with grid search. This is because of the way we were saving scores and best parameter settings.

src/parameter_search.py

Here we experiment with surprises grid search over parameters.

/notes.py

This is some exploratory python code. We are counting the number of train and test examples, number of businesses, number of users ect. This is also where we join reviews to train data and filter by keywords

data/sep_data.py

This file separates the train and test data into four separate datasets based on business type. This file got deleted when wiping and re-cloning our repo since the data directory was in our gitignore.

RESULTS

We managed to get the best RMSE from SVD with parameters as follows:

SVD: 1.29319

```
n_factors=4,
 n_epochs=27,
 init_mean=0.0445,
 init_std_dev=0.043,
 lr_bu=0.008,
 lr_bi=0.009,
 lr_pu=0.007,
 lr_qi=0.005,
 reg_bu=0.41,
 reg_bi=0.21,
 reg_pu=0.21,
 reg_qi=0.21,
```

Ensemble Weighted Average: 1.29324

SVD over separated data: 1.38914