SI650 Final Report
Learning to Rank on the Yelp Dataset

Brad Schwartz[1] and Yash Bhalgat[1]

[1]Department of Electrical Engineering and Computer Science, University of Michigan

December 15, 2017

1 Introduction

With the growing start-up culture, and the ever-growing trend of companies relying on the internet to generate foot-traffic, systems for ranking businesses against each other have risen in popularity. Foremost among those is Yelp. Yelp specializes in crowd-sourcing their data - users come to Yelp to rank, review, and discuss the different businesses they frequent. This crowd-sourcing generates massive amounts of data, and so Yelp must stay on the cutting edge of machine-learning, deep-learning, and ranking technologies in order to make rankings for all of the businesses reviewed on their site.

The data they collect is not limited to just the meta-information about businesses, like where they are located, when they are open, and the types of services they offer. Users will vote on how "cool" and "funny" businesses can be, submit their times when they most frequent places, and even captioned photos[1]. This presents interesting tactics for how to rank businesses. Should more weight be placed on users given rankings of any place, or do their reviews matter even more? Can a cool restaurant still be generally a poor company? These types of questions are just a few that need to be addressed when setting up a ranking system.

Of course, Yelp is just like any other company, and is interested in making money! This means that they do not release much about how and what they take into consideration when it comes to ranking. This leads us to our exploratory paper - can we learn to rank businesses too?

2 Methods

2.1 Dataset Collection

Fortunately, Yelp publishes a small portion of the data they have, as a part of their Yelp Open Dataset Challenge[1]. Their challenge encourages people to explore the data they provide, with some ideas to try out, such as photo classification, natural language processing and sentiment analysis, and exploring the relationships between companies and businesses through graph mining. The data they provide is nearly 5 gigabytes in size! It includes nearly 4,700,000 reviews and 200,000 pictures, for 156,000 businesses across 12 metropolitan areas.

The data can be obtained and accessed in two different ways - SQL tables, or JSON formatted items. We chose to use the JSON style formatting, where each item represents a single review, photo, or business. This format was chosen due to its ease of use with the Python programming language.

2.2 Data Pre-Processing

2.2.1 Nested Data

A significant challenge of any machine learning problem is getting the available information into a suitable format. The initial problem here is that JSON allows for nested attributes. For example, a business attribute could be the hours they are open, which can be further sub-divided into days of the week, and then hours. Intuitively, this makes perfect sense. Programmatically, we face the issue of trying to do text analysis on list or dictionary type objects. The best way around this was to *flatten* the information. Take all the data about business hours on Mondays, and make it into a top level object `hours.Mondays`.

2.2.2 Missing Information

With that out of the way, we have the curse of crowd-sourcing to deal with - People are lazy, and that means they don't fill out all the fields asked of

[1]https://www.yelp.com/dataset/

them! What this leads to is an extremely large portion of fields being empty. When we read in the data and start to work with it, these missing fields appear as "Not a Number", or *NaN*s.

There are many established ways of handling *NaN*, which the Pandas documentation[2] explores in light detail. While *NaN*s can be worked with in their raw form, it typically makes more sense to try and remove them. Options exist for forward or back propagating previous information. This means that we take the previous or next valid data chunk and use them to replace the current occurrence of missing information. Another method Pandas brings up is to just fill with scalar values, such as all 0's. A smart way of handling this is to do interpolation[4]. If we can take a few indicator values and essentially do a regression, we can then interpolate as to what these missing values would be. Of course, the problem here is still that user supplied information is not easily interpreted, and this makes regression without manually picking values difficult.

In the Yelp dataset, we saw that in addition to a large number of *NaN*s, they were widely distributed. This made it nearly impossible to just drop out rows that had missing values, as that was nearly all of the data! We opted to fill missing values with 0's or do nothing, depending on what was being looked at.

2.3 Linking Companies

Since Yelp offered the data in both SQL and JSON formats, it is easy to see why they had separated the JSON files into 6 separate files - they were treating them as relational database tables! While SQL provides natural ways of merging tables, we had to get creative with Python. Fortunately, this is one of the main reasons the Pandas library for Python was created. It allowed us to load and treat the data as relational tables, and even provided handy functions for merging tables into one. Each business is indexed by some unique string, and so we are able to do SQL-style joins on this attribute. At the end of a series of merges, we are able to have a single table that relates a business and their information, to any photos, reviews, and other meta-information linked to them.

2.4 Pipeline

Below, we show the general outline of our methodology. Despite looking at different features at different points, it can still be generalized to this outline. We start with our raw JSON files, loading and flattening them as described above. This table view is fed directly into some type of vectorizer. We were focusing on text information, so this was typically always a Tf-Idf transformation. Once we had numerical feature vectors, a form much easier to work with, we would move onto the feature extraction selection stage. This step helps cut down on the amount of data we need to deal with, by focusing on just the important features. From here, we have all the data we want to look at, in a form we can use. So we train some sort of classifier, and then start making predictions about businesses.

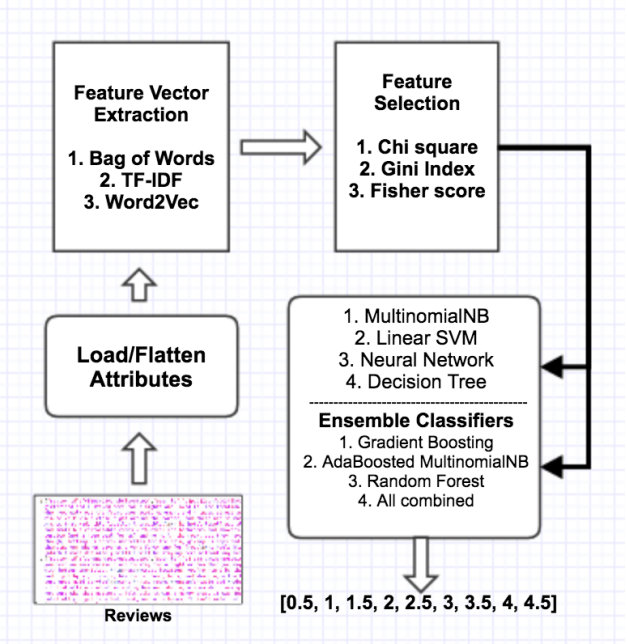

Figure 1: Graphic describing data pipeline and architecture

2.5 Evaluation

For the purpose of treating this is a classification problem, we focus on the *stars* attribute. This feature appears for both businesses as an aggregate, and as a feature on user reviews. For businesses, *stars* are a function of user ratings, reviews, site clicks, and more. The user *stars* are the values that a user has decided to rate that business. Higher stars correlate to a better business and experience.

In general, what we are trying to do is learn to predict what these *stars* should be. Given all the information about a business, can we predict what its overall *star* value should be, and for individual user given rankings, can we predict what they rated a business?

3 Exploration of Data, Machine Learning, and Ranking

3.1 Investigating the Effects of Data

One of the most practical issues when dealing with any kind of machine learning or big data application is training time. Of course we can spend hours training on all of our data and achieve a high accuracy, but at what cost? One cost is over-fitting. If we use all of the data at our disposal, depending on our model, we may only learn on how to classify our exact body of information. The other cost is time, which we will extrapolate into cost - cost of running machines, cost of not having an active application, etc. Here, we explore the effect of using different types of data, as well as different amounts. This data was gathered by manually alternating combinations of files tried, and rerunning the program, and the `time` Unix command was used to time the program execution. The text is vectorized using a Tf-idf transformer, and a Multinomial Naive Bayes classifier to do the actual predictions. Here, our goal was to predict the raw score of any given business, off of all information provided by the files.

As could have been expected, the best performance was achieved when all the data was used, but also the longest training time by a significant amount. What's interesting to note is how little the raw meta-information tells us about any given business. The Checkin file caused the largest jumps in training time. Despite using 10,000 lines for each dataset, the Checkin file is still multiple times larger, so this increase in time makes plenty of sense.

If we assume each line for any file has the same amount of information as any other line, then 10,000 lines is about 6% of the business file, 7.4% of the Checkin file, 0.2% of the review file, and 1.0% of the tip file. It's easy to see from how little of the data we used here, that training time could become prohibitive if we just used the data provided to us by Yelp. If they have on the order of terabytes, it could take a very long time.

3.2 Investigating Different Classifiers

Here we look at the accuracy (and other scoring methods) of different types of classifiers. At this point, the same data is used as above, with all files used. Note that the training for the neural network had to be interrupted for this portion. As the number and depth of the layers increases, the training time increases by an incredible amount.

Each of the above classifiers can loosely be described as a standalone learning algorithm. This is in contrast to "ensemble methods", where multiple classifiers are trained, with a majority vote style system to predict the class of a given example. We explore these next.

3.2.1 Ensemble and Boosting Methods

What is most interesting about these ensemble methods is how closely their summary statistics resemble their respective standalone classifiers. The only classifier to do better than their original one was the Random Forest classifier, which performed less than 1% better. The AdaBoosted Multinomial Naive Bayes classifier performed much worse in all regards, and may show that ensemble methods can be just as susceptible to over-fitting as other methods.

The only method that was wholly original was the final classifier. It was created by using Scikit-learns *VotingClassifier*, an object that takes any number of classifiers, and uses each as one of the possible voting methods. This classifier really highlights the benefits of ensemble methods. While its accuracy is only around the average of the individual classifiers accuracy, it is easy to see how if we heavily tuned each of the original estimators, we could greatly improve the accuracy of the Voting Classifier.

3.3 Ranking

Of course, we can't just explore machine learning and training times. We also try at our own ranking system. Here, the idea is to predict the rankings given by users to businesses, based off how other users ranked businesses. We look only at the text of a users review, and so this classification problem could also be considered a type of sentiment analysis. That is, the higher ranking we see, that user probably included more positive words, and vice versa.

In previous exploratory analyses, we collected and joined together all information about a business that we could, before predicting the score for that business. Now, our method is to predict user given rankings, and aggregate those to rank the business ourselves. While it may be simplistic, we just take the mean of all predicted values. The idea is to add some stability to the ranking process. If a user had given a low rank but a poor text review, they may have just required more granularity in the voting process.

Following much the same process as before, we load the files we are interested in first. Here, it was the business file, to know what the actual rankings are, and the user reviews, which are what we actually

Files Used	**Training Time** (s)	**Average 3-Fold CV Score**
Business	7.826	0.253
Business, Tip	12.241	0.6
Business, Review	16.456	0.6
Business, Checkin	22.768	0.609
Business, Tip, Review	21.023	0.724
Business, Checkin, Tip	30.668	0.717
Business, Checkin, Review	36.327	0.725
Business, Checkin, Review, Tip	44.319	0.781

Table 1: Comparison of Files, Training Time, CV Accuracy

Ensemble Classifier	# Estimators	**accuracy**	**f1-score**
Gradient Boosting (Logistic)	10	80.406	0.779
AdaBoosted Mutlinomial NB	10	73.074	0.617
Random Forest	10	80.087	0.794
All Original (above) Ensemble	8	81.0266	0.794

Table 2: Ensemble Classifier Comparison

Classifier	**accuracy**	**f1-score**
Logistic Regression	80.882	0.791
Linear SVM	81.285	0.806
MultinomialNB	78.930	0.755
Decision Tree	79.581	0.794
Neural Network (5,2)	73.074	0.617

Table 3: Standalone Classifier Comparison

want to train on. Since the files are unordered, we take the intersection of the businesses the two loaded datasets have in common. The businesses they have in common we hold out for testing, and use everything else as training data. Even just using a simple linear SVM, we find that we don't do that poorly!

We have to infer the rankings of the actual businesses, as the Yelp scoring system is only granular to whole integers. Fortunately, the number of reviews per business is included, so we sort them first by stars, and then by the number of reviews they have had.

For our own system, we train on the all the reviews for businesses not related to the ones the files had in common. We then predict the ranking a user would have given to a business, and take the average per business. Since we cannot assume that the number of reviews will be proportional to the ones we have in memory, taking the average allows us to rank on a finer scale than just integers allow.

Above, we see some information on *how* we measured our rankings. The left shows the average position offset of our rankings. What this means, is that for each of our rankings, we found the distance it was to its true ranking. It was expected that as the number of businesses we ranked went up, so would our

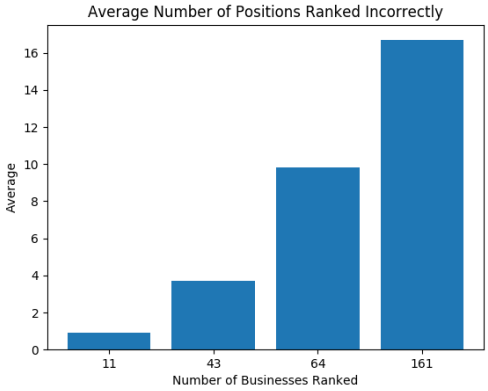

Figure 2: Mean ranked position offsets

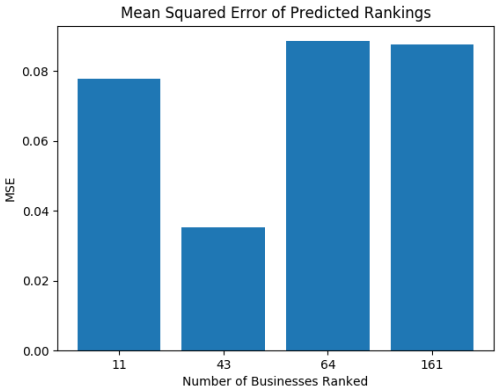

Figure 3: MSE of mean predicted rankings

average, but the largest value is still not as large as might be expected. This can be attributed to us only missing the true position by a smaller amount. Our ranking algorithm might not be the same as Yelps, we can somewhat approximate their system.

The right image shows the Mean Squared Error, calculated as:

$$MSE(T, P) = \frac{1}{N} \sum_{i=1}^{N} (T_i - P_i)^2$$

where P and T are the prediction and truth rankings, corresponding at their ith index, and not where the business id's match up. Where the average ranked offset could tell us how we did on the actual ranking portion, now we see how well our ranking system worked. Typically, MSE is used to to see how well some estimator predicts some parameter, and so we lose a little of the interpretability by having a full system and not a straight predictor model. Despite this, we see low MSE scores across the board. This indicates that for the most part, our predicted stars for businesses was pretty similar to what the actual value was. Due to differences in the granularity of our system and Yelps, we may see a small increase or decrease if we were to truncate our values after the ranking process.

References

[1] Wei-Hong C. How we use deep learning to classify business photos at yelp.

[2] Panda's Documentation. Working with missing data.

[3] Phil Rozek. Yelp ranking factors, August 2012.

[4] Jeff Sauro. 7 ways to handle missing data, June 2015.

[5] Rui W. Learning to rank for business matching, December 2014.