Probabilistic Restaurant Search Engine using Classification of Yelp Review Comments

Oliver Melgrove[†,‡]
[†]University of California, Los Angeles
[‡]melgrove@ucla.edu

2022-12-01

Abstract

Search engines produce results from a domain of interest given a user supplied query. Restaurant recommendation systems must connect the wanted attributes of users to the actual attributes found in restaurants. It's hard to measure the quality of a recommendation, since quality is defined by a somewhat intangible human feeling of closeness between the wanted and recommended attributes. In this paper, we aim to maximize restaurant search result quality probabilistically using existing Yelp reviews, and user defined desired restaurant criteria. We expose the search engine as a web application so that the users' queries can be input into the model over the internet, allowing for real-time recommendations based on arbitrary criteria. Later in the paper we discuss how our feature extraction and model selection processes are tailored to the problem statement, as well as considerations for model improvement.

1 Problem Statement

At its core, we would like to relate a query and the attributes of the different restaurants in our dataset, and then rank those restaurants based on a measure of closeness to the query. In our dataset we have reviews that people have written about restaurants they have been to, and from the user of the search engine we have a dynamically generated description of a desired restaurant. We observe that when someone describes the restaurant that they would like to go to, that is similar to when someone describes a restaurant that they have already gone to. Thus, we will formulate our problem as one of predicting the likelihood that the search query is an unseen review of a restaurant, based on that restaurant's existing reviews. We achieve this by encoding review comments as vectors, and training a classification model on the encoded reviews. Restaurant attribute desiderata are then supplied by the user, which the model uses to predict the restaurant. We discuss the assumption that the desired attributes supplied by the user are somewhat similar in structure to review comment text later in the paper.

2 Pre-processing

In order to make a good prediction, we would like the restaurant review data to be as similar as possible to the search query. At their core they are similar: both describe attributes of a restaurant. The most relevant difference is that review data is distinguished by sentiment, while the search query is not. Nobody is using a restaurant search engine expressly to find bad food. At the same time, a review that is purely sentimental, like *"The food was good"*, does not offer any predictive value for the search.

2.1 Feature Extraction

We aim to extract information from the reviews and the query such that we only include useful data for prediction. To increase the comparability between reviews, we clean the dataset of irregular punctuation, symbols, and words. We also lemmatize the existing words for the same reason. To minimize the aforementioned difference between the two data types, we remove all stop words. We think that this will emphasize qualitative words that are important for prediction, and de-emphasize sentiment. We think that short reviews are more likely to include only sentiment information, so we removed all reviews with less than 10 words after processing. Finally, we consider how easy it will be to predict a restaurant from a given amount of reviews. We removed all restaurants with less than 10 validated reviews, since we think that's a reasonable limit for the amount of information needed to identify a restaurant. Our processed dataset includes 1036 restaurants and 52871 reviews.

2.2 Text Embedding

We chose to embed the reviews in our dataset as vectors so they can be processed efficiently by popular machine learning (ML) models. We decided to use a pre-trained GloVe [1] embedding because of its semantic similarity property. We chose the `glove-twitter-50` embedding dataset because we felt that the formality of discussion on Twitter would be more similar to that of restaurant reviews than that of other popular datasets like Wikipedia and Common Crawl. To get a single vector from a review with many words, we took the element-wise mean of the word embedding vectors of all of the words in the review as $\frac{1}{n}\sum_{i=1}^{n} \text{GloVe}(\text{word}_i)$.

3 Prediction

Our prediction model must rank restaurants based on a measure of similarity to the user supplied query. We achieve this by training a classifier on every review, with each class being the restaurant that the review belongs to. We then use the classifier to calculate the probability that the query is a review from each restaurant, and return the ten most likely restaurants in the search results. Since we only return the top ten most likely results, and don't return a variable number of results based on a measure of the absolute likelihood, the precision vs. recall tradeoff is simplified. Recall is essentially fixed, since we always return the same amount of results, and we aim to maximize precision.

3.1 Model Selection

We chose the XGBoost [4] model for our task because of its popularity, quality of documentation, and quantity of parameters. We tuned the parameters such that it performs classification instead of regression, and outputs a probability instead of a single class.

3.2 Model Training

Since there are many restaurants with very similar reviews, we expect that the accuracy of the classifier will be low. That said, we expect that it will still offer good predictions, since there are many recommendations for a single prompt that are defined as good in the eye of the user. This intangible definition of good can be understood as the *real* loss function for our model, so even if the minimized loss function for the classifier is not very low, the model may still be performing well.

XGBoost compute time scales quadratically with number of classes, so due to the large number of classes, model training proved to be very slow. We chose a max tree depth of 6, a learning rate of 0.15, and multi-log-loss as the evaluation metric. We chose softprob as the objective function instead of softmax because we need the top ten highest likelihood results, instead of just the highest. We performed early stopping at 9 iterations, since the model was overfitting past it. If we had more compute resources, we would have chosen a lower learning rate of 0.1 or 0.05, and added an ℓ_2 regularization parameter to reduce overfitting. We also would have increased the maximum tree depth to compensate for the very large number of classes.

4 Search Engine Interface

Now that we have trained our model on the Yelp review data, we must construct an interface so that the search engine can be used by end users. We implement this as a website that allows users to enter their search query, and displays the search engine results back to them. The search engine is publicly accessible here.

4.1 Prediction Pipeline

To generate a prediction, the website must process the query and pass it into the trained XGBoost model described in section 3. We implement this with a Python web server exposed as an API over the internet which the website makes HTTP requests to. The server performs the same pre-processing step described in section 2, except this time on the user provided query in real-time. At this point the query has the same 50-dimensional GloVe representation as the Yelp review training data, and can be passed into the trained XGBoost model for prediction.

After prediction we encounter a problem, which is that the labels that our model predict have no tangible meaning to the end user, they are just integers that represent business ids. We first convert the integers into the business ids that they are a proxy for. At this point, to convert the business ids into something useful, we use the Yelp Fusion API [3] to fetch business information with the ids. This business information is then returned in the HTTP response body, and displayed to the user in the website.

4.2 Software Engineering Considerations

Deploying an ML system to production requires another layer of complexity on top of creating the ML model itself. Some of the challenges are within the field of software engineering, while some are specific to ML deployment [2]. Our prediction service is a continuously running web server that must be able to perform the following steps whenever an HTTP request is received:

1. Pre-process the user query
   (a) Perform rule based formatting with regex and casing
   (b) Lemmatize words using the `wordnet` and `omw-1.4` datasets
   (c) Remove stop words using the `en_core_web_sm` dataset
   (d) Encode words using the GloVe `glove-twitter-50` dataset
2. Predict classes using the pretrained XGBoost model
3. Request the Yelp API to convert the predicted classes to human readable data

The large datasets needed for pre-processing pose a serious problem for a continuously running service: collectively they are far too large to store in memory. This means that we are not able to simply load them in and reference them like we did when training the model. After some investigation, we concluded that the `glove-twitter-50` dataset was by far the largest, and if we could find a way to not store it in memory we could afford to store the other datasets in memory with a modestly sized machine (2GB of RAM). We solved this by loading the GloVe embeddings into a key-value database, and connecting to that database from the server. This makes the embedding lookups significantly slower than if they were in-memory, but still fast enough for our needs since we only need to use the lookup for the number of tokens in the user request, and the time complexity for each lookup remains *O(1)*.

The algorithm runs quick enough that it doesn't deter repeated use. We think that the majority of the response time is spent waiting for the requests for the GloVe embeddings from the key-value database, and waiting for the Yelp Fusion API requests to complete. Both of these are done synchronously for each of the tokens in the query and each of the ten search results respectively, meaning that the server waits for each request to come back before it sends the next. We posit that by batching these requests, and asynchronously waiting for the responses, our server response time would be at least twice as fast.

5 Discussion

We are somewhat satisfied with the quality of the results. It seems to have an affinity to recommend specific restaurants across many different searches, leading us to believe that there is bias in the model that can be reduced.

5.1 Prompt Engineering

Choosing the correct prompt has a profound impact on the results. Popular generalized search engines like Google perform well on very brief prompts because they have auxiliary information to make inferences upon like location and previous browsing activity. Our model treats search queries as if they are reviews, so we would expect the best results when queries are longer and more detailed. Conversely, our embedding strategy combines the information from all of the tokens in the query into a single vector, so queries with a single token shouldn't have any less information. We find that this is the case, since long queries that closer resemble reviews appear to perform no better than queries that are only a single word. Queries with words that are very specific to a single type of restaurant perform the best, which is an expected result. For example, the query *"beer bartender martini music"* returns bars, breweries, or wineries in 8 out of the top 10 results, which is a comparatively high accuracy result.

5.2 Embedding Strategy

A notable result is that when using queries that include a location, like *"German food"* or *"Chinese food"*, the top recommendation is *Santa Barbara Zoo*. We posit that this is because reviews for the zoo include many references to countries and continents, since that is part of the name of many animals. In the GloVe embeddings, all countries and continents are very close in the vector space because they have similar semantic meaning in the context of the twitter dataset it was trained on. Thus, when someone uses a country or continent to describe a cuisine, it closest matches reviews that also include countries or continents, even if they aren't the same continent. This is obviously bad for predictive performance, since we expect there to be a large difference between, say *German* and *Chinese*, but our embedding strategy does not reflect that. A potential solution is to train new GloVe embeddings ourselves using a dataset that better distinguishes words for our use case. One possibility is to train the embeddings on our Yelp review dataset, or another dataset that involves food.

Another consideration is the fact that we have combined all of the words in each review into single vectors. This clearly removes a lot of the information of each review, since highly specific words are diluted by the rest of the words in the review. Thus, longer reviews tend toward the center of the vector space, which is not a desired result. Using a higher dimensional word embedding may improve this, since the vector space becomes larger and dilution becomes less pronounced. Another possibility is to choose a different strategy for combining the words in each review.

References

[1] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.

[2] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. 2022. Challenges in Deploying Machine Learning: a Survey of Case Studies. ACM Comput. Surv. 1, 1, Article 1 (January 2022), 29 pages. https://doi.org/10.1145/3533378

[3] Yelp Fusion is an REST API that allows access to businesses on Yelp. https://fusion.yelp.com/

[4] Chen, T., Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785