Lauryn Fluellen
NLP Final Report
12/11/2022

One of the most utilized features that I rely on day-to-day is auto-suggest. Whether I am writing a text or searching for something on one of the many websites or search engines available, I rely on the software to know what I want to say without me having to put forth all the energy to type in the entire query. I thought it would be interesting to test out my ability to create an auto-suggest model while exploring the performance of different n-gram model sizes on the same data. I thought it would be interesting to do this to answer the question of what effect the length of the n-gram model have on the quality of the model. Before we continue, to define an n-gram, it is a sequence of n words, where n is a positive integer that is determined by the user. For example, if n is three, then the n-gram model will consider sequences of three words at a time. Do the predictions get better or worse with creating longer n-grams? Also, do they get better or worse with creating shorter n-grams? Additionally, getting a better understanding of how people chose the correct size for their models.

The datasets used to train and test the models were gathered from an online platform called hugging face. Here I was able to find a dataset of Google queries that I used to train the model. This dataset called “google_wellformed_query” was created by crowdsourcing wellformedness annotations for 25,100 queries from the Paralex corpus. Every query was annotated by five raters each with a 1/0 rating of whether or not the query is well-formed. This dataset contains English queries that appear to be complete in nature. Unlike the queries that I found in the Yahoo! query dataset, all of these queries are complete sentences or phrases. The dataset does not contain user information. The dataset used to test the models was also gathered from the online platform hugging face. I found a dataset called “jordane95/trec-dl-2019-query” which is a set of 200 complete English queries used to test the model.

The reason I chose to use the “google_wellformed_query” dataset to train my models instead of the original Yahoo! query dataset because I wanted to make sure that the model was trained on complete queries that you would see in a search engine when the search button was clicked. While this set does have less data than the original one the number of complete useable queries ended up being less than the total set of the wellformed Google query dataset. The Yahoo! dataset had more fragmented queries so the model was being trained on incomplete sentences and when it was tested on query sets of complete sentences it performed badly. To ensure this performance didn’t continue I found a data set that was complete, had minimal spelling errors or commonly made spelling errors, to ensure that when tested it would perform better.

To implement the n-gram model for word suggestion, the first step was to collect a large amount of training data. This was done using a corpus of text from Google queries. The next step is to pre-process the text to make it suitable for training the n-gram model. In this case, I chose to lowercasing all the words and tokenize the text. Tokenization involves splitting the text into individual words or units of meaning. Additionally, I wanted to make sure unknown words were handled accordingly when they were found in the data. To do this I made sure to go through the

data and calculate the frequency at which a word accrued and if it was less than the set expected frequency, which for this assignment I chose the frequency to be at least one. If the word frequency is greater than the expected frequency for the model then it is appended to our list and if the word does not appear at a frequency greater than the expected frequency threshold then “UNK” is added to the list in its place. Now that the data has been tokenized and pre-processed we are ready to train the model.

In the training stage, I wanted to make sure that my model was able to take in any data and output n-grams of varying sizes to compare their suggestions. To ensure this was possible, I had to ensure my model was highly flexible. Thus, the user can tell the model the size of the n-gram they want. Implementing this allowed me to use the same logic for all n-grams and only have to change the number of n-grams created during the training process. Now that we know we can make an n-gram of any size let's jump into the training process.

The training process involves calculating the probabilities of all possible n-grams in the training data. To calculate the probability of an n-gram, we simply count the number of times it appears in the training data and divide by the total number of n-grams. After the n-gram model has been trained, it was used to make predictions about the next word in a sentence. To do this, the model takes the previous n-1 words in the sentence and looks up the probabilities of all possible n-grams that start with those words. The model then predicts the next word to be the one that has the highest probability. For example, if the previous n-1 words are "I like to eat", the model might predict the next word to be "pizza" if the n-gram "I like to eat pizza" has a high probability in the training data. This process can be repeated to suggest multiple words for word completion or prediction tasks but for the case of this project, I focused just on the prediction of one word to emulate the functionality of a typical auto-suggestion system.

To illustrate the results my program returns the top five suggested words. These words will be in order by their priority so the first listed word will have the highest probability of occurring and the fifth will have the lowest. I chose to show the top five suggested words for ease of visibility but I designed my model to be flexible and have the ability to produce more suggestions easily if the user wants more. To test the model I chose to train four n-gram models: unigram, bigram, trigram, and an n-gram model with a size of four. These models were then used in the auto-suggest system by iterating over the words in the vocabulary and calculating the probability of those words appearing after the text given as context. The text context is represented by a tuple of the last n-gram before the suggestion is passed in. So for example, if I was looking for the suggestion of the sentence “How many words do you know” in a bigram I would be looking at the prior context (“do, you”) to find the probability and hopefully “know” will be included in the list of suggested next words.

The results from this showed that with the test set provided the different n-gram sizes performed differently across the 200 test queries. While this was the expected outcome I was able to visualize each query's performance across n-gram sizes to find a pattern. I found that typically the unigram and n-gram models did not perform well. Often the unigram did not have the correct word that we expect to be suggested based on the test sentences and the n-gram would be empty because of the length of the sentence.

As shown below in Figure 1 you can see the unigram either does not have the word we are looking for in the list of suggested words or if it is it is a lower probability than another:

```
Sentence: ['how', 'many', 'people', 'use', 'google', 'in', 'one']
Word we're looking for: day
Unigram model suggestions: (['septillion', 'day', 'what', 'slows', 'down'],)
Bigram model suggestions: (['septillion', 'day', 'what', 'slows', 'down'],)
Trigram model suggestions: (['day', 'what', 'slows', 'down', 'the'],)
Ngram model suggestions: (['day', 'what', 'slows', 'down', 'the'],)

Sentence: ['what', 'format', 'does', 'a', 'thumb', 'drive', 'need', 'to', 'be', 'for', 'a']
Word we're looking for: mac
Unigram model suggestions: (['nosebleed', 'synonym', 'captain', 'face', 'supplement'],)
Bigram model suggestions: (['mac', 'what', 'slows', 'down', 'the'],)
Trigram model suggestions: (['mac', 'what', 'slows', 'down', 'the'],)
Ngram model suggestions: (['mac', 'what', 'slows', 'down', 'the'],)
```

Figure 1

In Figure 2 you can see the n-gram does not have the word we are looking for in the list of suggested words:

```
Sentence: ['who', 'plays', 'dr', 'fell', 'on', 'vampire']
Word we're looking for: diaries
Unigram model suggestions: (['diaries', 'what', 'slows', 'down', 'the'],)
Bigram model suggestions: (['diaries', 'what', 'slows', 'down', 'the'],)
Trigram model suggestions: (['diaries', 'what', 'slows', 'down', 'the'],)
Ngram model suggestions: ([],)
```

Figure 2

The size of the n-gram can affect the performance of the auto suggestion model based on the n-gram approach. A larger n-gram size may provide the model with more context and enable it to make more accurate predictions, but it may also require more training data and computing resources. On the other hand, a smaller n-gram size may require less training data and be less computationally intensive, but it may also result in less accurate predictions. The appropriate n-gram size for a given auto-suggestion model will depend on factors such as the amount and quality of training data, the computing resources available, and the desired level of accuracy.

In seeing this discrepancy of which models are performing well based on the length of the queries I looked deeper into the average length of the queries in the dataset and found that the average length was 5.785. With this information, I wanted to look deeper to see how this average length leads to the performance of the models. I concluded that for my dataset with this average query length, the model that performed the best was the bigram model. This was concluded by going through each test sentence and seeing in what size n-gram models the word we were looking for was suggested. From here I kept a tally. I found that the unigram model suggested the word 151 times, bigram 180 times, trigram 160 times, and n-gram of size four 124 times.

In the future, to further test the different attributes of the model that could be changed and may affect the quality of the model I would like to experiment more with standardizing the length of the queries. I believe if I trained the model on queries that are all the same size I think it would further increase the performance of the model but may not be realistic unless there is a common query length to shoot for. If there is a standard query size to expect in search engines it may be beneficial to train based on that number so the performance when testing would be better. Additionally, I would like to go further and build out a front-end feature that would be able to display this auto-suggestion model in action. Making a search engine-esc mock site that is able to do the auto-suggestion in real time would be interesting to interact with.

Works Cited

Ashraf, Shawon. “Demystifying Autocomplete-Part 1.” Medium, Towards Data Science, 13 Oct. 2020, https://towardsdatascience.com/index-48563e4c1572.

community, The HF Datasets. “Google_wellformed_query · Datasets at Hugging Face.” google_wellformed_query · Datasets at Hugging Face, https://huggingface.co/datasets/google_wellformed_query.

“Lambda and Filter in Python Examples.” GeeksforGeeks, 9 July 2021, https://www.geeksforgeeks.org/lambda-filter-python-examples/.

Li, Zehan. “Jordane95/TREC-DL-2019-Query · Datasets at Hugging Face.” jordane95/Trec-Dl-2019-Query · Datasets at Hugging Face, https://huggingface.co/datasets/jordane95/trec-dl-2019-query.

Load, https://huggingface.co/docs/datasets/loading.

Matalka, Luay. “How to Easily Create Tables in Python.” Medium, Towards Data Science, 21 Nov. 2022, https://towardsdatascience.com/how-to-easily-create-tables-in-python-2eaea447d8fd.

Python Geeks. “Defaultdict in Python with Examples.” Python Geeks, 29 Nov. 2021, https://pythongeeks.org/defaultdict-in-python/.

“Python Lambda Functions.” GeeksforGeeks, 25 Oct. 2022, https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/.

“Python: Random.sample() Function.” GeeksforGeeks, 29 Aug. 2018, https://www.geeksforgeeks.org/python-random-sample-function/.

Tan, Liling. “N-Gram Language Model with NLTK.” Kaggle, Kaggle, 23 July 2019, https://www.kaggle.com/code/alvations/n-gram-language-model-with-nltk.