Text Analytics: 1st Assignment

Tsirmpas Dimitris
Droutzas Vassilis

January 28, 2024

Athens University of Economics and Business
MSc in Data Science

Contents

1 Introduction

This report will briefly discuss the theoretical background, implementation details and decisions taken for the construction of bi-gram and tri-gram models.

The full code can be found at `https://github.com/dimits-exe/textanalytics`. Note that the notebook does not contain the models, which are imported from python source code files in the `src` directory.

2 Datasets

2.1 Original Dataset

For the needs of this assignment, we picked the movie reviews corpus from the NLTK data repository, as well as a hand-picked selection of files from the Gutenberg dataset.

We followed the following Data preprocessing steps:

- We converted the text to lowercase letters.
- We used tokenization in terms of both sentences and words.
- We divided the dataset in 3 sets, the training set (60%), development set (20%) and test set (20%). We used the development set in order to get the optimal alpha value which would be used to find the bigram and trigram probabilities.
- We removed some special characters, such as [ ] ? !

We used a function to get the counters of unigrams, bigrams and trigrams. Regarding the OOV words, in the training dataset, we checked for words that appear less than 10 times. These words were filtered and their value was set to 'UNK'. (OOV words).

We initialized a new corpus, called 'replaced_corpus' , where OOV words are replaced with 'UNK'. It iterates through each sentence in the original corpus ('all_corpus') and replaces words with their corresponding "UNK" value if they are OOV.

To find the vocabulary, we simply iterated the word counter and added all the words that were not OOV. To make sure we did not include duplicates, we converted the vocabulary to a set.

The same process was applied for the development and test sets, except that now we kept the same vocabulary. We updated the sentences with the 'UNK' value when necessary. Finally, we calculated the 20 most frequent words of unigrams, bigrams and trigrams and the vocabulary length, which can be found in the notebook.

2.2 Corrupted Dataset

In order to test the spell checking models, a new dataset needed to be created. We decided to use a manually corrupted version of the combined dataset mentioned above.

Thus, we created a function `get_similar_char()` to define replacements from original characters. For example, a would be replaced by e, c would be replaced by s etc. This function returns a randomly chosen character from those defined.

The function was subsequently used by `corrupt_sentence()`, which takes as input a sentence and returns a new corrupted one with a probability for every character of 0.1 (user-defined parameter).

3 Language Modeling

3.1 Defining the models

Two models were primarily used in the language modeling task, those being the Bi-gram and Tri-gram models. We also include a Linear Interpolation model using the former models for the sake of comparison. The source code for the models can be found at `src/ngram_models.py`.

During fitting, the models simply memorize the different uni-grams and bi-grams, or bi-grams and tri-grams for the bi-gram and tri-gram models respectively, and their occurrences in the training corpus.

During inference, the models predict the next token based on the algorithm shown in Algorithm 1, where `candidates(sentence)` and `ngram_probability(sentence, word)` are defined according to the model.

The `is not` UNK condition makes it impossible to output UNKNOWN tokens without disrupting the distribution of the words as found in the original text (which would have been the case had we, for instance, removed ngrams containing UNK during fitting). Our model thus essentially selects the next best option when UNK would mathematically be the best guess.

The `candidates(sentence)` function is a look-up of the last or the two last words in the sentences in the model's respective bi-grams and tri-grams.

The `ngram_probability(sentence, word)` function for the bi-gram model would be $P(w_2|w_1) = \frac{C(w_1,w_2)+\alpha}{C(w_1)+\alpha \cdot |V|}$, where $w1$ is the last word of the sentence, $w2$ is the word under consideration, $C(w_1,w_2)$ is the bigram count, $C(w_1)$ is the unigram count, $0 \leq \alpha \leq 1$ is the smoothing hyper-parameter and —V— the vocabulary size.

Similarly, the `ngram_probability(sentence, word)` function for the tri-gram model would be $P(w_3|w_1,w_2) = \frac{C(w_1,w_2,w_3)+\alpha}{C(w_1,w_2)+\alpha \cdot |V|}$ where $w1 and w2$ the last words of the sentence, $w3$ the word under consideration, $C(w_1,w_2,w_3)$ the trigram count, $C(w_1,w_2)$ is the bigram count, $0 \leq \alpha \leq 1$ is the smoothing hyper-parameter and —V— the vocabulary size.

The reason we only need to calculate the last ngram in order to predict the next token is simple. Let t_1, t_2 be the tokens under consideration and $w_1 \cdots w_k$ the words of the sentence thus far. For t_1 to be selected over t_2 the total probability of the sentence which includes t_1 must exceed the one which includes t_2. For the bigram model, this is expressed as in Equation 1. Similarly, the trigram model case is explored in Equation 2.

$$
\begin{aligned}
& P(w_1^{k+1}) > P(w_1^{k+1}) \iff \\
& P(w_1| < start >)P(w_2|w_1) \cdots P(t_1|w_k) > \\
& P(w_1| < start >)P(w_2|w_1) \cdots P(t_2|w_k) \iff \\
& P(t_1|w_k) > P(t_2|w_k)
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
& P(w_1^{k+1}) > P(w_1^{k+1}) \iff \\
& P(w_1| < start >, < start >)P(w_2|w_1, < start >) \cdots P(t_1|w_k, w_{k-1}) > \\
& P(w_1| < start >, < start >)P(w_2|w_1, < start >) \cdots P(t_2|w_k, w_{k-1})) \iff \\
& P(t_1|w_k, w_{k-1}) > P(t_2|w_k, w_{k-1}))
\end{aligned} \tag{2}
$$

Algorithm 1 N-Gram model next-token prediction

Input sentence: a list of strings
Output max_token: the most probable string

```
max_prob = −∞
for token in candidates(sentence) do
    if token is not UNK then
        prob = ngram_probability(sentence, word)
        if prob >max_prob then
            max_prob = prob
            max_token = token
        end if
    end if
end for
return max_token
```

Meta tags such as <START >and <END >are appropriately automatically inserted depending on the model. The tri-gram model uses the same tag for its two starting tokens, because of restrictions of the `nltk` library which is used to produce ngrams. Because of this decision, during the probability estimation we ignore $P(word1| < start >, < start >)$.

3.2 Comparing the models

To find the cross-entropy and perplexity, we used the models defined in the .py files with the simple formulas of cross entropy and perplexity in the corresponding functions.

What we needed next was to find the optimal alpha for the probability formulas, as we stated earlier. In ngram_model_alpha_search() we initialize a numpy array to store the entropy values. Iterating the alpha values, we calculate the cross entropy for each alpha. Finally, we keep the index with the best alpha (the one with the smallest cross entropy value). We searched for 1000 alpha values taken from an exponential sequence in the range of $[10^{-10}, 1]$.

4 Spell Checking

In order to obtain the WER and CER scores of a sentence, we imported the `jiwer` package, from which we used the `wer()` and `cer()` functions to calculate the corresponding scores. Then, we just took the average of these scores.