Text Analytics: 1st Assignment

Tsirmpas Dimitris
Drouzas Vasilis

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.

In terms of how we collaborated, Dimitris Tsirmpas constructed the models and their corresponding functions (e.g. auto-complete, the spelling corrector) while Vasilis Drouzas handled data preprocessing (e.g. OOV words) and the evaluation of the models (creating an artificial dataset, calculating CER/WER scores). Both authors collaborated in the process of demonstrating the models' scores (cross entropy, perplexity).

The notebook code and this report were written by both authors. Dataset acquisition and pre-processing was performed by Vassilis Droutzas, while the code for the language models was written by Dimitris Tsirmpas.

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,

```
Vocabulary length:  8302
Unigram's 20 most common words:
(('<UNK>',), 78567)
(('the',), 59979)
(('.',), 53515)
(('and',), 31469)
(('of',), 30052)
(('a',), 29926)
(('to',), 28795)
(('in',), 18168)
(('is',), 17922)
(('it',), 13880)
(('that',), 13531)
(("'s",), 12979)
(('as',), 9820)
(('with',), 9361)
(('he',), 8846)
(('his',), 8656)
(('for',), 8545)
(('was',), 8111)
(('but',), 7148)
(('be',), 6820)


Bigram's 20 most common words:
(('.', '<end>'), 53287)
(('<UNK>', '<UNK>'), 7845)
(('of', 'the'), 6975)
(('the', '<UNK>'), 6803)
(('<UNK>', '.'), 6611)
(('<UNK>', 'and'), 5156)
(('<start>', 'the'), 4991)
(('in', 'the'), 4692)
(('a', '<UNK>'), 3794)
(('<start>', '<UNK>'), 3567)
(('?', '<end>'), 3533)
(('and', '<UNK>'), 3409)
(('<UNK>', 'of'), 3309)
(('the', 'film'), 2789)
(('to', 'be'), 2673)
(('!', '<end>'), 2546)
(('of', '<UNK>'), 2489)
(('<UNK>', 'the'), 2408)
(('to', 'the'), 2387)
(('it', "'s"), 2282)
```

Figure 1: Most common uni-grams and bi-grams in the training corpus.

we calculated the 20 most frequent words of unigrams, bigrams and trigrams and the vocabulary length, which can be found in Figures 1, 2 and 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). An example can be found in Figure 3.

```
Trigram's 20 most common words:
(('.', '<end>', '<end>'), 53287)
(('<UNK>', '.', '<end>'), 6577)
(('<start>', '<start>', 'the'), 4991)
(('<start>', '<start>', '<UNK>'), 3567)
(('?', '<end>', '<end>'), 3533)
(('!', '<end>', '<end>'), 2546)
(('<start>', '<start>', '.'), 1961)
(('<start>', '.', '<end>'), 1957)
(('<start>', '<start>', 'it'), 1907)
(('<start>', '<start>', 'I'), 1634)
(('<start>', '<start>', 'i'), 1488)
(('<start>', '<start>', 'and'), 1148)
(('<UNK>', '<UNK>', '<UNK>'), 1141)
(('<start>', '<start>', 'but'), 1128)
(('<start>', '<start>', 'he'), 1085)
(('<UNK>', 'and', '<UNK>'), 1062)
(('the', '<UNK>', 'of'), 1045)
(('<start>', '<start>', 'this'), 1007)
(('<start>', '<start>', 'in'), 992)
(('it', '.', '<end>'), 982)
```

Figure 2: Most common tri-grams in the training corpus.

```
Original: ['laura baker <UNK> helgenberger <UNK> <UNK> her role with admirable conviction .']
Corrupted: ['lauta baker <UNK> helgemberger <UNK> <UNK> her role with admirabla sonviction .']
```

Figure 3: An example of a ”corrupted” sentence compared to its original.

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 $w1andw2$ 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.

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

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

Algorithm 1 N-Gram model next-token prediction

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

max_prob = $-\infty$
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 $P(word1|<start>,<start>)$.

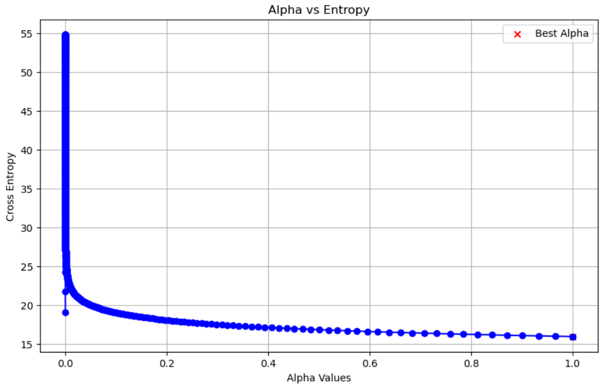

Figure 4: Cross Entropy of the bi-gram language model depending on the values of the α smoothing hyperparameter.

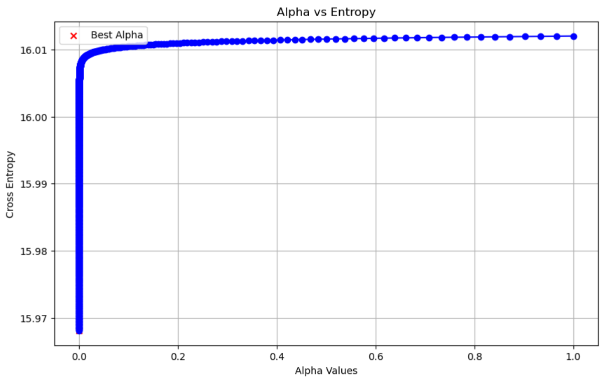

Figure 5: Cross Entropy of the trigram language model depending on the values of the α smoothing hyperparameter.

3.2 Comparing the language 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 $[10^{-15}, 1]$.

The change in entropy depending on the α smoothing hyperparameter can be seen in Figures 4, 5, 6. The Bi-gram model exhibits high variance depending on different α values, where for a very tight region close to 0 (but not too close) result in catastrophic loss of performance,. The tri-gram and linear interpolation models on the other hand exhibit relative stability, with very small α values being slightly favored.

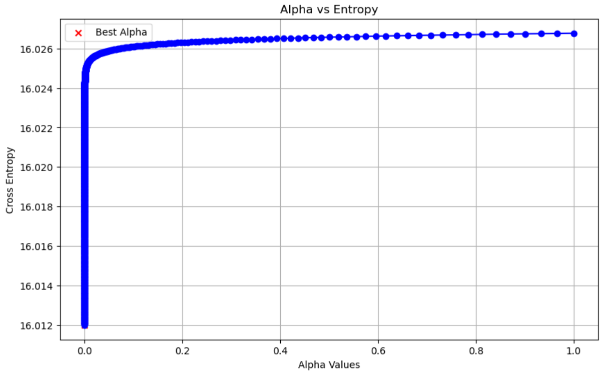

Figure 6: Cross Entropy of the linear interpolation language model depending on the values of the α smoothing hyperparameter.

```
Bi-gram model Cross Entropy: 10.14
Bi-gram model Perplexity: 1130.991608
```

Figure 7: Cross entropy and perplexity scores for the bigram model.

The cross-entropy and perplexity scores of the two models on the test corpus can be found in Figures 7, 8.

4 Spell Checking

4.1 Defining the models

In order to design models capable of correcting spelling mistakes we need to adapt our previous language models to factor in the user-provided (and likely incorrectly spelled) sentence. Thus, we make the assumption that the generated sentence must have a length equal to the user-provided one. As a logical consequence, the model can never predict meta tokens (no START tokens by design, UNKNOWN tokens for the reasons detailed in the previous section, and no END tokens since the predicted sentence's size is known and constant).

We use Beam Search (defined in the file `src/beam_search.py`) to construct the best candidate sentence. This is a generalization of Algorithm 1, where for each new token, we take into consideration the k most probable tokens, where k the beam search width hyper-parameter. Large values of k lead to increased computational complexity but also more reliable results.

We define two models, one bi-gram and one tri-gram spell checking model, which internally use the respective language models defined in the section above. The `candidates(sentence)`

```
Tri-gram model Cross Entropy: 11.49
Tri-gram model Perplexity: 2869.05
```

Figure 8: Cross entropy and perplexity scores for the trigram model.

```
Bigram model: Computing correct sentences for 50 sentences
  0%|          | 0/50 [00:00<?, ?it/s]
Sample original sentence: ['The girl was a daughter of the rich and this police suspicion under which all the poor live day an
d night <UNK> her for the first time into speech .']


Corrupted(wrong) sentence: ['The girl vas a daughter of the ricn amd tnis poiice suspicion under which all tne poor live day a
nd night <UNK> her fot the first time into sbeech .']


Final bi-gram result (corrected sentence): The girl as a daughter of the film and this police suspicion under which all the po
or live day and night on her for the first time into speech .
```

Figure 9: Testing the spell corrector (bi-gram model).

```
Bigram Average Word Error Rate (WER): 0.200
Bigram Average Character Error Rate (CER): 0.224
```

Figure 10: WER and CER scores for the bi-gram spell checking model.

are delegated to said internal models, while the `ngram_probability(sentence, word)` function is defined as $P(w_1^k) = log(P(t_1^k)) + log(P(w_1^k|t_1^k))$ where $P(t_1^k)$ is defined as in the language models and $P(w_1^k|t_1^k) = \sum_i^n \upsilon_i \frac{1}{Lev(w_i,t_i)+1}$, where n is the current search depth, $\upsilon_i = 0$ if the i-th word is the unknown token and 1 otherwise, and $Lev(w_i, k_i)$ is the levenshtein distance between the original word w_i and the candidate token t_i. Thus UNKNOWN tokens are only handled by the language models, with no input from the distance score function.

4.2 Comparing the language models

Regarding the spell correction, we cite an example of our bi-gram model corrector in Figure 4.2.

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. The final results can be found in Figures 10, 10.

```
Trigram Average Word Error Rate (WER): 0.05549206349206349
Trigram Average Character Error Rate (CER): 0.0515449491690643
```

Figure 11: WER and CER scores for the tri-gram spell checking model.