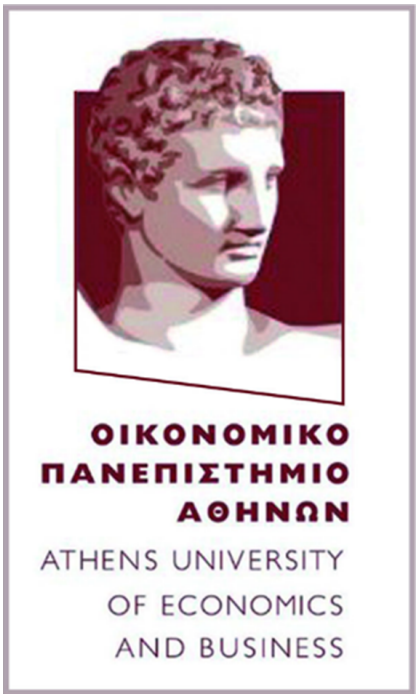
ATHENS UNIVERSITY OF ECONOMICS & BUSINESS

M.Sc. in Data Science



Text Analytics

Instructor: Ion Androutsopoulos

Assignment 01 - Exercises on n-gram language models

Anagnos Theodoros (p3352323) - Michalopoulos Ioannis (p3352314) - Kafantaris Panagiotis (p3352328) - Vigkos Ioannis (p3352326)

25.04.2024

Contents

Exercise 3:

The python script can be found in the link of google colab.

1 (i) Bigram and a Trigram language models

Corpus

We utilized the NLTK library to develop our models using the Gutenberg corpus. We gathered the downloaded files into a single string that contained all the texts.

Pre-Processing

Before we began building the models, we did data cleaning to the text and retain only the words in each sentence. Our pre-processing steps included:

1. **Lower Case:** We used the `.lower()` method to convert all text to lowercase, ensuring uniformity and preventing discrepancies in computations caused by letter casing.
2. **Clean the Text:** We employed regular expressions to eliminate special characters, keeping only words in each sentence.
3. **Tokenize the Input String:** We then tokenized the input string, creating a list where each item represented a sentence from the corpus. We used NLTK's `sent_tokenize` method for this purpose.
4. **Word Tokenize:** For each sentence in the tokenized list, we applied the `word_tokenize` method to divide the sentence into individual words.

Train/Dev/Test Split

To develop our model, we split the input into three segments, using a seed for consistent results and shuffling the sentences:

1. **Train:** We used 60% of the input sentences for training the models.
2. **Dev:** Another 20% of the sentences were used for validation.
3. **Test:** The remaining 20% of the sentences were used to test the model's performance on unseen data text.

N-Grams, Vocabulary, and OOV Words

The construction of our models required calculating the frequency of n-grams and developing a vocabulary. We used padding at the beginning and end of each sentence for our bigram and trigram models using NLTK's `ngrams` function. The vocabulary was compiled using a Unigram Counter, and words appearing less than 10 times were replaced with a special token `*UNK*`. N-grams were computed using the training set, but out-of-vocabulary (OOV) words were replaced across all datasets.

Laplace Smoothing

We used the Laplace smoothing with add-α for our models. For trigrams, this is given by:

$$P_{\text{Laplace}}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + \alpha}{c(w_{k-2}, w_{k-1}) + \alpha \cdot |V|}$$

where $|V|$ represents the number of words in the vocabulary. To determine the optimal value of α, we tested various values and calculated CrossEntropy and Perplexity in the training set:

$$CrossEntropy = -\frac{1}{N} \sum^{bigrams} log_2(P(w_2|w_1))$$

N: Number of bigrams

$$Perplexity = 2^{H(p)}$$

The best values for α for the case of bigram model was and for the trigram was .

We use Laplace smoothing and the dev set as a single sequence of sentences to calculate the n-gram probabilities, by summing the logs. Logs are used as they are more convenient, compared to the product, since the product of many probabilities might result in underfloat events.

2 (ii) Cross-entropy and Perplexity of the Bigram and Trigram models

For this we performed the followed actions:

- Using Laplace smoothing and the development set as a single sequence of sentences, we will calculate the n-gram probabilities by summing the logs. Logs are used because they are more convenient, compared to the product, since the product of many probabilities might result in underflow events.
- We add `start` (or `start1`, `start2`) at the beginning of each sentence, and `end` at the end of each sentence.
- Probabilities of the form $P(\text{start}\ldots)$ or $P(\text{start1}\ldots)$, $P(\text{start2}\ldots)$ will be excluded in the computation of cross-entropy and perplexity.
- Final tuning of the α parameter for α-smoothing.

We calculated the Cross Entropy and Perplexity scores for both models using the test set, which comprised 20% of the total sentences in the corpus, as displayed in Table 1.

Model	**Perplexity**	**Cross Entropy**	α **param**
Bigram	371.386	8.537	0.011
Trigram	916.987	9.841	0.003

Table 1: Cross Entropy and Perplexity for Bigram and Trigram Models

The results are somewhat counter intuitive. Typically, we would expect the Trigram Model to outperform the Bigram Model, which is generally the case. To make sense of these outcomes, we need to consider the methodology employed. The main reason the Bigram model surpassed the Trigram is due to the corpus size and diversity. The Trigram model could have potentially shown better results e.g. if the entire Wikipedia dataset had been used. A secondary effect of the relatively small corpus size is reflected in the Perplexity scores for both models, indicating that the bi- and tri-gram combinations were relatively unfamiliar to our model, resulting in perplexity scores of 371 and 917, respectively.

3 (iii) Language models testing

We are implementing autocomplete functionality for incomplete sentences using both bigram and trigram language models. Our code analyzes the provided prefix to predict the most probable next word based on the context provided by the preceding words. By leveraging probabilities associated with bigrams and trigrams, our functions generate contextually appropriate completions, ensuring smoother and more relevant sentence suggestions. Additionally, we've included safeguards such as maximum iteration counts to prevent potential infinite loops and handled end tokens to properly terminate generated sentences.

To accomplish this, we'll create two functions—one for a bigram-based model and one for a trigram-based model. These functions will finish sentences based on what comes before, using the trained language models to suggest what word might come next in an unfinished sentence. We'll compare the probabilities each model assigns to different words for a given incomplete sentence. This will help us understand how well each model predicts the next word. Then, we'll show the original sentence alongside the suggestions made by both the bigram and trigram models. This way, we can see which model gives more accurate suggestions and how they differ. After developing the functions for sentence autocompletion based on trigram and bigram models, we conducted testing using various examples to assess their performance. We found that the trigram model consistently generated more accurate sentences compared to the bigram model. Our observation throughout this process highlighted that the size and diversity of the training corpus significantly influence the quality of the generated completions. Larger corpus covering a wide range of topics tend to yield better results in sentence autocompletion.

After testing a few examples, the following were found as shown in Figure 1.

```
[ ]  1 prefix = "make sure"
     2 print('Bigram: ',bigram_autocomplete(prefix))
     3 print('Trigram: ',trigram_autocomplete(prefix))
```

```
Bigram:  make sure i
Trigram:  make sure that
```

```
[ ]  1 prefix = "believe it or"
     2 print('Bigram: ',bigram_autocomplete(prefix))
     3 print('Trigram: ',trigram_autocomplete(prefix))
```

```
Bigram:  believe it or the
Trigram:  believe it or not
```

```
[ ]  1 prefix = "i would like"
     2 print('Bigram: ',bigram_autocomplete(prefix))
     3 print('Trigram: ',trigram_autocomplete(prefix))
```

```
Bigram:  i would like a
Trigram:  i would like to
```

```
[ ]  1 prefix = "Try not to"
     2 print('Bigram: ',bigram_autocomplete(prefix))
     3 print('Trigram: ',trigram_autocomplete(prefix))
```

```
Bigram:  Try not to the
Trigram:  Try not to be
```

Figure 1: Results for autocorrection

4 (iv) Spelling corrector

Task Overview

Our objective entailed the development of an advanced spelling correction tool that considers contextual information to suggest accurate word corrections. This task required us to design a program capable of analyzing text, detecting potential spelling errors, proposing corrections, and ultimately selecting the most appropriate correction based on surrounding context.

Implementation Approach

Our solution involves the utilization of a beam search algorithm to efficiently explore various correction options. For each potentially misspelled word, the program generates a set of candidate corrections based on similarity metrics. Moreover, contextual information is incorporated by examining neighboring words to discern the most suitable correction.

Additionally, we integrated the Levenshtein distance metric to estimate the probability of a misspelling, thus enhancing the program's ability to discern likely corrections. These components are harmonized using a scoring mechanism, which weighs both contextual relevance and error likelihood to determine

the optimal correction.

Process

We utilize both bigram and trigram language models along with a beam search decoder to correct misspelled words in a sentence. The spelling correction probability $P(w_i \mid t_i)$ is modeled inversely by the Levenshtein distance between the misspelled word w_i and the candidate correction t_i.

Key Components

Levenshtein Distance Function

This function computes the Levenshtein distance between two words, which quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other. The probability of correction is inversely proportional to this distance.

Beam Search Correction Algorithm

Initialization Start with a beam containing the start token.

Expansion For each token in the input, generate candidate corrections from the vocabulary that have a non-trivial Levenshtein probability with the current token.

Scoring Calculate the combined score for each candidate using both the language model probability (from bigrams or trigrams) and the spelling correction probability.

Selection Retain the top scoring candidates as per the beam size constraint.

Parameter Tuning The importance of the language model score $\log P(t_1)$ versus the spelling correction score $\log P(w_1 \mid t_1)$ can be adjusted using the hyperparameters λ_1 and λ_2. We think that $\log P(w_1 \mid t_1)$ is more important because the algorithm must understand a sentence and not an individual word. As a result, we set $\lambda_2 = 0.8$ and $\lambda_1 = 0.2$.

Execution The provided Python function `beam_search_correction` accepts an input sentence, tokenizes it, and processes it through the beam search algorithm using either bigram or trigram models. The final output is a corrected version of the input sentence.

Results

Input Sentence: *have not i some riguv to iokplain*

Corrected Sentence with Bigrams: have not i come nigh to complain
Corrected Sentence with Trigrams: have not i am right is complain

Analysis

Corrected Sentence with Bigrams

The phrase "come nigh to" suggests that the bigram model struggled with contextual appropriateness. The model likely inferred these words based on local adjacency probabilities from its training on bigrams without a broader understanding of context, leading to semantically incorrect suggestions.

The model successfully corrected "iokplain" to "complain", demonstrating effective spelling correction for that word based on likely bigram sequences.

Corrected Sentence with Trigrams

The introduction of the words "am" and "is" indicates a limitation in the trigram model's ability to maintain grammatical structure in the presence of initial spelling errors. This suggests that the trigram context was not sufficient to override the erroneous outputs or that the error severely affected the trigram probabilities.

The word "right" as a correction for "riguv" could reflect a plausible phonetic or partial spelling match in the model, showing a successful correction attempt. However, it's contextually inappropriate in this sentence.

General Observations

- **Contextual Awareness:** Both models exhibit limitations in maintaining the syntactic and semantic structure of the sentence. This is a common challenge in spelling correction, especially when errors significantly deviate from correctly spelled words, impacting the predictive capabilities of n-gram models.
- **Error Types:** The models handle phonetically similar errors better (e.g., "iokplain" to "complain") compared to errors that introduce unusual word forms ("riguv"). This can be attributed to the reliance on statistical probabilities derived from n-gram frequencies, which are less robust to uncommon or novel errors.
- **Model Tuning:** The effectiveness of the models could potentially be enhanced by tuning the hyperparameters (λ_1 and λ_2), which balance the influence of the language model versus the spelling correction probability. Additionally, including a larger context (e.g., using higher-order n-grams or integrating syntactic parsers) might improve performance.

Conclusion

The provided results underscore the importance of using robust, context-aware models for spelling correction. They also highlight the need for careful parameter tuning and possibly integrating additional linguistic features to improve both the grammatical and contextual accuracy of the outputs.

5 (v) Artificial dataset testing

We've developed a function called *replace_letters* that takes a sentence as input and randomly replaces two letters in each word that has more than four characters with random uppercase or lowercase letters. The replacements are done in-place, and the modified sentence is returned. This function effectively introduces typographical errors into longer words of the input sentence. By providing this functionality, we can simulate errors in text data, which is useful for evaluating spelling correction algorithms.

6 (vi) WER-CER metrics

Initialization

- **Input:** `corrector_function` — a callable that takes a sentence as input and returns the corrected version.
- **Process:** During initialization, the class loads the WER and CER metrics using `load_metric('wer')` and `load_metric('cer')` from the Hugging Face `datasets` library.
- **Stored Metrics:** The loaded metrics are stored as instance variables, which will be utilized in the evaluation method.

Evaluation Method

- **Input:** `test_data` — a list of tuples, where each tuple contains an *original* list of tokens (correctly spelled sentence) and a *noised* list of tokens (sentence with spelling errors).
- **Process:**
  1. **Prediction and Reference Preparation:** Iterates over each tuple in `test_data`. For each tuple, it:
     - Joins the tokens of the noised sentence and passes this string to the `corrector_function` to obtain the corrected sentence.
     - Joins the tokens of the original sentence to use as the reference.
     - Appends the corrected and original sentences to `predictions` and `references` lists, respectively, if the original is not empty.
  2. **Metric Calculation:** Uses the stored metric functions to compute WER and CER, comparing each corrected sentence to its corresponding original sentence.
- **Output:** Returns the calculated WER and CER scores, representing the average error rates across all test sentences.

WER and CER scores

Average Word Error Rate (WER) with Bigrams: 24.55% Average Character Error Rate (CER) with Bigrams: 14.02%

Average Word Error Rate (WER) with Trigrams: 39.52% Average Character Error Rate (CER) with Trigrams: 20.91%