M.Sc. in Data Science

Text Analytics

Assignment 1 - Statistical language models

Grammatikopoulou Maria - f3352310
Phevos A. Margonis - f3352317
Moniaki Melina - f3352321

Instructor: Ion Androutsopoulos
Grader: Foivos Charalampakos

January 30, 2024

Abstract

The paper focuses on developing statistical language models for auto-completing sentences and spellchecking, using n-grams and Lidstone smoothing. It includes data preprocessing, vocabulary construction, and probability estimation. Key aspects are the use of bigram and trigram models, their evaluation via cross-entropy and perplexity, and their application in auto-completion and context-aware spelling correction. The results highlight the trade-offs between bigram and trigram models in terms of coherence, fluency, and error rates, with bigrams performing better in some aspects due to n-gram sparsity in trigrams.

Contents

1 Introduction

Statistical language models paved the way in the field of natural language processing (NLP). These models, essentially, learn to predict the probability of a sequence of words and are applied to a variety of fields such as machine translation, speech recognition, and text auto-completion. They operate by calculating the likelihood of word occurrence based on the frequencies of words and their combinations in a given corpus. This probabilistic approach allows the model to generate or complete sentences that are syntactically and semantically coherent.

The general aim of this assignment is to develop tools capable of auto-completing sentences and performing spellchecking on incorrect sentences. This involves constructing and employing statistical language models that can understand and predict textual patterns. Both tools are built as an experiment to benchmark the efficiency of simpler n-gram language models, at either generating or correcting sentences with cohesion in mind.

2 Implementation

The preparatory part of the code is designed to implement a set of language models using bigrams and trigrams, incorporating Lidstone (add-a) smoothing. The primary tasks include data preprocessing, vocabulary creation, and probability estimation for n-grams. The code is accompanied by essential libraries and modules for text processing and evaluation. The script begins by importing relevant libraries such as string, random, math, copy, nltk, Levenshtein, and a custom module 'evaluate' with the 'load' function. These libraries provide essential tools for text manipulation, language modeling, and model evaluation.

Dependencies The downloading of NLTK resources ('punkt' for tokenization and 'brown' for the Brown Corpus) is also necessary. We load all words from the Brown Corpus using `brown.words()`. The Brown Corpus contains a diverse set of texts from various genres, including fiction, news, academic papers, and essays. This will later prove useful in the training phase, since a broader range of topics will improve the model's generalizing ability. This step is crucial for obtaining a representative dataset for the tasks. A string text is created by concatenating all words from the Brown Corpus.

Tokenization The code proceeds by segmenting the concatenated text into individual sentences. Utilizing `nltk.sent_tokenize()`, the text string is divided into a list of sentences. Additionally, a refinement is made to remove trailing full stops from each sentence, ensuring consistent sentence representation. The script then tokenizes each sentence into words, creating a list of tokenized sentences. Employing `WhitespaceTokenizer()`, each sentence is split into words based on whitespace. Furthermore, a lowercase transformation is applied to standardize the case of all words, contributing to uniformity in the subsequent language model.

Train-Test-Split The dataset is partitioned into training, development, and test sets. A random seed is set for reproducibility using `random.seed(4444)`. The `random.shuffle()` function is applied to shuffle the tokenized sentences. This step is crucial since we want to evenly distribute the corpus topics contained in each subset. The lengths of the training, development, and test sets are 80%, 10%, 10% of the initial data set respectively.

Vocabulary The next step is vocabulary construction. A vocabulary list `(vocab_words)` is generated, containing words that appear at least 10 times in the training set. This filtering ensures that the vocabulary is comprised of relatively common words enhancing the generalizability of the language model. The size of the vocabulary `(vocab_size)` is determined based on the count of unique words in this filtered list, which in this case is over 7000 words. To handle out-of-vocabulary (OOV) words, each sentence in the training, development, and test sets is iterated. Words that are not present in the filtered vocabulary `(vocab_words)` are replaced with the special token 'UNK' (unknown), so that the model can handle words that were not present in the vocabulary during the training phase. An additional step is taken to include the 'UNK' token in the vocabulary `(vocab_words)`.

Language Models The code proceeds with the training phase of the models, by counting unigrams, bigrams, and trigrams frequency in the trainset. For each sentence in the training set, the

script utilizes the ngrams function from NLTK to generate padded with `start` and `end` pseudo-tokens (unigrams, bigrams, and trigrams). The Counter objects (`unigram_counter`, `bigram_counter`, and `trigram_counter`) are then updated to accumulate the counts of these n-grams.

A dictionary `(bigram_model)` is created to store the log probabilities for bigrams. A loop iterates through the bigram counts, and for each bigram, it calculates the Lidstone smoothed log probability using the formula for bigram probabilities with Lidstone smoothing. The log probability is then stored in the dictionary. This step is crucial for transitioning from raw counts to log probabilities, facilitating numerical stability and meaningful probability comparisons. Like the bigram model, a dictionary `(trigram_model)` is created to store the log probabilities for trigrams.

In a bigram language model, each word in a sequence is modeled based on the probability of occurring given only its preceding word. The model is trained on the training portion of the corpus, and the probability of each bigram is calculated by counting occurrences and applying Lidstone smoothing. The log probabilities are used to avoid numerical underflow and facilitate computations. Mathematically, for a bigram, the model estimates the conditional probability:

$$P(w_2|w_1) = \frac{C(w_1, w_2) + \alpha}{C(w_1) + \alpha \cdot |V|} \tag{1}$$

Where:

$$\begin{aligned} C(w_1, w_2) &: \text{bigram count} \\ C(w_1) &: \text{unigram count} \\ 0 \leq \alpha \leq 1 &: \text{smoothing hyper-parameter} \\ |V| &: \text{vocabulary size} \end{aligned}$$

Lidstone smoothing involves adding a small constant (alpha) to the numerator and adjusting the denominator to ensure that all n-grams, even those not observed in the training data, have non-zero probabilities. We have chosen the Lidstone hyper-parameter to be $a = 0.01$, which yields the best performance, as can be seen from Table 3.

Extending the concept of the bigram model, a trigram language model considers the conditional probability of a word given its two preceding words. Like the bigram model, trigram probabilities are computed through counting occurrences in the training corpus and applying Lidstone smoothing. the model estimates the conditional probability:

$$P(w_3|w_1, w_2) = \frac{C(w_1, w_2, w_3) + \alpha}{C(w_1, w_2) + \alpha \cdot |V|} \tag{2}$$

Where:

$$\begin{aligned} C(w_1, w_2, w_3) &: \text{trigram count} \\ C(w_1, w_2) &: \text{bigram count} \\ 0 \leq \alpha \leq 1 &: \text{smoothing hyper-parameter} \\ |V| &: \text{vocabulary size} \end{aligned}$$

Evaluation In the sequence, the two language models are being evaluated on the test set, and the evaluation metrics used are cross entropy and perplexity. Both cross entropy (3) and perplexity (4) provide insights into the performance of language models by quantifying how well their predictions align with the true distributions of words. Lower values for both metrics indicate better performance.

$$\text{CrossEntropy} = -\frac{1}{N} \sum_{\text{bigrams}} \log_2(P(w_2|w_1)) \tag{3}$$

$$\text{Perplexity} = 2^{H(p)} \tag{4}$$

Where:

$$\begin{aligned} N &: \text{Number of bigrams} \\ H(p) &: \text{Entropy of the probability distribution } p \end{aligned}$$

The code calculates the cross entropy and perplexity for a bigram language model. It initializes variables to store the sum of language probabilities `(sum_prob)` and the count of bigrams `(bigram_cnt)`. For each sentence in the test set, the sentence is prepended and appended with

pseudo-tokens (`<s>` and `<e>`). The loop then iterates over the bigrams in the sentence, calculating the probability of each bigram using the bigram model's probabilities obtained during training. The log probabilities are summed, and the count of bigrams is incremented. The cross entropy and perplexity are then computed and printed for the bigram model. In this script, loops over `pairwise(sent)` and `windowed(sent, n=3)` exclude probabilities with start pseudo-tokens (`start, start1, start2`), as they are not needed for the predictions. However, probabilities with the `end` token are included to predict sentence endings. The total length of the test corpus (N) counts `end` tokens, but not `start` tokens, aligning the model's focus on predicting actual sentence terminations. Similarly to the bigram model, we calculated the cross entropy and perplexity for the trigram language model. Consequently, the results are presented in Table 1.

	Bigram	Trigram
Cross-Entropy	7.32	9.25
Perplexity	160.06	609.51

Table 1: Cross-Entropy and Perplexity for both language models.

Theoretically, a trigram model is expected to be more accurate and better capture language dependencies than a bigram model. The reason for this improvement lies in the increased context that a trigram considers compared to a bigram. However, our bigram model seems to be more effective than the trigram since the perplexity has lower value. This happens because we have chosen Lidstone smoothing, which leads to poor estimates of language models.

2.1 Auto-complete module

After building the language models, we use them to auto-complete an incomplete sentence. Initially, part of a sentence is given to the `autocomplete bigram` function which then tokenizes and keeps the last word. This word is hereby referred as *seed* because it is used in the bigram model to predict the next word. The next word is simply selected as the one that is most probable to occur after the *seed* word.

The process of iteratively predicting the next word is handled by the `beam-search decoder` function. More specifically, a function `generate candidates`, lists all the possible words that can occur after the *seed word*, based on the n-gram LM, trained on that particular vocabulary. Consequently, each next word is used to create a new possible sentence. Each sentence or *state* is then forwarded in a `score` function, which utilizes the Markov assumption[1] of Equation 5 and the trained LM, for probability assignment. If the n-gram is not present in the training dataset, then $\log_2(1 \times 10^{-10})$ assigns a very small log probability to it.

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{5}$$

Apart from that, the beam-search algorithm simultaneously builds and evaluates k possible states, as specified in the `beam width`. The logic behind this approach is that the n-grams of the language models will punish non-frequent n-grams, and thus reduces the odds of building incoherent sentences. The process of auto-completion stops after `max depth` number of words have been predicted, or if there is no combination of n-gram in the trained language model. The main differences between auto-complete functions of different n are (a) the number of `seed` words used, which is one for the bigram and two for the trigram, and (b) the LM used to score the sentences.

A set of examples demonstrating how our sentence completion works, including interesting cases, bad and good completions is presented in Figure 1, and the results can be summarized as follows:

- The bigram model can autocomplete almost any sentence. This happens because there are more combinations of bigrams generated from the test data set and the beam-search decoder takes into account only the last unigram. The downside is that the models tend to hallucinate by cycling through the most common bigrams.

- The trigram model is more semantically and syntactically coherent. On the other hand, if a bigram used to predict the next word is not present in the language model, the autocompletion breaks, and this behavior is more easily noticed in the trigram rather than the bigram model. To alleviate this problem, the unknown n-gram could have been replaced with `UNK`, but due to its disproportionate occurrence in the training set, this will inevitably lead to hallucinations.

[1] Speech and Language Processing (3rd ed. draft) section 3.1

2.2 Context-aware spelling corrector module

The challenge of correcting a given sentence which contains spelling errors is twofold. Firstly, each misspelled word must be replaced with an orthographically correct one. Secondly, the resulting sentence should be syntactically and semantically coherent. The latter problem has been dealt with using the auto-completion module. The former, requires the utilization of *Levenshtein distances*, between the misspelled word and each word contained in the vocabulary, calculated and stored in the dictionary `distances`. Consequently, when a new state is passed in the `score` function along with the misspelled sentence `word_list`, the additional score of $\frac{1}{edit\_distance+1}$ is calculated. If the `distances` dictionary does not contain the combination of the misspelled word and the `new_candidate` word, it returns a large *edit distance* value, indicating that this combination is implausible. Then, the two scores are combined using the interpolation of Equation 6.

$$\hat{t} = \arg\max_{t_{1:k}} \left(\lambda_1 \log P(t_{1:k}) + \lambda_2 \log P(w_{1:k} \mid t_{1:k})\right) \tag{6}$$

Due to sub-optimal smoothing, the weight that regulates the effect of the LM had to be set to a low value of 20% and thus the Levenstein score weight to 80%. The decoder iterates in the same manner as the auto-complete module and it terminates when every word of the `word_list` has been examined.

2.3 Artificial test-set and Benchmarks

The code is designed to generate an artificial test dataset for evaluating a context-aware spelling corrector. It operates on a copy of the original test set (testSet). It iterates through each sentence and word in the dataset. For non-unknown words (those not labeled as 'UNK'), it introduces artificial spelling errors by randomly replacing each character with a small probability. This is achieved by creating a new word where each character has a 10% chance of being replaced by a random ASCII letter. The resulting dataset, referred to as `artTestSet`, serves as a modified version of the original test data, simulating spelling errors. Subsequently, for reasons of time efficiency, a portion of 100 texts from `artTestSet` are being corrected using both spelling correctors. Following this, a set of outputs is presented in Figure 2, and the results can be summarized as follows:

- The bigram model is better at correcting misspelled words and creating coherent sentences. This problem stems from the way the new candidate sentences are generated and evaluated. For example, in the last output, the bigram model found the pair (walked, toward) during its training phase and thus was able to provide a likelihood of that occurrence. In contrast, the trigram model did not find the combination (they, walked, toward) in its dictionary and could not consider it as a viable alternative.

- Given the above explanation, it is surmised that the interpolation presented in Equation 6 can effectively punish the LM's decision in the case of the bigram model. Conversely, in many cases of the trigram model, this dynamic is not present and the Levenshtein distance is punished instead.

2.4 Word Error Rate and Character Error Rate metrics

We made an evaluation of the performance of the context-aware spelling corrector, both in the bigram and trigram language model. The code computes both *Word Error Rate* (WER) and *Character Error Rate* (CER) metrics by comparing the corrected predictions with the original reference sentences. These metrics offer valuable insights into the effectiveness of the bigram and trigram-based spelling correction model in both word and character-level accuracy. WER and CER are calculated using an external library called `evaluate`, and the results are presented in Table 2.

	Bigram	Trigram
Word Error Rate	0.3185	0.4640
Character Error Rate	0.3035	0.4178

Table 2: Word Error Rate and Character Error Rate metrics for both language models.

We can observe that the bigram model has both lower WER (0.3185) compared to the trigram model (0.4640) and lower CER (0.3035) compared to the trigram model (0.4178), indicating that the bigram model performs better in terms of word-level and character level accuracy. The fact

that we observe a lower CER in the bigram model compared to the trigram model was also noticed in the examples we created earlier using the spellcheck correctors.

3 Conclusions

The pivotal findings of this experiment were:

1. There is a negative correlation between the size of the vocabulary and the Language Cross-Entropy. This implies that as the vocabulary size increases, the model's ability to predict the next word in a sequence becomes less precise, increasing cross-entropy.
2. The replacement of low-frequency words, or words that do not appear in the vocabulary, with `UNK` tokens negatively impact the LM's generalizing ability. For example, if the word 'few' in a part of our vocabulary but during testing the OOV word 'fewer' was presented to the LM, the algorithm would replace it with UNK. Instead, a better approach would be to use stemming, which would reduce the word 'fewer' to simply 'few', but this is outside the scope of this experiment.
3. The trigram model can generate more fluent sentences, compared to bigram.
4. The trigram model is prone to brakes, due to n-gram sparsity, compared to bigram.
5. The bigram model can generate longer sentences without breaking, but it converges to a local maximum neighborhood and thus tends to be repetitive.
6. The bigram model was better at correcting sentences with spelling errors, compared to the trigram which was hampered by its n-gram sparsity.
7. The above results are also reflected by the WER and CER scores, which are better for the bigram model.

Contributions

In this assignment, the work was divided between the members of our team. The modules (i) language models and (ii) evaluation were implemented by *Maria Grammatikopoulou*. Building upon those, the (iii) auto-completion and (iv) context-aware spelling-corrector were constructed by *Phevos A. Margonis*. Ultimately, the (v) creation of artificial-test-set and (vi) WER – CER metrics for evaluation were formulated by *Melina Moniaki*.

A Appendix

```
# Example usage:
input_text = "I would like"
print(f"Autocompleted Bigram:  '{autocomplete_bigram(input_text, max_depth=5, beam_width=2)}'")
print(f"Autocompleted Trigram: '{autocomplete_trigram(input_text, max_depth=5, beam_width=2)}'")
```

```
Autocompleted Bigram:  'I would like a few years ago ,'
Autocompleted Trigram: 'I would like to see the car <e>'
```

```
input_text = "The most popular"
print(f"Autocompleted Bigram:  '{autocomplete_bigram(input_text, max_depth=15, beam_width=2)}'")
print(f"Autocompleted Trigram: '{autocomplete_trigram(input_text, max_depth=15, beam_width=2)}'")
```

```
Autocompleted Bigram:  'The most popular as a few years ago , and the same time , and the same time'
Autocompleted Trigram: 'The most popular man on the other hand , the first time in the world , and the'
```

```
input_text = "In conclusion"
print(f"Autocompleted Bigram:  '{autocomplete_bigram(input_text, max_depth=15, beam_width=2)}'")
print(f"Autocompleted Trigram: '{autocomplete_trigram(input_text, max_depth=15, beam_width=2)}'")
```

```
Autocompleted Bigram:  'In conclusion , and the same time , and the same time , and the same time'
Autocompleted Trigram: 'In conclusion <e>'
```

```
input_text = "federal policies"
print(f"Autocompleted Bigram:  '{autocomplete_bigram(input_text, max_depth=15, beam_width=2)}'")
print(f"Autocompleted Trigram: '{autocomplete_trigram(input_text, max_depth=15, beam_width=2)}'")
```

```
Autocompleted Bigram:  'federal policies and the same time , and the same time , and the same time ,'
Autocompleted Trigram: 'federal policies will produce a better understanding of the united states , and the other hand ,'
```

Figure 1: Input/output examples demonstrating the efficiency of the auto-complete module.

	$a = 1$	$a = 0.1$	$a = 0.01$	$a = 0.001$
Bigram Perplexity	481	229	160	163
Trigram Perplexity	2.039	988	610	576

Table 3: Hyper-parameter tuning for the add-a smoothing using Perplexity as evaluation score.

```
# %% (vi) example use
inputText = artTestSet[91]
print(f"Inpput text:        {inputText}")
print(f"Bigram spellcheck:  {spellcheck_bigram(inputText)}")
print(f"Trigram spellcheck: {spellcheck_trigram(inputText)}")
```

```
Inpput text:        ['i', 'AaJted', 'him', 'p', 'with', 'a', 'UNK', 'UNK']
Bigram spellcheck:  ['i', 'have', 'him', ',', 'with', 'a', 'UNK', 'UNK']
Trigram spellcheck: ['i', 'asked', 'him', ',', 'with', 'a', 'UNK', 'UNK']
```

```
# %% (vi) example use
inputText = artTestSet[143]
print(f"Inpput text:        {inputText}")
print(f"Bigram spellcheck:  {spellcheck_bigram(inputText)}")
print(f"Trigram spellcheck: {spellcheck_trigram(inputText)}")
```

```
Inpput text:        ['and', 'onc', 'had', 'been', 'too', 'mPny']
Bigram spellcheck:  ['and', 'UNK', 'had', 'been', 'too', 'many']
Trigram spellcheck: ['and', 'one', 'had', 'been', 'too', 'many']
```

```
# %% (vi) example use
inputText = artTestSet[191]
print(f"Inpput text:        {inputText}")
print(f"Bigram spellcheck:  {spellcheck_bigram(inputText)}")
print(f"Trigram spellcheck: {spellcheck_trigram(inputText)}")
```

```
Inpput text:        ['yoc', 'cyuld', 'wish', 'that']
Bigram spellcheck:  ['you', 'could', 'wish', 'that']
Trigram spellcheck: ['you', 'could', 'wish', 'for']
```

```
# %% (vi) example use
inputText = artTestSet[348]
print(f"Inpput text:        {inputText}")
print(f"Bigram spellcheck:  {spellcheck_bigram(inputText)}")
print(f"Trigram spellcheck: {spellcheck_trigram(inputText)}")
```

```
Inpput text:        ['they', 'walked', 'tAwPrd', 'each', 'otheU']
Bigram spellcheck:  ['they', 'walked', 'toward', 'each', 'other']
Trigram spellcheck: ['they', 'walked', 'the', 'deck', ',']
```

Figure 2: Input/output examples demonstrating the efficiency of the context-aware spelling corrector.