Exercises on n-gram language models № 1

Kagioglou Maria, Panourgia Evangelia 28/01/2025

Google Colab Notebook

Task 3i-(Preprocess & Tuning)

We use the `Brown Corpus` (Brown Corpus), the first million-word electronic corpus of English. It includes text from 500 sources categorized by genre (e.g., news, editorial), containing 57,340 sentences. The dataset is partitioned into three subsets: **training** (80%), **validation (development)** (10%), and **test** (10%), using the `train_test_split` function. The test set is initially separated, followed by partitioning the remaining data into training and validation sets. A fixed random seed ensures reproducibility of the splits, as shown in the box below.

Dataset Split Sizes

Train set size: 45,872
Dev set size: 5,734
Test set size: 5,734

Based on the training set, we created the vocabulary (function: create_vocab) by removing punctuation, converting text to lowercase, and excluding words with a frequency of less than 10, resulting in a vocabulary size of 7631. Sentences in the training, validation, and test sets were preprocessed (function: preprocess) by tokenizing (using `brown.sents()`), lowercasing, and replacing out-of-vocabulary (OOV) tokens with <UNK>. The most frequent words in the train set were <UNK> (238215 occurrences), the (56144), and of (29370).

After preprocessing, we calculated unigram, bigram, and trigram counts for the training, test, and validation sets using the `ngrams` function. Note that counters were computed for both the test and validation sets, as our evaluation functions (evaluate_model and laplace_smoothing) rely on ngram iteration, not individual sentences. To determine the optimal Laplace smoothing parameter a, we tuned it for bigrams and trigrams by evaluating perplexity for values $a = [0.001, 0.01, 0.1, 1.0, 2.0, 10.0]$ on the development set, using training set counts. The best perplexity (indicating a better model) was achieved with $a = 0.01$. Due to limited time, we did not perform cross-validation; instead, we set k-fold to 1 during tuning.

Perplexity Results for Trigrams and Bigrams

Trigrams:	**Bigrams:**
Alpha: 0.001, Perplexity: 454.37	Alpha: 0.001, Perplexity: 231.46
Alpha: 0.01, Perplexity: 452.30	Alpha: 0.01, Perplexity: 198.13
Alpha: 0.1, Perplexity: 666.39	Alpha: 0.1, Perplexity: 253.99
Alpha: 1.0, Perplexity: 1183.19	Alpha: 1.0, Perplexity: 482.71
Alpha: 2.0, Perplexity: 1397.71	Alpha: 2.0, Perplexity: 614.82
Alpha: 10.0, Perplexity: 1917.24	Alpha: 10.0, Perplexity: 1099.23
Best Alpha: 0.01, Best Perplexity: 452.30	Best Alpha: 0.01, Best Perplexity: 198.13

The lower perplexity for `bigrams` vs. trigrams is due to trigrams requiring more specific context, leading to more unseen combinations. Add-a smoothing increases the probability of unseen trigrams but penalizes observed ones, resulting in higher perplexity. With rare words replaced by <UNK>, $\alpha = 0.01$ is suitable for both bigram and trigram models.

Task 3ii-(Evaluation)

The objective at this stage is to evaluate the performance of the `bigram` and `trigram` models on the **test set** using Add-α (Laplace) smoothing. This evaluation is conducted by analyzing metrics such as perplexity and cross-entropy, which measure the model's predictive accuracy. Lower values for these metrics indicate better performance and a stronger alignment between the model's predictions and the observed data.

Trigram and Bigram Evaluation Results

Evaluation trigram add-a:	**Evaluation bigram add-a:**
Cross entropy on the test set: 8.84	Cross entropy on the test set: 7.65
Perplexity on the test set: 459.28	Perplexity on the test set: 201.09

Task 3iii-(Beam Search-Autocomplete)

For the auto-complete task, we use the `beam_search` algorithm with bigram or trigram models to predict the next words. The process works as follows:

1. **Initialization:** Start with the input sentence and an initial probability of 1.0.
2. **Prediction Loop:** Predict a predefined number of next words (`num_next_words`).
3. **Prefix Determination:** Use the last word (bigram) or last two words (trigram) as the prefix.
4. **Probability Calculation:** Calculate word probabilities with Laplace smoothing.

5. **Beam Search Pruning:** Keep the top-k most probable candidates to reduce complexity.
6. **Penalizing** <UNK>**:** Scale down <UNK> predictions to avoid unknown token dominance.
7. **Updating Candidates:** Generate new sentences by appending predicted words.
8. **Output Selection:** Return the sentence with the highest cumulative probability.

This approach balances efficient prediction with exploration of multiple completions. Beam search controls prediction space, while penalizing <UNK> reduces its impact on n-grams with unknown tokens. The user specifies the number of words to predict, set to 15 for demonstration. For example, given "and you", the trigram model predicts 3 potential next words with respective probabilities. The final sentence is: "and you have to be the only one of the United States of America in congress <e>".

Trigram Model Autocompletion

Initial sentence:
and you think you have language problems <UNK>

Top 3 candidates at this step:
Candidate 1: `and you have` (Probability: 0.04734893248247065)
Candidate 2: `and you will` (Probability: 0.03947057433230914)
Candidate 3: `and you were` (Probability: 0.03947057433230914)

... .
....
... .
Top 3 candidates at this step:
Top 3 candidates at this step:
Candidate 1: `and you have to be the only one of the united states of america in congress <e>` (Probability: 1.0119005050595385e-17)
Candidate 2: `and you have to be the only one of the united states of america in congress assembled` (Probability: 5.0721197385777364e-18)
Candidate 3: `and you have to be the only one of the united states of america and the other` (Probability: 3.756544348708043e-18)

and you have to be the only one of the united states of america in congress <e>

Bigram Model Autocompletion

Initial sentence:
and you think you have language problems <UNK>

Top 3 candidates at this step:
Candidate 1: `and you <e>` (Probability: 0.06055506643932049)
Candidate 2: `and you can` (Probability: 0.04681028109943424)
Candidate 3: `and you are` (Probability: 0.04272399356595454)
... .
....
... .
Top 3 candidates at this step:
Candidate 1: `and you can be a few years ago the same time to the first time to the` (Probability: 1.80878893275087e-19)
Candidate 2: `and you can be a few years ago the same time to be a few years ago` (Probability: 1.5606548487218664e-19)
Candidate 3: `and you can be a few years ago the same time to be a few years <e>` (Probability: 1.5058998325400626e-19)

and you can be a few years ago the same time to the first time to the

Task 3iv-(Beam Search Decoder)

Total Score

$$\text{Total score} = \text{LM score} + \text{EM score}$$

Where:

- **LM score**: Based on the language model.
- **EM score**: Based on Levenshtein distance.

1. Language Model Score

The **LM score** calculates the probability of a word sequence using the n-gram model (e.g., bigram, trigram). Laplace smoothing is applied, and the logarithm is used to avoid very small values.

LM Score

$$\text{LM score} = \sum_{i=1}^{|\text{new\_seq}|-n+1} \log_2 \left(\frac{\text{count}(ngram_i) + \alpha}{\text{count}(n-1\_gram_i) + \alpha \cdot \text{vocab\_size}} \right)$$

Where:

- count($ngram_i$) is the count of the n-gram.
- count($n-1\_gram_i$) is the count of the (n-1)-gram prefix.
- α is the Laplace smoothing parameter.
- vocab_size is the vocabulary size.

2. Error Model Score

The **EM score** calculates the similarity between the observed and candidate words using **Levenshtein distance**, which counts the number of edits (insertions, deletions, substitutions).

EM Score

$$\text{EM score} = -\text{Levenshtein distance}(\text{observed}, \text{candidate})$$

Where:

- Levenshtein distance is the number of edits needed.

3. Total Score

The **Total score** is the sum of the **LM score** and the **EM score**, used to rank candidate word sequences, with higher scores indicating better candidates.

4. Beam Search Update

Beam search tracks the top k sequences, updating the beam by keeping only the best candidates.

Beam Search Update

$$\text{new\_beam} = \text{Top-k}(\text{candidates}, k = \text{beam\_width})$$

Where:

- candidates are the possible continuations of the sequence.
- Top-k returns the top k sequences.

5. Final Output

The best sequence is selected based on the highest score after processing the entire sequence. If no valid candidates are found, the algorithm returns the best previous sequence.

Final Output

$$\text{Best sequence} = \max(\text{beam}, \text{key} = \lambda x : x[0])[1]$$

Where:

- **beam** is the list of candidate sequences with their scores.
- The sequence with the highest score is returned.

Handling '<UNK>':

The <UNK> token indicates an out-of-vocabulary word. To handle it, Laplace smoothing can be applied to select the most probable word from the vocabulary. Contextual information from surrounding words can further improve predictions by choosing a word that fits the sentence's meaning, especially in cases of heavy misspellings or rare words. However, predicting the correct word for an <UNK> token remains challenging when the misspelled word is significantly distorted.

Task 3v - (Artificial Dataset)

To create an artificial dataset from the test set, we introduced **typos** by randomly replacing characters with visually, acoustically, or keyboard-proximity similar ones, based on a predefined set of mappings. This simulates real-world noisy inputs or transcription errors and allows for evaluating the **beam search decoder** performance.

The objective was to assess the decoder's ability to reconstruct original sentences with misspellings. The replacement probability was set to 0.05, balancing noise introduction with readability. This choice prevents excessive deviation, which would increase the Levenshtein distance threshold, leading to higher computational costs for generating more candidate words.

For both the test set and the typos dataset (v), we filtered the sentences to ensure they differed by at least one character. Subsequently, for the filtered typos dataset, we utilized the `bigram` and `trigram` models to correct spelling errors by invoking the `beam_search_decoder` function.

As an example, introducing typos into the test set transformed the sentence:

`"and you think you have language problems"` $\rightarrow$ `"and you thimk yov have language problems"`

Spelling Correction per Sentence

At this step, we manually select a sentence from the test set and introduce two types of errors: (1) misspellings, such as changing "language" to "langage," and (2) incorrect word substitutions, such as replacing "to" with "two." This allows us to evaluate whether the beam search decoder can effectively correct both types of errors and recover the original sentence.

Type Error 1: Wrong Spelling

Initial Sentence:

['and', 'you', 'think', 'you', 'have', 'language', 'problems']

Misspelled Sentence:

['and', 'you', 'thnk', 'yo', 'hve', 'langage', 'problems']

Bigram Model:

['end', 'of', 'the', 'us', 'the', 'language', 'problems']

Trigram Model:

['and', 'you', 'think', 'of', 'the', 'language', 'problems']

Type Error 2: Wrong Word Substitution

Initial Sentence:

['this', 'will', 'permit', 'you', 'to', 'get', 'a', 'rough', 'estimate', 'of', 'how', 'much', 'the', 'materials', 'for', 'the', 'shell', 'will', 'cost']

Misspelled Sentence:

['this', 'will', 'permit', 'you', 'two', 'get', 'a', 'rough', 'estimate', 'off', 'how', 'much', 'the', 'materials', 'four', 'the', 'shell', 'will', 'cost', '<UNK>']

Bigram Model:

['this', 'will', 'permit', 'to', 'do', 'not', 'a', 'rough', 'estimate', 'of', 'how', 'much', 'the', 'material', 'for', 'the', 'hell', 'will', 'not', 'that']

Trigram Model:

['this', 'kill', 'permit', 'you', 'two', 'get', '', 'rough', 'estimate', 'of', 'how', 'much', 'the', 'material', 'for', 'the', 'small', 'hall', 'cost', 'it']

Notes

1. **Low Levenshtein Distance ((≤ 1)):**
   A low distance ((≤ 1)) limits the candidate pool to words close to the misspelled one,

increasing the likelihood of selecting the correct word.

2. **Higher Levenshtein Distance (≤ 2):**
   A distance of ≤ 2 increases the candidate pool, but the models are more likely to confuse the correct word, reducing precision.

3. **Handling the** <UNK> **Token:**
   The <UNK> token is not handled well because the words after it are not in the training vocabulary, leading to higher WER and CER.

4. **Strict Threshold (Levenshtein Distance ≤ 1):**
   A strict threshold (≤ 1) limits candidates to minor changes. Heavily misspelled words may have no valid candidates.

5. **Increasing Leniency (Higher Levenshtein Distance):**
   A higher distance allows for more lenient candidate selection, ensuring even heavily misspelled words can be matched.

6. **Trade-offs with Larger Candidate Pools:**
   A larger candidate pool introduces irrelevant options, increasing the risk of selecting the wrong word. A balance is struck with a distance of ≤ 2 to minimize errors.

Task 3vi - (WER-CER)

The outputs generated by the aforementioned process in (v) section, along with the corresponding filtered test set, were then used to compute the evaluation metrics `WER` (Word Error Rate) and `CER` (Character Error Rate), leveraging the `jiwer` library.

The resulting evaluation metrics are presented in the box below.

Evaluation Results	
Bigram Evaluation Results:	**Trigram Evaluation Results:**
• **Average WER**: 0.60	• **Average WER**: 0.19
• **Average CER**: 0.30	• **Average CER**: 0.15

The `trigram` model outperformed the `bigram` model, achieving better scores in both Word Error Rate (WER) and Character Error Rate (CER).

What This Means:

- **WER Improvement:** A WER of 41% is a significant improvement, approaching a more acceptable range for practical use. However, for high-quality systems, the ideal WER target is below 10%.
- **CER Improvement:** A CER of 15% demonstrates a reduction in character-level errors, indicating the system is capturing text and structure more accurately.

Overall Implication: The transcription quality of the system has significantly improved. It now produces fewer errors at both the word and character levels, showcasing enhanced capability in understanding and converting speech to text. While further refinement is necessary, these improvements suggest that the system is becoming increasingly reliable for practical applications.

General Notes

The analysis below examines data sparsity in the Brown corpus and its impact on bigram and trigram perplexities. Data sparsity arises when many n-grams occur only once, hindering the model's ability to generalize. This leads to higher perplexities as the model struggles with unseen or infrequent sequences.

N-gram Analysis Summary	
Bigram Analysis:	**Trigram Analysis:**
• **Total Count:** 1,161,191	• **Total Count:** 1,161,190
• **Unique Count:** 436,003 (37.5%)	• **Unique Count:** 895,057 (77.1%)
• **Single-occurrence Count:** 323,168 (74.1% of unique bigrams)	• **Single-occurrence Count:** 801,997 (89.6% of unique trigrams)

The table summarizes total, unique, and single-occurrence n-gram counts, shedding light on the Brown corpus's adequacy for training effective language models.

From the results, we can infer the following:

1. **For Bigrams:**

    - This suggests that a significant portion of bigrams are unique, with most occurring only once in the entire corpus.

2. **For Trigrams:**

    - The situation is even more severe for trigrams, with a very high proportion of unique trigrams and the majority of them appearing only once.

The above results are verified by the following graph, where we can observe that the first bar in both histograms (near 0 frequency) is exceptionally tall, indicating high sparsity in the corpus—most bigrams and trigrams occur very rarely:

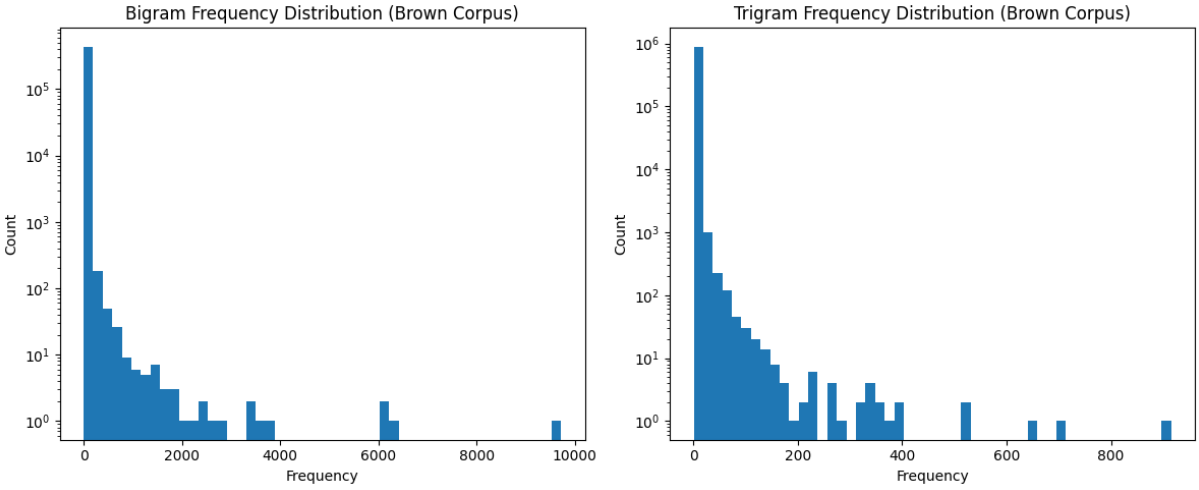

Figure 1: Frequency Distribution.

While analyzing the common n-grams in the train, test, and validation sets, we observe the following:

1. Unseen bigrams are high, indicating many word combinations in validation and test sets are missing from the training set.
2. Unseen trigrams are much higher than bigrams, reflecting increased sparsity due to the rapid growth of trigrams.
3. The similarity in unseen bigrams (15,724) and trigrams (53,339) across validation and test sets suggests both sets share similar complexity with the training data.

Unseen N-grams Analysis

Unseen N-grams in the Validation Set:

- Unseen unigrams: 0
- Unseen bigrams: 15,724
- Unseen trigrams: 53,339

Unseen N-grams in the Test Set:

- Unseen unigrams: 0
- Unseen bigrams: 15,919
- Unseen trigrams: 53,599

Acknowledgment

This project was a collaborative effort, where both team members worked together to achieve a deeper understanding of the theory behind n-grams. Both members contributed equally in implementing the exercise.