Assignment 1 - n-gram Language Models

Contributors: Dimitris Vougioukos - f3352411, Ioannis Papadopoulos - f3352409

Contribution: Dimitris Vougioukos: 1,2,3,4,5,6 - Ioannis Papadopoulos: 1,2,3

The assignment aims to develop bigram and trigram language models, evaluate their cross-entropy and perplexity, and apply them for sentence auto-completion and auto-correction.

The assignment comprises six tasks: (1) building bigram and trigram language models with Laplace smoothing, (2) estimating cross-entropy and perplexity for both models, (3) implementing a sentence auto-completion mechanism, (4) developing a context-aware spelling corrector, (5) creating an artificial dataset for evaluating the spelling corrector, and (6) assessing the spelling corrector using the artificial dataset based on Word Error Rate (WER) and Character Error Rate (CER).

In the next chapters we will look at the steps and design decisions made in all tasks to achieve the desired results.

Task 1

At the start of the task, we examined the Reuters corpus to understand its structure. We identified two file types: `training` and `test`. Using these, we split the raw text into training and testing subsets. Initially, we applied the `sent_tokenize` function from the `nltk` library to tokenize sentences, yielding 37,700 training sentences and 13,281 test sentences. We then used `word_tokenize` to extract tokens from each sentence in both subsets. Below, we present the token counts for each subset and the most common tokens.

Subset	Total Tokens	Unique Tokens	10 Most Frequent
Training	1,136,318	41,908	('the', 51,383), (',', 39,586), ('.', 37,651), ('of', 27,306), ('to', 27,306)
Test	412,952	25,399	('the', 17,862), (',', 14,079), ('.', 13,255), ('of', 9,443), ('to', 8,969)

Next, we filtered the training subset to include only tokens appearing more than a specified number of times. We retained tokens with a frequency greater than five, replacing all others with the special token 'UNK.' This resulted in 31,407 token replacements.

To build the bigram and trigram language models, we implemented functions for each training and testing step, as described below.

Function 1 - train_lms

This function trains bigram and trigram language models by computing the frequencies of all unigrams, bigrams, and trigrams in a given training subset.

Function 2 - calculate_bigram_prob

This function calculates the probability of a bigram model for two given words/tokens using Laplace smoothing, based on the following formula.

$$P_{Laplace}(W = w_k|w_{k-1}) = \frac{c(w_{k-1}, w_k) + a}{c(w_{k-1}) + a|V|}$$

Function 3 - calculate_trigram_prob

This function calculates the probability of a trigram model for three given words/tokens using Laplace smoothing, based on the following formula.

$$P_{Laplace}(W = w_k|w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + a}{c(w_{k-2}, w_{k-1}) + a|V|}$$

Function 4 - bigram_lm

This function illustrates the bigram model for a given sentence. It prepends the `<s>` token at the beginning and appends the `<e>` token at the end. Then, it iterates over the sentence tokens starting from the second one, calculating bigram probabilities for all consecutive token pairs. Finally, it returns the sum of the corresponding logarithmic probabilities and the sentence length, excluding the starting token.

Function 5 - trigram_lm

This function illustrates the bigram model for a given sentence. It prepends the `<s>` token twice at the beginning and appends the `<e>` token at the end. Then, it iterates over the sentence tokens starting from the third one, calculating trigram probabilities for all three consecutive tokens. Finally, it returns the sum of the corresponding logarithmic probabilities and the sentence length, excluding the starting tokens.

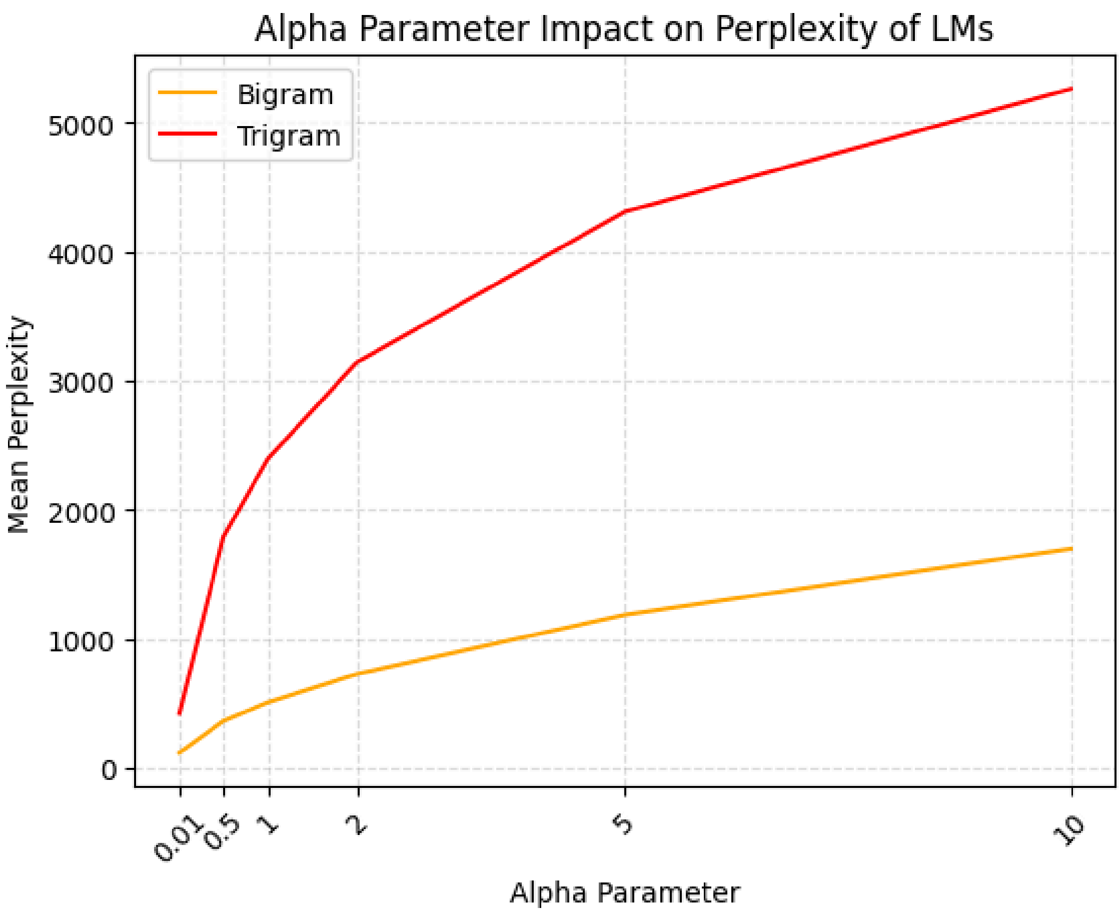
Function 6 - calculate_cross_entropy_perplexity

This function calculates the cross-entropy and perplexity of a bigram or trigram language model. It processes a list of sentences, computing the total number of tokens and the sum of the logarithmic probabilities of tokens of each one sentence, based on the chosen model. Finally, it aggregates these values and computes the cross-entropy and perplexity using the following formulas.

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

$$Perplexity = 2^{H_{CrossEntropy}}$$

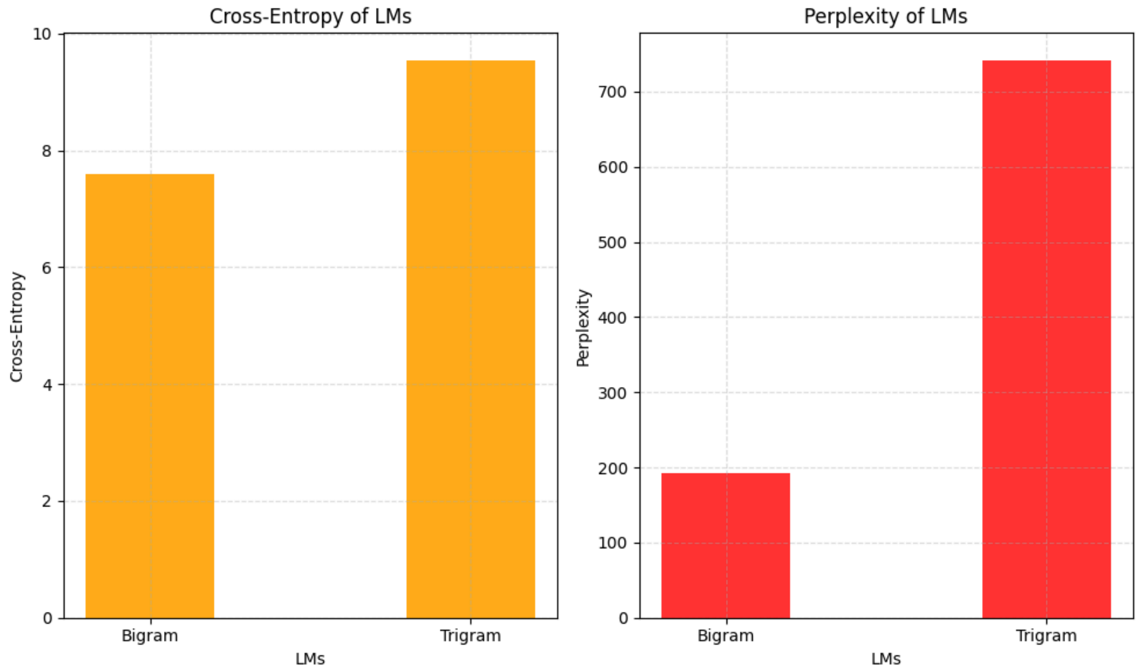
To build the models, we fine-tuned the alpha parameter for Laplace smoothing in both models. We conducted a 5-fold cross-validation, testing various alpha values and selecting the optimal one based on the average perplexity as the evaluation metric. Below is a visualization of how different values of the Laplace alpha parameter impact the average perplexity of each model.



As shown in the line chart, the optimal alpha value for both models is 0.01, yielding an average perplexity of 120.1200 for the bigram model and 424.7571 for the trigram model.

Task 2

In this task, we utilized the predefined functions from Task 1 to train the models on the full training subset. We then computed the cross-entropy and perplexity for both models. The results are presented below.



Bigram LM -> Cross-Entropy: 7.5934 & Perplexity: 193.1207

Trigram LM -> Cross-Entropy: 9.5335 & Perplexity: 741.1069

The Bigram LM seems to be the best model since it gives the lowest cross-entropy and perplexity.

Task 3

In this task, we implemented two functions to build a sentence completion algorithm — one using the bigram model and the other using the trigram model. The functions and their descriptions are presented below.

Function 1 - complete_sentence_bigram

This function implements a bigram-based sentence completion algorithm using the beam search method. It takes an incomplete sentence as input and attempts to generate a completed sentence based on bigram probabilities while maintaining the most probable sentence candidates within a given beam width.

The algorithm follows these steps:

- Initializes a beam search queue, starting with the incomplete sentence.
- Iteratively expands the beam by predicting the most probable next words based on bigram probabilities.
- Retains only the top `beam width` candidates at each step using as comparison metric the sum of the logarithmic probabilities of each sentence candidate.

- Stops when a maximum sentence length (30) is reached or all candidate sentences end with the `<e>` token.
- Returns the most probable completed sentence (excluding start and end tokens) and the final log probability score of the sentence.

Function 2 - complete_sentence_trigram

This function implements a trigram-based sentence completion algorithm using the beam search method. It takes an incomplete sentence as input and attempts to generate a completed sentence based on trigram probabilities while maintaining the most probable sentence candidates within a given beam width. It extends the bigram approach by considering sequences of three consecutive words to predict the next word in a sentence.

The algorithm follows these steps:

- Initializes a beam search queue, starting with the incomplete sentence.
- Iteratively expands the beam by predicting the most probable next words based on trigram probabilities.
- Retains only the top `beam width` candidates at each step using as comparison metric the sum of the logarithmic probabilities of each sentence candidate.
- Stops when a maximum sentence length (30) is reached or all candidate sentences end with the `<e>` token.
- Returns the most probable completed sentence (excluding start and end tokens) and the final log probability score of the sentence.

Unlike the bigram-based sentence completion algorithm, here instead of considering only the last word, it uses the last two words to determine the next word and also provides more accurate contextual predictions than the bigram model due to the increased context window.

We created some examples of incomplete sentences to see the related generated texts and to show how the two sentence completion algorithms work. The results are presented in the table below.

Incomplete Sentences	Complete Sentences - Bigram LM	Score - Bigram LM	Complete Sentences - Trigram LM	Score - Trigram LM
you are	you are expected to the company said .	-17.346	you are not expected to be identified .	-28.755
he decided to play	he decided to play a share .	-10.624	he decided to play a role in the second quarter .	-28.86
the important thing	the important thing they said .	-9.899	the important thing is the first quarter .	-24.008

Incomplete Sentences	Complete Sentences - Bigram LM	Score - Bigram LM	Complete Sentences - Trigram LM	Score - Trigram LM
she wants to comment	she wants to comment .	-2.979	she wants to comment on the sale of the company said .	-21.938
you need to find	you need to find it said .	-9.225	you need to find a way to cut its trade surplus .	-32.35
the german economy	the german economy .	-2.786	the german economy , '' he said .	-12.553
yesterday	yesterday .	-2.939	yesterday , dealers said .	-9.845
i would like to eat		0		0
i got an message	i got an message ''	-8.815		0
`<s>`	he said .	0.705	he said .	-7.131

As observed, in the first few incomplete sentences, the trigram language model generates more fluent and meaningful completions compared to the bigram model. However, in the last five cases, both models struggle, producing incoherent text. In some instances, the models generate empty sentences, indicating their inability to find vocabulary tokens connected to the last words of the given sentences.

Task 4

Here, we created two additional functions in order to implement the context-aware spelling corrector using both language models. The functions and their descriptions are presented below.

Function 1 - compute_edit_distance_probability

This function calculates probabilities inversely proportional to edit distances for a given list of candidate words.

The function follows these steps:

- Computes an inverse probability for each candidate using the formula:

$$P = \frac{1}{1 + \text{Levenshtein edit distance}}$$

This ensures that candidates with smaller edit distances have higher probabilities.

- Normalizes the probabilities by dividing each by the total sum of computed probabilities.
- Removes the edit distance value from each candidate, keeping only the word and its final probability.
- Returns the input list, where each candidate is now associated with its normalized probability.

Function 2 - find_candidates

This function is used to find candidates for a specific word based on vocabulary and using the Levenshtein distance.

The function follows these steps:

- Iterates through the vocabulary and computes the Levenshtein distance between the input word and each word in the vocabulary.
- Filters words that have an edit distance less than or equal to `max_distance`.
- Computes probabilities for the valid candidates using `compute_edit_distance_probability`.
- Returns the list of candidates, each paired with its normalized probability.

Function 3 - apply_spelling_corrector

This function applies a spelling correction algorithm to an input sentence by identifying and replacing potentially misspelled words using edit distance and the bigram-trigram language models. It performs beam search to explore only the most promising sentence correction paths.

The algorithm follows these steps:

- Finds spelling correction candidates for each word in the input sentence using `find_candidates`, which leverages Levenshtein distance.
- Initializes a beam search queue with an empty sentence prefix `["<s>", "<s>"]` and some starting scores.
- Iterates through each word in the sentence
  - If a word has no correction candidates, it is retained as-is.
  - If candidates exist:
    - It expands possible sequences by expanding them with each candidate.
    - Computes the bigram and trigram logarithmic probabilities and keeps the maximum and updates the total sum.
    - Computes edit distance probabilities and updates the total sum.
- Retains the `beam_width` sequences based on a weighted combination (`λ1` and `λ2`) of language model and edit distance scores. Below, is the relevant formula:

$$\lambda_1 \log P(t_1^k) + \lambda_2 \log P(w_1^k|t_1^k)$$

- Returns the most probable corrected sentence as a string.

To finalize the spelling correction algorithm, we fine-tuned the lambda parameters in Task 6 where, we have created two artificial datasets — a development one for tuning and a test one for evaluating the algorithm.

Task 5

Here, we created two artificial datasets to evaluate the context-aware spelling corrector. The first, called the development dataset, consists of 40 sentences and was used to tune the lambda parameters (Task 6). The second, called the test dataset, was used to assess the performance of the spelling corrector (Task 6).

In both datasets, we introduced noise by randomly replacing each non-space character with another non-space character at a small probability. To achieve this, we implemented a function, described below.

Function 1 - add_noise_to_sentences

This function introduces random character noise into sentences by modifying some characters within words.

The function follows these steps:

- Iterates through each sentence in the provided list.
- Iterates through each word in the sentence.
- Iterates through each character in the word.
  - 80% chance - the character remains unchanged.
  - 20% chance - The character is randomly replaced with a random lowercase letter.
- Constructs a noisy version of each word and then reconstructs the full noisy sentence.
- Returns a list of noisy sentences, preserving the structure of the input.

Task 6

Here, we fine-tuned the lambda parameters of the context-aware spelling corrector and tested it on the constructed test dataset using the optimal lambda values. To achieve this, we first defined a function to evaluate the spelling corrector's performance using Word Error Rate (WER) and Character Error Rate (CER). The function's description is provided below.

Function 1 - evaluate_spelling_corrector

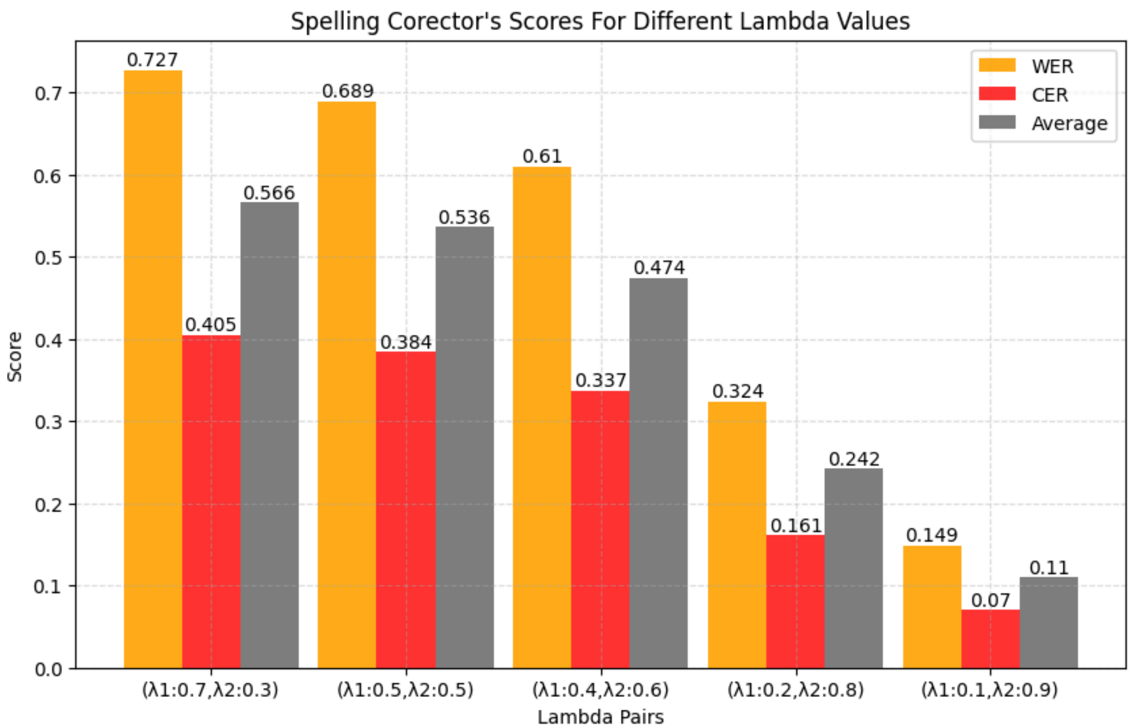
This function evaluates the performance of a spelling correction system using Word Error Rate (WER) and Character Error Rate (CER) metrics.

The function follows these steps:

- Iterates through all sentences.
- Computes the Word Error Rate (WER) and Character Error Rate (CER).

- Averages the WER and CER across all sentences.
- Returns the final WER and CER scores, rounded to three decimal places.

For the tuning process, we tested multiple pairs of lambda values by applying the spelling corrector to the development dataset and getting the corresponding scores using the `evaluate_spelling_corrector` function. We then selected the pair with the lowest score values. Below is a bar chart illustrating the spelling corrector's performance across different lambda parameter values.



As shown in the bar chart, the optimal values for the lambda parameters are λ1 = 0,1 and λ2 = 0,9.

Finally, using the optimal lambda values, we evaluated the spelling corrector on the test dataset. Below, we present the corresponding scores along with examples of correctly and incorrectly 'corrected' misspelled sentences.

Scores

WER	CER	Average
0.12	0.05	0.085

Few Examples

Original Test Sentences	Noisy Test Sentences	Corrected Test Sentences
aluminium is sold in dollars	ahuminium is sold ix qollars	aluminium is sold in dollars
prior year results restated	pcmor yeak results eesdatwd	prior year results restated
economist with merrill lynch capital markets	economist iith merrill lench capwtal maryets	economist with merrill lynch capital markets
both cited market conditions	both rited mardet conqibkonq	both sides market conditions
one rocket was fired but missed	nne rocket was fired kgt misfed	the market was fired yet missed
previous prices in parentheses	urevious pnices in palehthejes	previous prices in palehthejes

In the first three cases, the spelling corrector performs well, accurately correcting the misspellings. However, in the last three cases, it struggles to generate the correct sentence forms.