Text Analytics
(MSc Data Science)

ASSIGNMENT 1 - N-GRAM LANGUAGE MODELS

Nikolaos Vitsentzatos| f3352405
Stylianos Giagkos | f3352410

Nikolaos Vitsentzatos| f3352405
Stylianos Giagkos | f3352410

Table of Contents

Nikolaos Vitsentzatos| f3352405
Stylianos Giagkos | f3352410

Introduction

Analysis and Commentary on the Implementation of N-Gram Models in the NLTK Corpus

This report presents a detailed analysis and commentary on the implementation of n-gram models using the NLTK corpus, as part of the assignment for the course "Text Analytics - Assignment 1." N-gram models are fundamental in text analysis as they represent a way of capturing statistical dependencies between words in a text, and are essential for tasks like language modeling, text prediction, and information retrieval.

In this assignment, the implementation of n-gram models was carried out using the Natural Language Toolkit (NLTK), a comprehensive library for working with human language data. The NLTK corpus provides access to a wide range of datasets, enabling the creation of various n-gram models. Through this implementation, various challenges and nuances of working with n-grams, such as tokenization, frequency counting, and smoothing techniques, were explored.

This assignment as well as the future assignments of this course will be hosted on the Github repository.

Final code can be executed on the Colab notebook.

Nikolaos Vitsentzatos| f3352405
Stylianos Giagkos | f3352410

3. (i) Implement (in any programming language) a bigram and a trigram language model for sentences, using Laplace smoothing (slide 8) or optionally (if you are very keen) KneserNey smoothing (slides 50–51), which is much better. In practice, n-gram language models compute the sum of the logarithms of the n-gram probabilities of each sequence, instead of their product (why?) and you should do the same. Assume that each sentence starts with the pseudo-token *start* (or two pseudo-tokens *start1*, *start2* for the trigram model) and ends with the pseudo-token *end*. Train your models on a training subset of a corpus (e.g., a subset of a corpus included in NLTK – see http://www.nltk.org/). Include in the vocabulary only words that occur, e.g., at least 10 times in the training subset. Use the same vocabulary in the bigram and trigram models. Replace all out-of-vocabulary (OOV) words (in the training, development, test subsets) by a special token *UNK*. Alternatively, you may want to use BPEs instead of words (obtaining the BPE vocabulary from your training subset) to avoid unknown words. See Section 2.5.2 ("Byte-Pair Encoding") of the 3rd edition of Jurafsky & Martin's book

1. Bigram and Trigram Models Implementation

1.1. Data Preprocessing

- **Tokenization:** The first step involves tokenizing the text data, which splits the text into individual words (or tokens). This is done using the `nltk.corpus.reuters.sents()` method that provides sentences as lists of words.
- **Lowercasing:** All words are converted to lowercase to ensure that variations of the same word (e.g., "Apple" and "apple") are treated as the same token.
- **Start/End Tokens:** For n-gram models (bigram and trigram), start and end tokens (`*start*`, `*start1*`, `*start2*`, and `*end*`) are added to the sentences. These tokens help in capturing the context when modeling sequences of words, such as handling the start and end of a sentence in n-grams.
- **Filtering Low-Frequency Words:** Words that appear less than a specified minimum frequency (`min_freq=10`) are excluded from the vocabulary to filter out noise and reduce the dimensionality. This helps in focusing on more frequent and relevant words in the model.

1.2. N-gram Model Construction

- **Bigram and Trigram Models:** For n = 2 (bigram) and n = 3 (trigram), the program extracts n-grams from the preprocessed corpus. An n-gram is a sequence of `n` consecutive words. For example, in a bigram model, consecutive pairs of words such as "the cat" or "cat sat" would be considered.
- **Frequency Count of N-grams:** The frequency of occurrence for each n-gram in the corpus is computed. These counts are stored in a dictionary to keep track of how often each n-gram appears.

1.3. Laplace Smoothing

- **Laplace Smoothing:** This technique is used to handle the problem of zero probabilities for unseen n-grams. In traditional probability estimation, if a word sequence (n-gram) is not observed during training, its probability would be zero, which is not helpful. Laplace smoothing adds 1 to the count of all n-grams, ensuring that no probability is zero. It adjusts the probability estimates for observed n-grams and also includes the total number of unique words in the vocabulary.

1.4. Log Probability Calculation

- **Log Probability of Train Data Sentences:** For each sentence of the train data , the log probability of the sentence given the n-gram model is computed. The log probability is preferred over direct probability calculations, as it avoids underflow issues (very small values) and is easier to work with when multiplying many probabilities. The log probability is the sum of the log of the probabilities of each n-gram in the sentence.

1.4.1. Why do n-gram language models compute the sum of the logarithms of the n-gram probabilities instead of their product?

N-gram language models use the **sum of logarithms** instead of the product of probabilities for **numerical stability** and practical reasons:

Avoiding Numerical Underflow:

- Probabilities in language models are often very small (e.g., 10^6).
- Multiplying many small numbers results in values too small for the computer to handle, leading to **numerical underflow**.

Logarithms Prevent Problems with Zero Probabilities:

- Directly multiplying probabilities of unseen events (zero probability) results in invalid calculations.
- **Logarithms** of small probabilities can be computed even for unseen n-grams (using smoothing), and avoid the issue of log(0)

Easier to Optimize:

- **Log probabilities** are additive, making them easier to work with when optimizing models.
- The sum of log-probabilities is more **intuitive** for comparing the likelihood of different sequences.

Computational Efficiency:

- Adding logs is computationally more efficient than multiplying very small numbers, and it's simpler and more stable, especially for large datasets.

1.5. Model Evaluation

- **Training Set Evaluation:** The log probabilities are computed for 10 random sentences from the training set to give an insight into how well the model fits the training data.

3. (ii) Estimate the cross-entropy and perplexity of your two models on a test subset of the corpus, treating the entire test subset as a single sequence of sentences, with *start* (or *start1*, *start2*) at the beginning of each sentence, and *end* at the end of each sentence. Do not include probabilities of the form P(*start*|...) or P(*start1*|...), P(*start2*|...) in the computation of cross-entropy and perplexity, since we are not predicting the start pseudotokens; but include probabilities of the form P(*end*|...), since we do want to be able to predict if a word will be the last one of a sentence. You must also count *end* tokens (but not *start*, *start1*, *start2* tokens) in the total length N of the test corpus.

2. Perplexity and Entropy Calculations

2.1. Laplace Smoothing (Probability Estimation)

- **Laplace Smoothing:** This function `estimate_probabilities` calculates the smoothed probabilities for each n-gram using Laplace smoothing. For each n-gram, it computes the probability as the count of the n-gram plus 1, divided by the frequency of its context (the preceding `n-1` words) plus the size of the vocabulary. This ensures that unseen n-grams will have a non-zero probability.
- The smoothed probability is calculated as: $P(w_n|w_{n-1},w_{n-2},...)= \frac{count(w1,w2,...,wn)+1}{count(w1,w2,...,wn-1)+V}$ where V is the vocabulary size.

2.2. Log Probability Calculation

- **Log Probability of Sentences:** The `compute_log_probabilities` function computes the log probability for each sentence in the development set. This is done by iterating through each sentence, extracting n-grams, and summing the log of their probabilities. The function skips the start tokens and counts the total number of tokens (excluding start tokens but including end tokens). For each unseen n-gram, the probability is set to a smoothed value (using Laplace smoothing).

2.3. Cross-Entropy and Perplexity

- **Cross-Entropy:** Cross-entropy is a measure of the difference between the true distribution of the language and the model's predicted distribution. It is calculated as the negative log-likelihood of the sentence probabilities normalized by the total number of tokens

$$H(p)=-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i)$$

  where N is the total number of tokens, and $P(w_i)$ is the probability of the token w_i

- **Perplexity:** Perplexity is a measure of how well the language model predicts a sample and is derived from cross-entropy. It is the exponentiation of the cross-entropy: $PPL=2^{H(p)}$

  A lower perplexity indicates better predictive power of the model.

2.4. Laplace Smoothing Model Evaluation

- **Development Set Evaluation:** The function `evaluate_ngram_models_on_dev` processes both bigram and trigram models. For each model:
  - It preprocesses the training set, estimates probabilities using Laplace smoothing, and constructs the n-gram model.
  - It processes the development set and computes the log probability sum for all sentences.
  - The cross-entropy and perplexity are then computed based on the log probability of the entire development set.
- The results are printed, including the log probability sum, cross-entropy, and perplexity for both bigram and trigram models.

```
Evaluating 2-gram model on development set...
Log probability sum for 2-gram model: -1243677.2647457323
Cross-entropy for 2-gram model: 6.8489333748876975
Perplexity for 2-gram model: 115.27480179098079

Evaluating 3-gram model on development set...
Log probability sum for 3-gram model: -2802978.4294866496
Cross-entropy for 3-gram model: 14.984461744618807
Perplexity for 3-gram model: 32416.97260618198
```

2.5. Additive-α Smoothing for Probability Estimation:

- **Additive-α smoothing** is used to estimate probabilities for unseen n-grams. This is done using a parameter `alpha`, which is a smoothing factor.
- The probabilities for each n-gram are computed as:

$$P(w_n|w_{n-1},w_{n-2},\ldots)= \frac{\text{count}(w1,w2,\ldots,wn)+\alpha}{\text{count}(w1,w2,\ldots,wn-1)+\alpha|V|}$$

where:

- $\text{count}(w1, w2, \ldots, wn)$ is the count of the n-gram,
- $\text{count}(w1, w2, \ldots, wn-1)$ is the count of the context (n-1 gram),
- α is the additive smoothing parameter,
- $|V|$ is the vocabulary size (the number of unique words).

2.6. Additive-α Smoothing Model Evaluation:

- The best alpha for perplexity and cross-entropy is selected for each n-gram model by finding the alpha that minimizes these metrics.
- The results (perplexity and entropy vs. alpha) are visualized using **matplotlib**. The best alpha for each metric is highlighted on the plot.

2.6.1. Plots

- The plots show the variation in **perplexity** and **entropy** with respect to different values of the smoothing parameter alpha.

- The optimal value of alpha is marked on the plot for both perplexity and entropy, and the best alpha values for each n-gram model are printed.

Model	Best Alpha (Perplexity)	Best Perplexity	Best Alpha (Entropy)	Best Entropy
2-gram	0.001	2.29602	0.001	1.199135
3-gram	0.001	2356.22	0.001	11.20226

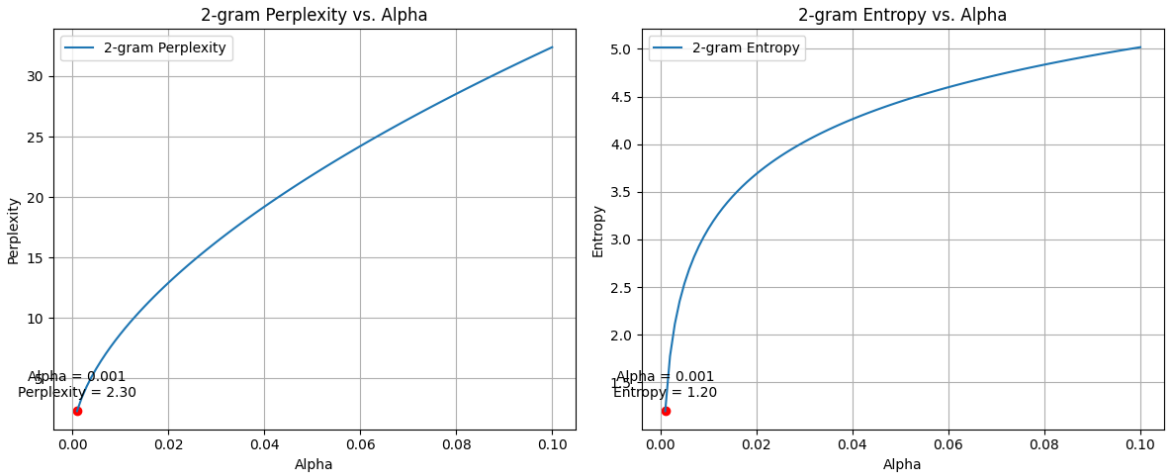

Figure 1 Bigram Model Perplexity vs Alpha and Entropy vs Alpha plots

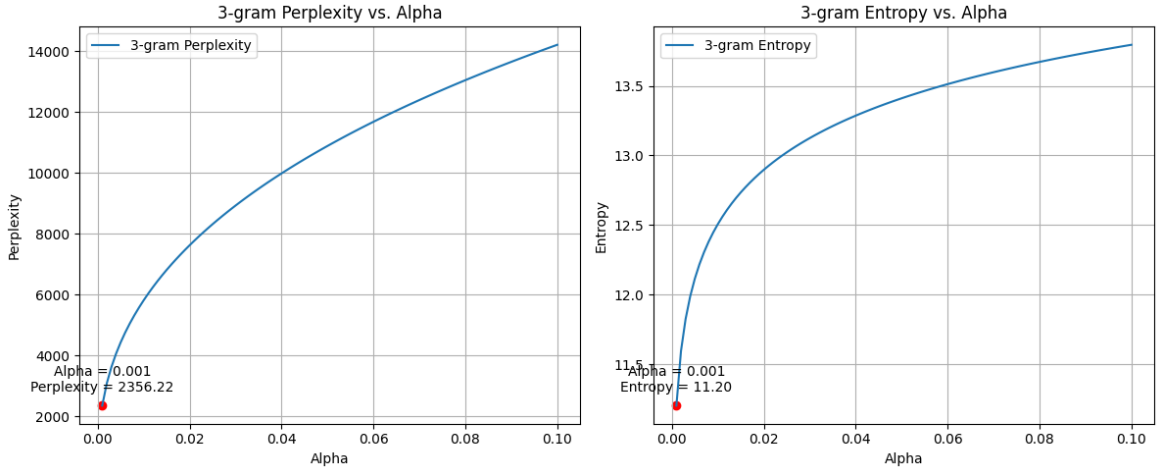

Figure 2 Trigram Model Perplexity vs Alpha and Entropy vs Alpha plots

3. (iii) Write some code to show how your bigram and trigram language models can autocomplete an incomplete sentence. For example, given "I would like to commend the" (slide 44), generate completions like "rapporteur on his work *end*" or "president of the Commission on his intervention *end*". You can simply use the most probable next word (according to your language model) at each time-step. If you are keen, you may also want to use beam search, or methods like top-K or nucleus sampling, to improve the diversity of the texts you generate.1 Confirm (showing some examples of generated texts) that your trigram model generates more fluent texts than your bigram model.

3. Auto Complete Task

The code employs two primary algorithms—**bigram** and **trigram models**—for generating sentence completions based on a given input, leveraging **Laplace smoothing**, **nucleus sampling**, and **probability calculations**.

3.1. Bigram and Trigram Models:

- **Bigrams** and **trigrams** are models that predict the probability of a word appearing based on the previous one (bigram) or two (trigram). They work by counting the occurrences of consecutive word pairs (for bigrams) or triples (for trigrams) in a corpus.
- In this approach, the corpus is first processed to extract **bigrams** (pairs of consecutive words) and **trigrams** (triplets of consecutive words). The counts of these pairs and triplets are stored and used to calculate the probability of a word occurring given its predecessor(s).

3.2. Laplace Smoothing:

- Laplace smoothing is applied to avoid the problem of zero probabilities for unseen word combinations. By adding a small constant (`a=0.001`) to the frequency counts, it ensures that all word pairs/triplets have a non-zero probability, which is particularly useful for generative models.

3.3. Nucleus Sampling (Top-p Sampling):

- **Nucleus sampling** is used in the autocomplete functions for both the bigram and trigram models. Instead of always choosing the word with the highest probability (greedy selection), nucleus sampling selects the next word based on a cumulative probability distribution. The cumulative probability is cut off at a threshold (`p=0.9`), meaning the model will only consider the most probable words whose cumulative probability exceeds this threshold. This allows for more diversity in the generated text.

3.4. Autocompletion:

- The **autocomplete functions** generate the next word(s) in a sequence, iteratively building a sentence. They use the last word (bigram model) or last two words (trigram model) to predict the next word, applying the bigram/trigram probability models to determine the most likely word. The sentence is extended word-by-word until an **end token** (`<end>`) is encountered or the maximum sentence length is reached.

3.5. Auto completion results

The **bigram** and **trigram model completions** presented here exhibit some interesting results, though they also highlight certain challenges in the current model's performance.

3.5.1. Bigram Model Results:

- **Input:** "The company has been"
  Completion: "the company has been in net shr loss 469 , 000 sales . s beef subsidy , 996 , the"
  Comment: The sentence starts off well but quickly becomes nonsensical with numbers and jargon.
- **Input:** "He is planning to"
  Completion: "he is planning to stick because the chairman alan residential and exports and serious . 0 . s decision by"
  Comment: Begins logically but ends with irrelevant terms and repetition.
- **Input:** "I would like to"
  Completion: "i would like to discuss a bbl , the most likely the company said it ." citicorp unit buys british"
  Comment: Starts off sensibly, but gets interrupted with unusual abbreviations and company names.

3.5.2. Trigram Model Results:

- **Input:** "The company has been very"
  Completion: "the company has been very shearson qintex wendy critical postings 1983 broke salmonella 972 strategy naira authorised dealing spectra contribution"
  Comment: More detailed but still introduces irrelevant or confusing terms.
- **Input:** "We are looking forward to"
  Completion: "we are looking forward to next arkla cutting textiles 536 12 disagreed proposal pessimistic withdraw founder brierley sachs lb nasdaq"
  Comment: Unrelated terms appear, making it hard to follow.
- **Input:** "I haven't seen the report"
  Completion: "i haven't seen the report said zambian use gardner wtc fiscal series miller horner adhesives regulated offering slower repurchase disclosed"
  Comment: The result is filled with random names and terms, lacking coherence.

3.5.3. General Observations:

Both models generate text that lacks coherence, with some sentences containing unrelated or nonsensical terms. The bigram model struggles more with basic grammar, while the trigram model adds more context but still faces issues with irrelevant phrases.

3. (iv) Develop a context-aware spelling corrector (for both types of errors, slide 14) using your bigram and/or trigram language model, a beam search decoder (slides 26-32), and the formulae of slide 20. As on slide 20, you can use the inverse of the Levenshtein distance between wi,ti as P(wi|ti). If you are very keen, you may want to use better estimates of P(wi|ti) that satisfy ∑iP(wi|ti)=1. You may also want to use: t1=argmaxλ1logP(t1)+λ2 logP(w1|t1)

4. Context Aware Spelling Corrector

4.1. Levenshtein Distance (Edit Distance):

- This metric is used to calculate the similarity between two words by counting the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other. The inverse of this distance is used in the candidate generation to identify words that are more similar to the input word.

4.2. N-gram Language Model:

- In this case, a trigram model (or bigram if specified) is used to estimate the probability of a word occurring given the previous words in the sentence. This helps ensure that the corrected word fits well into the context of the sentence. The language model probability (`lm_prob`) is computed based on the previous tokens and the candidate word.

4.3. Beam Search Decoder:

- This search algorithm is used to explore possible word corrections and select the best candidate. The beam search evaluates multiple candidates at each step, keeping track of the top `beam_width` candidates based on a scoring function. This helps find the most likely correction by balancing both word similarity (via Levenshtein distance) and language model probability.

4.4. Score Calculation:

- The score for each candidate is computed as a weighted sum of the logarithms of the language model probability (`lm_prob`) and the inverse of the Levenshtein distance (`lev_distance`). The weights (`lambda_1 and lambda_2`) control the influence of the language model and the word similarity. A higher lambda for `lambda_2` will prioritize word similarity, while a higher lambda_1 will prioritize language model context.

4.5. Spelling Correction:

- The function `correct_spelling_using_beam_search` applies the beam search decoder iteratively to each word in the input sentence, using the previous tokens for context and selecting the most probable word correction for each step. The sentence is progressively corrected by replacing each word with its best candidate.

```
Incorrect sentence: This is a smple sentnce with erors.
Corrected sentence: his is a ample settle with frost
```

3. (v) Create an artificial test dataset to evaluate your context-aware spelling corrector. You may use, for example, the test dataset that you used to evaluate your language models, but now replace with a small probability each non-space character of each test sentence with another random (or visually or acoustically similar) non-space character (e.g., "This is an interesting course." may become "Tais is an imterestieg kourse").

5. Artificial Dataset for model evaluation

5.1. Transform to Continuous Sentences:

- The `transform_to_continuous_sentences` function takes a list of tokenized sentences and combines them into continuous sentences. It joins the words with spaces and ensures that punctuation is correctly spaced (using regex to remove spaces before punctuation marks like .,?!).

5.2. Add Random Noise:

- The `add_random_noise` function introduces noise to the sentences by randomly replacing 1 or 2 characters in each word with randomly chosen letters. This simulates the kind of errors or distortions that can occur in real-world noisy data (e.g., OCR errors, typing mistakes).

5.3. Correct Noisy Sentences Using Beam Search:

- The `correct_sentences_for_noisy_data` function applies the previously discussed beam search algorithm to correct the noisy sentences. It iterates over a subset of noisy sentences, correcting each using the `correct_spelling_using_beam_search` function, and prints out the original, noisy, and corrected versions for comparison.
- The beam search algorithm is parameterized with `lambda_1` and `lambda_2` values that control the balance between language model probability (`lambda_1`) and Levenshtein distance (`lambda_2`). The function is run with different values of `lambda_1` and `lambda_2` to see how the results change depending on the weight given to the language model vs. the word similarity.

5.3.1. Lambdas tuning

For `lamdba_1 = 0.5` and `lamdba_2 = 0.5`

```
correct_sentences_for_noisy_data(noisy_sentences,continuous_sentences, ngram_model='trigram', beam_width=5, vocab=vocab, lambda_1=0.5, lambda_2=0.5)
```

```
Ground Truth:It said the date of the special meeting has not yet been determined.
Incorrect sentence: ht saiL vhG dale uf teo spectCl meeming Aas noU yRt bevn dytermined.
Corrected sentence: at said the sale of the total outstanding . nov yet bean determined

Ground Truth:" We are still concerned about S and W Berisford being long - term owners of British Sugar," a spokesman said.
Incorrect sentence: A Wk Zre Cthll cQncernId aboRt o afd H BerisfQrd heing loEg Q tJrz owNeas af rritKsh SugarZ" I spbkesman saYd.
Corrected sentence: p k re toll concerned about the same as in the long - term debt of the year . spokesman ,
```

For `lamdba_1 = 0.7` and `lamdba_2 = 0.3`

```
# For lamdba_1 = 0.7 and lamdba_2 = 0.3
correct_sentences_for_noisy_data(noisy_sentences,continuous_sentences, ngram_model='trigram', beam_width=5, vocab=vocab, lambda_1=0.7, lambda_2=0.3)
```

```
Ground Truth:It said the date of the special meeting has not yet been determined.
Incorrect sentence: ht saiL vhG dale uf teo spectCl meeming Aas noU yRt bevn dytermined.
Corrected sentence: at said the sale of the total outstanding . nov yet bean determined

Ground Truth:" We are still concerned about S and W Berisford being long - term owners of British Sugar," a spokesman said.
Incorrect sentence: A Wk Zre Cthll cQncernId aboRt o afd H BerisfQrd heing loEg Q tJrz owNeas af rritKsh SugarZ" I spbkesman saYd.
Corrected sentence: p k re toll concerned about the same as in the u . s . and british firms , boston -
```

For `lamdba_1 = 0.3` and `lamdba_2 = 0.7`

```
# For lamdba_1 = 0.3 and lamdba_2 = 0.7
correct_sentences_for_noisy_data(noisy_sentences,continuous_sentences, ngram_model='trigram', beam_width=5, vocab=vocab, lambda_1=0.3, lambda_2=0.7)
```

```
Ground Truth:It said the date of the special meeting has not yet been determined.
Incorrect sentence: ht saiL vhG dale uf teo spectCl meeming Aas noU yRt bevn dytermined.
Corrected sentence: at said the sale of the special meeting as an oil fee determined

Ground Truth:" We are still concerned about S and W Berisford being long - term owners of British Sugar," a spokesman said.
Incorrect sentence: A Wk Zre Cthll cQncernId aboRt o afd H BerisfQrd heing loEg Q tJrz owNeas af rritKsh SugarZ" I spbkesman saYd.
Corrected sentence: p k re toll concerned about a bid to berisford being long p term owners at british sugar , spokesman says
```

3. (vi) Evaluate your context-aware spelling corrector in terms of Word Error Rate (WER) and Character Error Rate (CER), using the original (before character replacements) form of your test dataset of question (v) as ground truth (reference sentences), and averaging WER (or CER) over the test sentences. CER is similar to Word Error Rate (WER), but operates at the character level.

6. Word Error Rate (WER) and Character Error Rate (CER)

6.1. Definition of Word Error Rate (WER)

Word Error Rate (WER) is a metric used to evaluate the performance of speech recognition or text correction systems by measuring the number of errors at the word level. It compares the predicted sentence to the ground truth by counting the number of insertions, deletions, and substitutions required to convert the predicted sentence into the ground truth.

Formula:

WER = (S + D + I) / N

Where:

- S = Number of substitutions (where a word in the predicted sentence differs from the ground truth)
- D = Number of deletions (where a word in the ground truth is missing in the predicted sentence)
- I = Number of insertions (where an extra word is present in the predicted sentence)
- N = Total number of words in the ground truth sentence

6.2. Definition of Character Error Rate (CER)

Character Error Rate (CER) is similar to WER, but it evaluates errors at the character level instead of the word level. CER counts the number of character-level insertions, deletions, and substitutions required to transform the predicted sentence into the ground truth.

Formula:

CER = (S_char + D_char + I_char) / N_char

Where:

- S_char = Number of character substitutions (where a character in the predicted sentence differs from the ground truth)
- D_char = Number of character deletions (where a character in the ground truth is missing in the predicted sentence)
- I_char = Number of character insertions (where an extra character is present in the predicted sentence)
- N_char = Total number of characters in the ground truth sentence (ignoring spaces)

6.3. Word Error Rate (WER) Calculation

- **WER Formula:** The WER is calculated by comparing the reference (ground truth) sentence with the hypothesis (corrected sentence) using dynamic programming.
- The function `compute_wer` uses dynamic programming to compute the minimum number of edits needed to transform the hypothesis into the reference sentence. The function returns the error rate as a fraction of the number of reference words.

6.4. Character Error Rate (CER) Calculation

- **CER Formula:** Similar to WER, CER is calculated at the character level by comparing the reference (ground truth) string and the hypothesis string.
- The function `compute_cer` computes the minimum edit distance at the character level, excluding spaces, to give a more fine-grained measure of errors.

6.5. Beam Search for Spelling Correction

- **Beam Search:** The function `evaluate_corrector` assumes a spelling correction model using **beam search** for hypothesis generation. The beam search algorithm explores multiple possible word sequences (hypotheses) and selects the most likely ones based on a predefined beam width (the number of candidate sequences kept at each step).
- The beam search is influenced by parameters such as the choice of n-gram model (`bigram` or `trigram`), beam width, and two weighting factors (`lambda_1`, `lambda_2`) for combining language model probabilities and character-level spellings.

6.6. Evaluation Framework

- **Evaluation Function** (`evaluate_corrector`)**:** This function iterates through a specified number of test sentences, computes the WER and CER for each, and prints the results. It returns the average WER and CER over the specified number of sentences.
- For each test case, it performs the following:
    - Takes noisy sentences (containing errors).
    - Applies beam search-based correction to get a corrected sentence.
    - Computes WER and CER for the corrected sentence compared to the continuous (correct) sentence.

6.7. Hyperparameter Search

- **Evaluation for Different Parameter Combinations:** The `evaluate_and_generate_dataframe` function evaluates the performance of the spelling corrector across different combinations of the following parameters:
    - **ngram_model**: The choice between bigram or trigram models.
    - **beam_width**: The width of the beam search, controlling how many candidate hypotheses are considered.
    - **lambda_1, lambda_2**: The weighting factors for combining the language model (ngram model) and the character-level model during beam search.
- The function performs a grid search over these hyperparameters and computes the average WER and CER for each combination. The results are stored in a DataFrame for easier analysis.

6.8. Generating and Returning Results

- After evaluating all combinations, the function `evaluate_and_generate_dataframe` returns a DataFrame with the results for each combination of parameters (ngram model, beam width, lambda values). This DataFrame includes the average WER and CER for each set of parameters.
- **Printing Results:** The code prints the row with the minimum average WER and CER to identify the best-performing configuration for spelling correction.

6.9. Output Example:

- The function `evaluate_and_generate_dataframe` will output a DataFrame with columns such as `ngram_model`, `beam_width`, `lambda_1`, `lambda_2`, `avg_wer`, and `avg_cer`. The row with the minimum `avg_wer` and `avg_cer` can be printed to highlight the best-performing configuration for correcting noisy sentences.

6.9.1. Model with minimum avg_wer:

Best Model	
ngram_model	bigram
beam_width	5
lambda_1	0.1
lambda_2	0.9
avg_wer	0.527316
avg_cer	0.220229

6.9.2. Row with minimum avg_cer:

Best Model	
ngram_model	**bigram**
beam_width	**5**
lambda_1	**0.1**
lambda_2	**0.9**
avg_wer	**0.527316**
avg_cer	**0.220229**

Evaluating with parameters: ngram_model=bigram, beam_width=5, lambda_1=0.1, lambda_2=0.9

Average WER over 50 sentences: 0.5204

Average CER over 50 sentences: 0.2502