n-gram Language Models

Aluiona Mancho f3322405 alv.mantso@aueb.gr / aluiona.mantso@gmail.com
Anna Chatzipapadopoulou f3322411 ann.chatzipapadopoul@aueb.gr / annachatzipap@gmail.com

Natural Language Processing, 2024-2025[1]

Notebook link

For this assignment we used the Brown Corpus. From the diverse categories available in the Brown Corpus, we selected the 'romance' category.

Data preprocessing:

The class TextProcessor handles the preprocessing of the data. It features the following methods:

- `tokenize_corpus(corpus)`: Tokenizes the corpus by splitting it into sentences, and then tokenizing each sentence into words.
- `create_vocabulary(tokenized_corpus, occurrences_threshold=10)`: The vocabulary is derived from the training set by retaining only those words that appear at least occurrences_threshold times, specifically set to a minimum of 10 occurrences. This filtering process ensures that the vocabulary focuses on more frequently occurring words, enhancing its relevance for the model.
- `process_tokenized_corpus(tokenized_corpus, vocabulary)`: Processes the tokenized corpus by replacing words not in the vocabulary with '*UNK*'.
- `introduce_noise(word, mapping, replace_prob = 0.1, delete_prob = 0.04)`: Introduces random noise to a given word by selectively replacing characters (according to the mapping) or deleting characters based on the specified probabilities. This method is utilized in the task of context-aware spelling correction, which we will refer to in subsequent sections.

Initially, the dataset is split into an 80% training set, with 10% allocated for testing and 10% for validation (development). Although the corpus is pre-tokenized, the `tokenize_corpus` method is included for completeness, even though it is not directly applied in this process. The vocabulary is then derived from the training set. Using this

[1] *Both team members engaged equally in all tasks, collaborating synchronously throughout the assignment.*

vocabulary, the `process_tokenized_corpus` method prepares the training, development, and testing sets. Note that neither the punctuation nor any stopwords are removed from the dataset.

Algorithms:

The Ngram class is an implementation of bigram and trigram language models with an option for interpolation between the two. It provides methods for training the n-gram model on a given vocabulary, as well as fine-tuning hyperparameters to optimize performance based on relevant metrics. Additionally, it supports functionalities for context-aware autocomplete and spelling correction.

- `train_ngram:` Counts unigrams, bigrams, and trigrams from the tokenized corpus to build frequency distributions for each n-gram type. For each sentence in the corpus, it extracts unigrams, bigrams, and trigrams using padding symbols (‘*start*’ and ‘*end*’) to mark the beginning and end of each sentence. These paddings allow the model to better understand sentence boundaries.
- `calculate_ngram_prob (self, tokens, n):` Calculates the smoothed probability of an n-gram using Laplace **α**-smoothing, providing a flexible approach to estimate probabilities for both bigram and trigram models depending on the specified n parameter. It applies additive (Laplace) smoothing to ensure that even rare or unseen n-grams receive a non-zero probability:

$$P_{\mathrm{bigram}}(\mathrm{w}_2|\mathrm{w}_1) = \frac{\mathrm{C}(\mathrm{w}_1, \mathrm{w}_2) + \boldsymbol{\alpha}}{\mathrm{C}(\mathrm{w}_1) + \boldsymbol{\alpha} \cdot |\mathrm{V}|}$$

$$P_{\mathrm{trigram}}(\mathrm{w}_3|\mathrm{w}_1, \mathrm{w}_2) = \frac{\mathrm{C}(\mathrm{w}_1, \mathrm{w}_2, \mathrm{w}_3) + \boldsymbol{\alpha}}{\mathrm{C}(\mathrm{w}_1, \mathrm{w}_2) + \boldsymbol{\alpha} \cdot |\mathrm{V}|}$$

- `sequence_ngram_prob(self, sentences_tokenized, n=None):` Calculates the cross-entropy, perplexity, and probability of a sequence of tokenized sentences utilizing the n-gram model, specifically supporting bigrams when n=2 and trigrams when n=3. If the n parameter is not specified, the method seamlessly interpolates between the bigram and trigram models, applying the specified **λ** as the interpolation weight.

$$P_{inter}(w_3|w_1, w_2) = \boldsymbol{\lambda} \cdot P(w_3|w_1, w_2) + (\mathbf{1} - \boldsymbol{\lambda}) \cdot P(w_3|w_2)$$

The cross entropy and perplexity are calculated as follows:

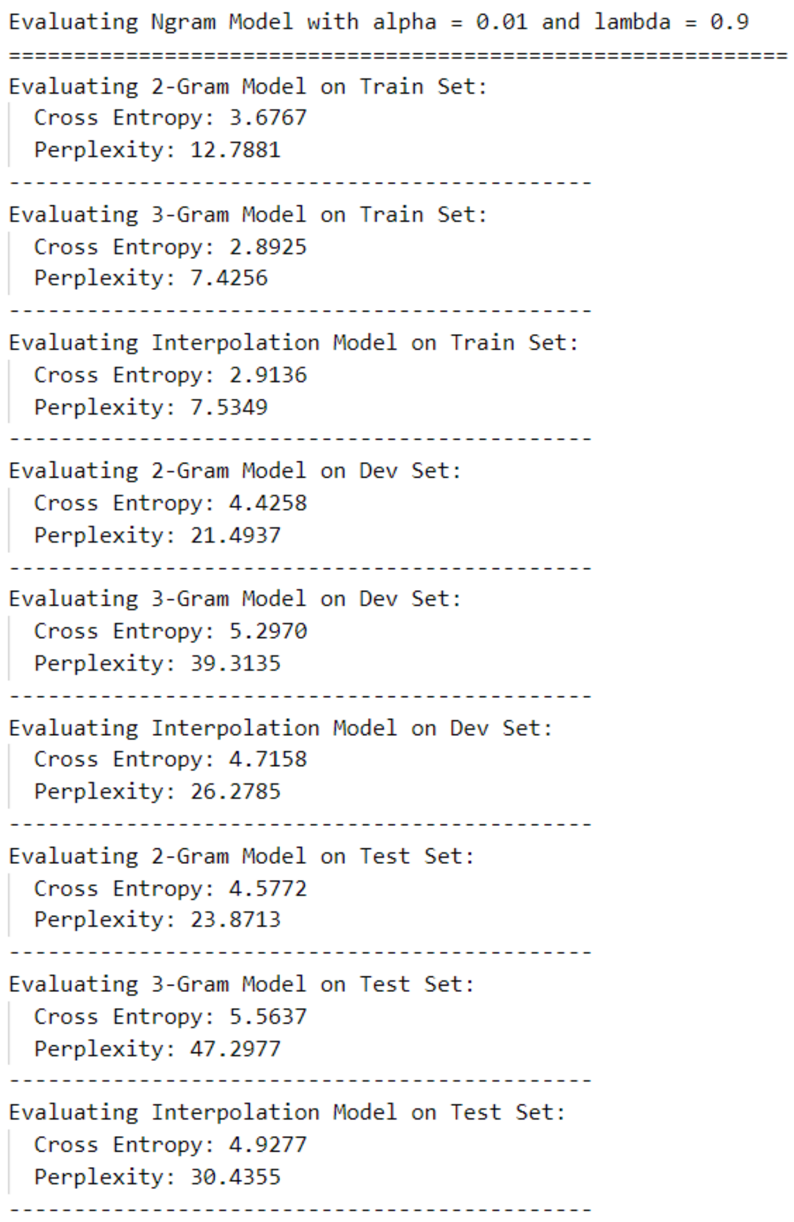
$$HC_{bigram} = -\frac{1}{N} \sum_{bigrams} log_2(P(w_2|w_1))$$

$$HC_{trigrams} = -\frac{1}{N} \sum_{trigrams} log_2\big(P(w_3|w_1, w_2)\big)$$

$$HC_{inter} = -\frac{1}{N} \sum_{trigrams} log_2\big(\boldsymbol{\lambda} \cdot P(w_3|w_1, w_2) + (\mathbf{1} - \boldsymbol{\lambda}) \cdot P(w_3|w_2)\big)$$

$$\text{Perplexity:} \quad P = 2^{HC} \quad \text{(in all cases)}$$

Using this evaluation method, we assess the performance of the bigram, trigram, and interpolated models across the training, development, and testing sets. The results are summarized as follows:



```
Evaluating Ngram Model with alpha = 0.01 and lambda = 0.9
============================================================
Evaluating 2-Gram Model on Train Set:
  Cross Entropy: 3.6767
  Perplexity: 12.7881
---------------------------------------------
Evaluating 3-Gram Model on Train Set:
  Cross Entropy: 2.8925
  Perplexity: 7.4256
---------------------------------------------
Evaluating Interpolation Model on Train Set:
  Cross Entropy: 2.9136
  Perplexity: 7.5349
---------------------------------------------
Evaluating 2-Gram Model on Dev Set:
  Cross Entropy: 4.4258
  Perplexity: 21.4937
---------------------------------------------
Evaluating 3-Gram Model on Dev Set:
  Cross Entropy: 5.2970
  Perplexity: 39.3135
---------------------------------------------
Evaluating Interpolation Model on Dev Set:
  Cross Entropy: 4.7158
  Perplexity: 26.2785
---------------------------------------------
Evaluating 2-Gram Model on Test Set:
  Cross Entropy: 4.5772
  Perplexity: 23.8713
---------------------------------------------
Evaluating 3-Gram Model on Test Set:
  Cross Entropy: 5.5637
  Perplexity: 47.2977
---------------------------------------------
Evaluating Interpolation Model on Test Set:
  Cross Entropy: 4.9277
  Perplexity: 30.4355
---------------------------------------------
```

Subsequently, the **α** (Laplace smoothing) and **λ** (interpolation) hyperparameters are fine-tuned (`fine_tune` method), with the goal of finding values that minimize the perplexity on the development set. The best values turned out to be **α**=0.001 and **λ**=0.4. Upon reassessing the models following fine-tuning, we obtain the following results, which reflect an improved performance:

```
Evaluating Ngram Model with alpha = 0.001 and lambda = 0.4
============================================================
Evaluating Interpolation Model on Train Set:
  Cross Entropy: 2.5379
  Perplexity: 5.8076
---------------------------------------------
Evaluating Interpolation Model on Dev Set:
  Cross Entropy: 4.2193
  Perplexity: 18.6266
---------------------------------------------
Evaluating Interpolation Model on Test Set:
  Cross Entropy: 4.4071
  Perplexity: 21.2157
---------------------------------------------
```

Autocompletion

There are two methods available to autocomplete an incomplete sentence:

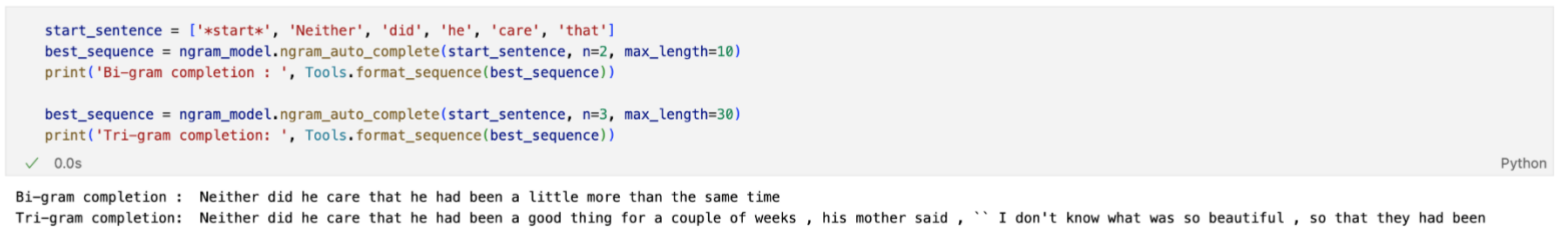
1. `ngram_auto_complete(self, start_sentence, n, max_length=20):`
   Autocompletes a given start sentence using an n-gram model up to a specified maximum length or until the '*end*' pseudo-token appears. The model selects each subsequent word by choosing the one with the highest probability.

2. `beam_search_autocomplete (self, initial_state, n, max_depth = 10, beam_width = 2):`
   Autocompletes a sentence using beam search with n-gram scoring. Expands each sequence by selecting the top beam_width candidates which have the highest cumulative probability.

Presented below are several examples of autocompletion generated using either the bigram or trigram model. The results clearly illustrate that the trigram model outperforms its bigram counterpart, producing more fluent and coherent text.

1. *Selecting the most probable next word* (`ngram_auto_complete`)

'Neither did he care that...'

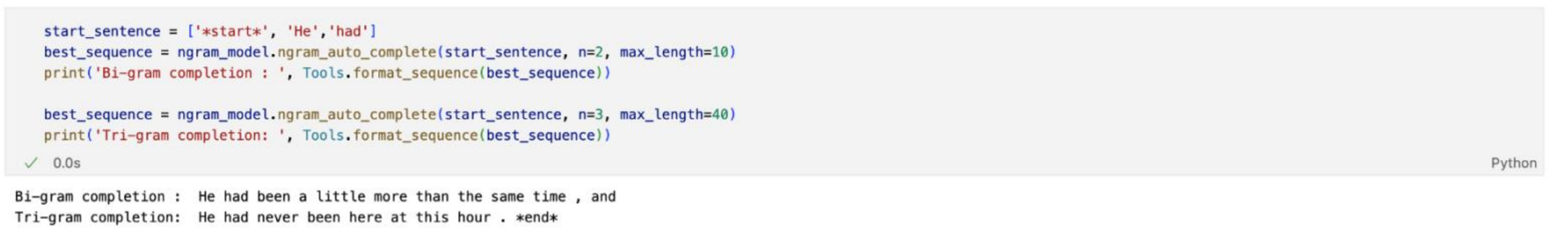


```
start_sentence = ['*start*', 'Neither', 'did', 'he', 'care', 'that']
best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=2, max_length=10)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=3, max_length=30)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```
✓ 0.0s Python

```
Bi-gram completion :  Neither did he care that he had been a little more than the same time
Tri-gram completion:  Neither did he care that he had been a good thing for a couple of weeks , his mother said , `` I don't know what was so beautiful , so that they had been
```

'He had...'

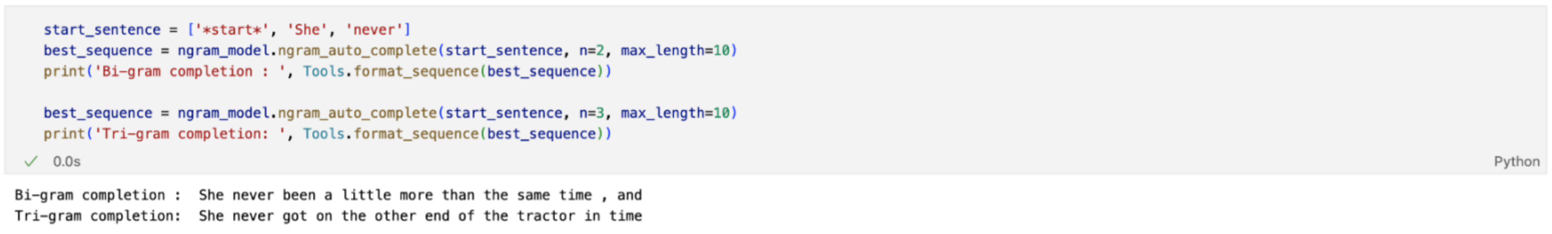


```
start_sentence = ['*start*', 'He','had']
best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=2, max_length=10)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=3, max_length=40)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```
✓ 0.0s Python

```
Bi-gram completion :  He had been a little more than the same time , and
Tri-gram completion:  He had never been here at this hour . *end*
```

'She never ...'

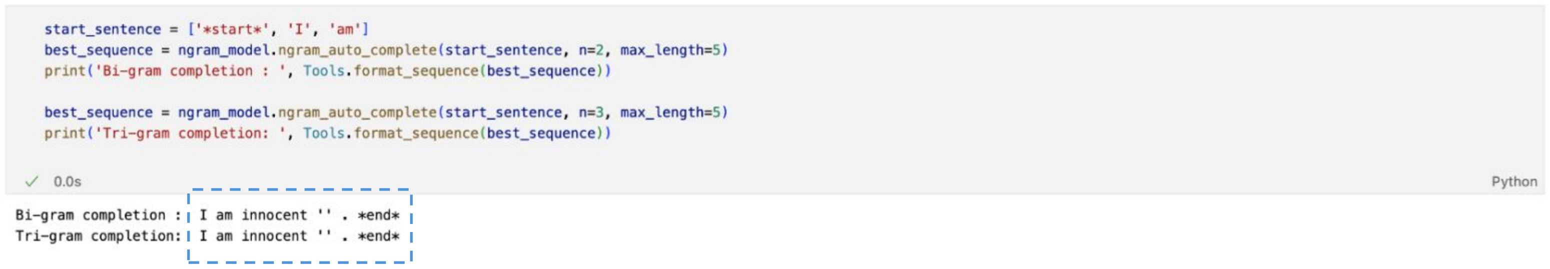


```python
start_sentence = ['*start*', 'She', 'never']
best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=2, max_length=10)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=3, max_length=10)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

```
Bi-gram completion :  She never been a little more than the same time , and
Tri-gram completion:  She never got on the other end of the tractor in time
```

'I am...'



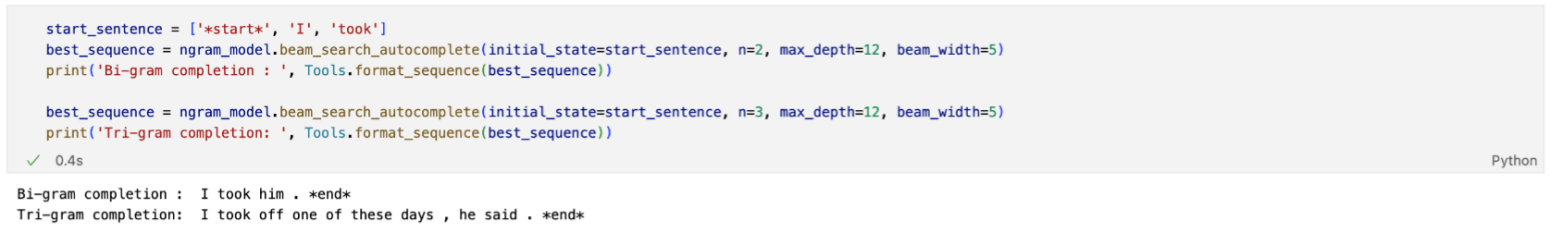
```python
start_sentence = ['*start*', 'I', 'am']
best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=2, max_length=5)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.ngram_auto_complete(start_sentence, n=3, max_length=5)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

```
Bi-gram completion :  I am innocent '' . *end*
Tri-gram completion:  I am innocent '' . *end*
```

2. *Using Beam Search* (`beam_search_autocomplete`)

'I took...'

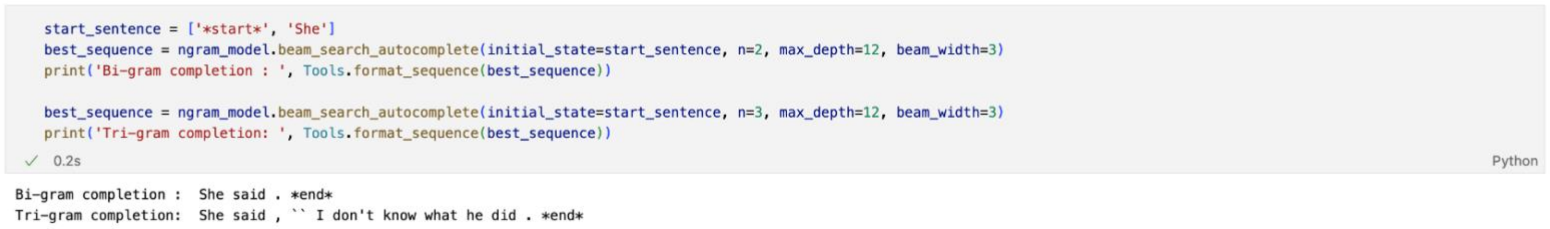


```python
start_sentence = ['*start*', 'I', 'took']
best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=2, max_depth=12, beam_width=5)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=3, max_depth=12, beam_width=5)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

```
Bi-gram completion :  I took him . *end*
Tri-gram completion:  I took off one of these days , he said . *end*
```

'She...'

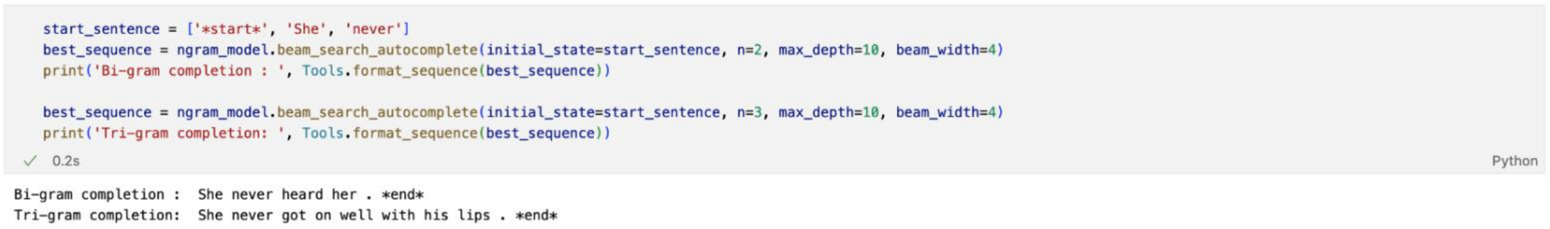


```python
start_sentence = ['*start*', 'She']
best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=2, max_depth=12, beam_width=3)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=3, max_depth=12, beam_width=3)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

```
Bi-gram completion :  She said . *end*
Tri-gram completion:  She said , `` I don't know what he did . *end*
```

'She never...'

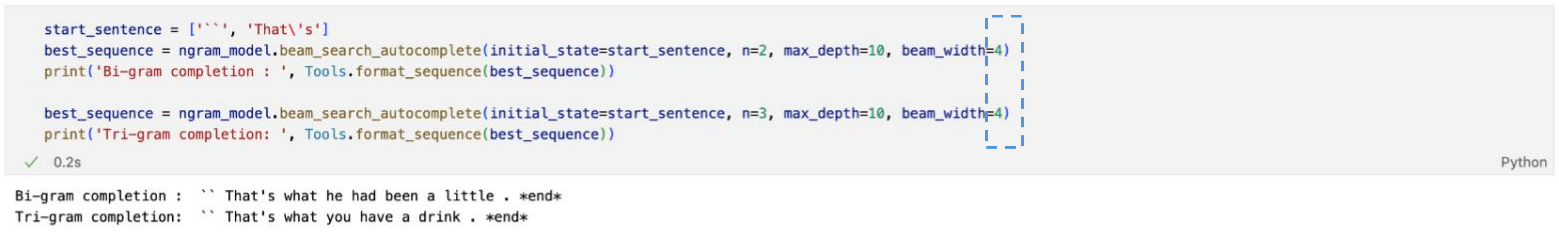


```python
start_sentence = ['*start*', 'She', 'never']
best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=2, max_depth=10, beam_width=4)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=3, max_depth=10, beam_width=4)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

```
Bi-gram completion :  She never heard her . *end*
Tri-gram completion:  She never got on well with his lips . *end*
```

’ “That’s...’

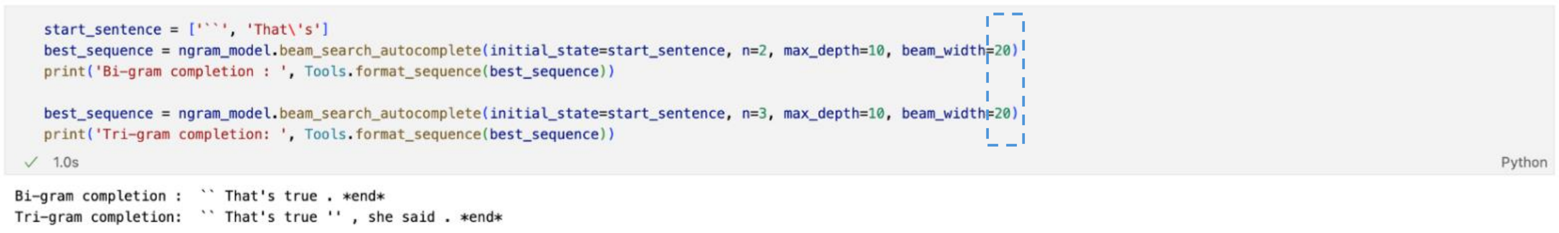


```
start_sentence = ['``', 'That\'s']
best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=2, max_depth=10, beam_width=4)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=3, max_depth=10, beam_width=4)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

✓ 0.2s Python

```
Bi-gram completion :  `` That's what he had been a little . *end*
Tri-gram completion:  `` That's what you have a drink . *end*
```



```
start_sentence = ['``', 'That\'s']
best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=2, max_depth=10, beam_width=20)
print('Bi-gram completion : ', Tools.format_sequence(best_sequence))

best_sequence = ngram_model.beam_search_autocomplete(initial_state=start_sentence, n=3, max_depth=10, beam_width=20)
print('Tri-gram completion: ', Tools.format_sequence(best_sequence))
```

✓ 1.0s Python

```
Bi-gram completion :  `` That's true . *end*
Tri-gram completion:  `` That's true '' , she said . *end*
```

Context-aware spelling correction

- `beam_search_spelling_corrector(self, w_sentence, n, beam_width=2)`:
  Takes a sentence with potential noise or typos (a "noisy sentence") and apply a beam search algorithm to find the most likely corrected version. The function leverages a combination of n-gram probabilities and inverse Levenshtein distances to score candidate corrections. The sequence returned ($\widehat{t_1^k}$) is:

$$\widehat{t_1^k} = \arg\max_{t_1^k} \boldsymbol{\lambda_1} \cdot log_2 P\left(t_1^k\right) + \boldsymbol{\lambda_2} \cdot log_2 P\left(w_1^k \middle| t_1^k\right)$$

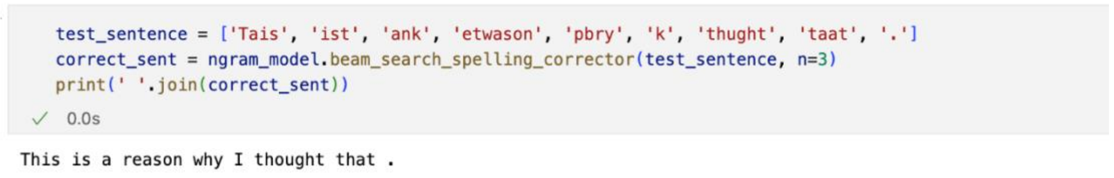
  where $P\left(w_1^k \middle| t_1^k\right) \approx \prod_{i=1}^{k} P(w_i|t_i)$ is calculated using the inverse Levenshtein distance, w_1^k is the observed sequence (“noisy sentence”) and $0 \le \boldsymbol{\lambda_1, \lambda_2} \le 1$ weight the respective probabilities.

An artificial test dataset is created to evaluate the context-aware spelling corrector. The process involves the following steps:

1. **Dataset Selection**: The same test dataset used to evaluate the language models is applied here
2. **Character Replacement**: To introduce noise into the sentences, each character is replaced with an adjacent keyboard character with a small probability. This approach simulates common typing errors, as characters that are near each other on the keyboard are more likely to be mistyped.
3. **Character Deletion**: There is also a small probability that some characters will be randomly deleted from the sentences. This simulates realistic scenarios where characters might be inadvertently omitted during typing.

The resulting dataset preserves the overall structure and context of the original sentences while introducing typographical errors that effectively challenge the spelling corrector's capabilities. Subsequently, the noisy dataset is processed and corrected using the `beam_search_spelling_corrector`.

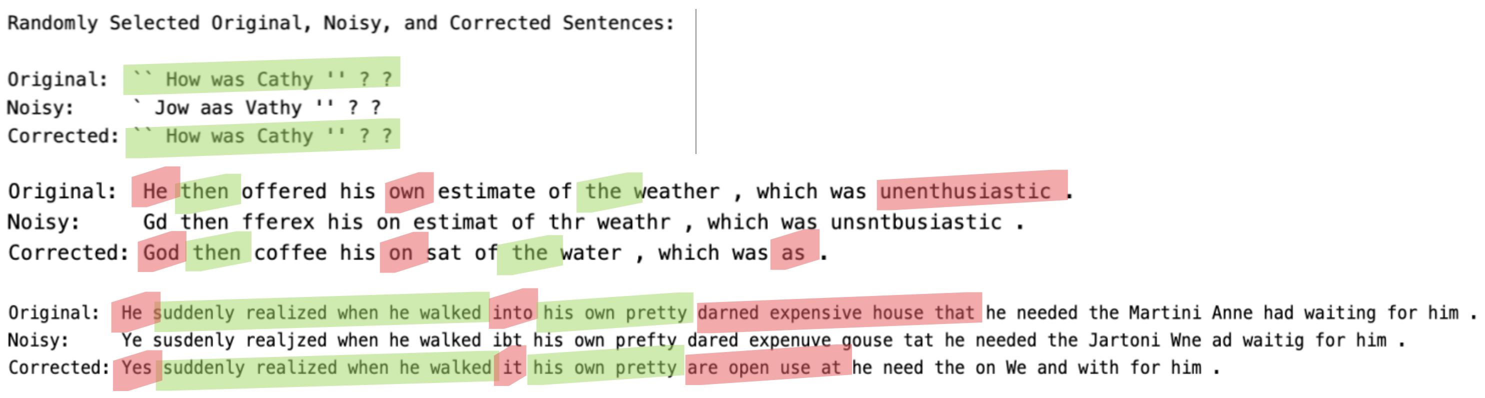
$$\text{'This is a reason why I thought that .'} \xrightarrow{\text{noise}} \text{'Tais ist ank etwason pbry k thught taat .'} \xrightarrow{\text{spelling corrector}} \dots$$



```
test_sentence = ['Tais', 'ist', 'ank', 'etwason', 'pbry', 'k', 'thught', 'taat', '.']
correct_sent = ngram_model.beam_search_spelling_corrector(test_sentence, n=3)
print(' '.join(correct_sent))
✓ 0.0s

This is a reason why I thought that .
```

Below are illustrative examples of original, noisy, and corrected sentence triplets, accompanied by indicative corrections—those marked in green represent accurate corrections, while those highlighted in red denote incorrect ones.



```
Randomly Selected Original, Noisy, and Corrected Sentences:

Original:   `` How was Cathy '' ? ?
Noisy:       ` Jow aas Vathy '' ? ?
Corrected:  `` How was Cathy '' ? ?

Original:  He then offered his own estimate of the weather , which was unenthusiastic .
Noisy:     Gd then fferex his on estimat of thr weathr , which was unsntbusiastic .
Corrected: God then coffee his on sat of the water , which was as .

Original:  He suddenly realized when he walked into his own pretty darned expensive house that he needed the Martini Anne had waiting for him .
Noisy:     Ye susdenly realjzed when he walked ibt his own prefty dared expenuve gouse tat he needed the Jartoni Wne ad waitig for him .
Corrected: Yes suddenly realized when he walked it his own pretty are open use at he need the on We and with for him .
```

The model is then assessed using two key performance metrics: Word Error Rate (WER) and Character Error Rate (CER).

Word Error Rate (WER) $= \frac{S+D+I}{N} = \frac{S+D+I}{S+D+C}$

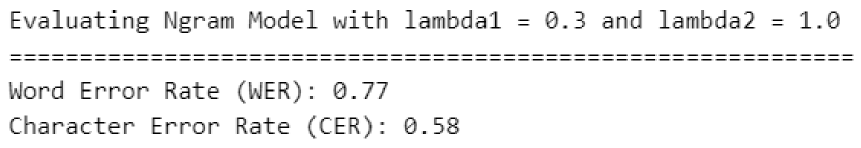
> This value indicates the average number of errors per reference word. The lower the value, the better the performance (0 is a perfect score).

Character error rate (CER) $= \frac{S+D+I}{N} = \frac{S+D+I}{S+D+C}$

> CER's output is not always a number between 0 and 1, in particular when there is a high number of insertions. The lower the value, the better the performance (0 is a perfect score).

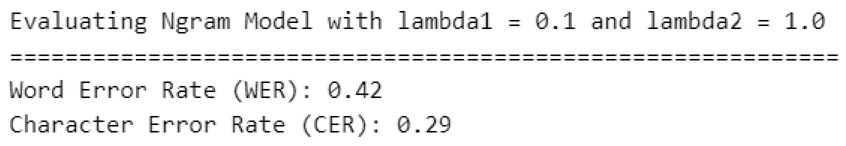
where

- S : # of substitutions
- D : # of deletions
- I : # of insertions
- C : # of correct words/characters
- N : # of words/characters in the reference $(N = S + D + C)$



```
Evaluating Ngram Model with lambda1 = 0.3 and lambda2 = 1.0
============================================================
Word Error Rate (WER): 0.77
Character Error Rate (CER): 0.58
```

Subsequently, the λ_1 and λ_2 hyperparameters are fine-tuned (`fine_tune_lambdas` method) to identify the values that optimally reduce the Word Error Rate (WER), with a secondary focus on minimizing the Character Error Rate (CER). The best performing values were λ_1=0.1 and λ_2=1.0, suggesting that, for this task, placing greater emphasis on word similarity (as measured by the inverse Levenshtein distance) is more beneficial than relying solely on the n-gram probability. Upon reevaluating the models after fine-tuning, we observe significantly improved results, as summarized below:



```
Evaluating Ngram Model with lambda1 = 0.1 and lambda2 = 1.0
============================================================
Word Error Rate (WER): 0.42
Character Error Rate (CER): 0.29
```