Athens University of Economy and Business

School of Information Sciences & Technology: Department of Informatics

Master of Science in "Data Science"

Course: Text Analytics

"ASSIGNMENT 1 – N-GRAM LANGUAGE MODELS"

Students: 1) Dionysios Voulgarakis (f3352307)

2) Rafail Mpalis (f3352308)

3) Dimitrios Stathopoulos (f3352318)

Instructor: Prof. Ion Androutsopoulos

Grader: Foivos Charalampakos

Part (I): Training of bigram and trigram language models

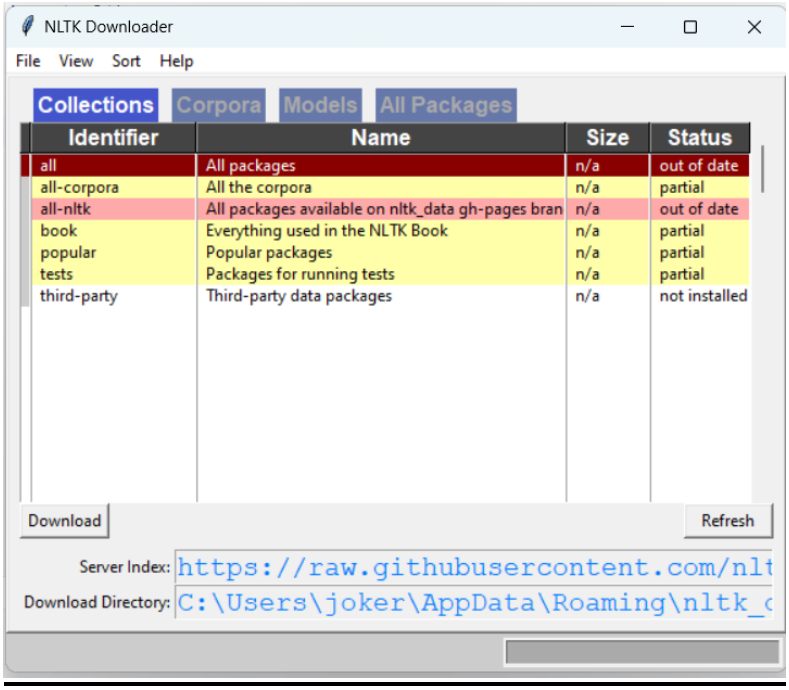
First of all, we started by downloading a corpus via **nltk** package in order to be able to train our unigram, bigram & trigram models. So, we downloaded corpus "reuters" from **nltk** package. We also, downloaded the tokenization method "punkt", so we can tokenize our sentences via the same format.

Moreover, we separated our corpus into training (70%), development (15%) and test set (15%), for the reason that we wanted to evaluate our models into a test set and if necessary to fine tune them via the development set. For the separation of our model we used the function **test_train_split** by **sklearn** package.

However, during the process of training and evaluation we spotted that these models are not very efficient with only one corpus, thus we added more corpus via nltk package. With the commands

```
import nltk
nltk.download()
```

we achieved opening the nltk downloader and choosing the corpus of our choice. We downloaded all the corpora in order to be able to choose locally the preferred corpus.



Finally, in our dataset we included the corpus:

- reuters
- brown
- alpino
- indian
- genesis
- gutenberg
- inaugural
- treebank
- product_reviews_1
- product_reviews_2

Furthermore, we preprocess our dataset as follows:

- Training set: We replaced very rare words by a special token *<UNK>* (words that occur at the most 10 times in the whole corpus).
- Development set: We replaced all out-of-vocabulary (OOV) words by a special token *<UNK>*.
- Test set: We replaced all out-of-vocabulary (OOV) words by a special token *<UNK>*.

The total size of our vocabulary words is: **20547**

Here are some example sentences of our set (train, dev, test):

Example sentence of our training set:
The|potential|purchase|of|the|interest|in|Champlin|followed|its|earlier|acquisition|of|a|part|interest|in|Southland|Corp|'|s|&|lt|;|<UNK>|>|<UNK>|Petroleum|Corp|subsidiary|.
Example sentence of our development set:
The|company|could|spend|between|250|mln|and|500|mln|dlrs|to|buy|another|non|-|copper|firm|,|<UNK>|said|,|citing|100|mln|dlrs|of|cash|and|580|mln|dlrs|of|<UNK>|bank|credit|.
Example sentence of our test set:
Robert|D|.|Hunter|,|Chase|Manhattan|area|executive|for|Europe|,|Africa|and|the|Middle|East|,|said|at|a|news|conference|that|plans|to|broaden|the|bank|'|s|activities|on|the|Italian|market|have|not|been|<UNK>|,|however|.

Build our unigram, bigram & trigram model

In order to build our models we used via **nltk** package the ngram function and Counters. We included padding for the start and the end of the sentences. So, we built a unigram, a bigram and a trigram Counter by using the training set.

The 10 most common words for each model are:

```
10 most common words for unigram counter:
[(('<UNK>',), 273451),
 ((',',), 255436),
 (('the',), 190786),
 (('.',), 177650),
 (('of',), 109922),
 (('and',), 102719),
 (('to',), 82492),
 (('in',), 61532),
 (('a',), 61322),
 ((':',), 38915)]
```

```
10 most common words for bigram counter:
[(('.', '<e>'), 141917),
 (('<UNK>', '<UNK>'), 39507),
 ((',', 'and'), 38258),
 (('<UNK>', ','), 33467),
 (('of', 'the'), 27334),
 (('<s>', '<UNK>'), 18892),
 (('<UNK>', '.'), 18367),
 (('the', '<UNK>'), 16888),
 (('in', 'the'), 16722),
 ((',', '<UNK>'), 15518)]
```

```
10 most common words for trigram counter:
[(('.', '<e>', '<e>'), 141917),
 (('<s>', '<s>', '<UNK>'), 18892),
 (('<UNK>', '.', '<e>'), 17026),
 (('<s>', '<s>', 'The'), 14643),
 (('<UNK>', '<UNK>', '<UNK>'), 12295),
 (('<s>', '<s>', '"'), 10128),
 (('?', '<e>', '<e>'), 7640),
 (('&', 'lt', ';'), 6096),
 (('said', '.', '<e>'), 6017),
 (('."', '<e>', '<e>'), 5743)]
```

Part (II): Calculation of probabilities, Cross-Entropy and Perplexity for our models

Calculation of bigram and trigram probabilities via Laplace smoothing

We built a function called **calc_ngram_proba** in order to calculate the probabilities of our bigram and trigram Counters using the Laplace smoothing technique. By calculating the probabilities for each model we also extracted the total cross-entropy and perplexity.

For Laplace smoothing technique we tested various values of alpha, thus we ended up using **alpha = 0.1**.

Cross-entropy and perplexity for bigram model:

```
The total Cross-Entropy of bigram model for our Test set is: 7.834
Perplexity of bigram model for Test Set: 228.135
```

Cross-entropy and perplexity for trigram model:

```
The total Cross-Entropy of trigram model for our Test set is:  9.749
Perplexity of trigram model for Test Set: 860.430
```

Results:

As we can observe from the results, the perplexity for trigram model is four (4) times bigger than the perplexity for bigram model. We have a huge corpus, thus we consider that Laplace smoothing technique is not very efficient and we must try a more efficient and effective technique, such as **Kneser-Ney.**

Calculation of bigram and trigram probabilities via improved Kneser-Ney smoothing

We built a function called **calc_kneser_ney_proba** in order to calculate the probabilities of our bigram and trigram Counters using the Kneser-Ney smoothing technique. This technique is more complex so we will explain it in more detail[1].

Equation of Kneser-Ney smoothing technique

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \underbrace{\boxed{\frac{\max\left(c_{KN}(w_{i-n+1}^{i}) - d, 0\right)}{c_{KN}(w_{i-n+1}^{i-1})}}}_{\texttt{firstTerm}} + \underbrace{\boxed{\lambda\left(w_{i-n+1}^{i-1}\right)}}_{\texttt{lambda}} \underbrace{\boxed{P_{KN}(w_i|w_{i-n+2}^{i-1})}}_{\texttt{Pcont}}$$

$$c_{KN}(\cdot) = \begin{cases} \mathrm{count}(\cdot) & \text{for the highest order} \\ \mathrm{continuationcount}(\cdot) & \text{for lower orders} \end{cases}$$

[1] More information for the technique:
-A simple numerical example for Kneser-Ney Smoothing [NLP] | by Denny Ceccon | Medium,
-MITx MicroMasters Program in Statistics and Data Science | Learner Testimonials (youtube.com)

For the first term the numerator is the frequency/count of the n-gram minus a discount, which is the hyperparameter delta, and for the denominator is the frequency/count of the n-gram without the final word (or we can take the ngram_minus_one counter). In our function the first term is:

$$\frac{\texttt{max(ngram\_count - delta, 0)}}{\texttt{(ngram\_minus\_one\_count + epsilon)}}$$

`epsilon = 1e-10` : used for the math division error

For term lambda the equation is:

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})} |\{w : c(w_{i-1}, w) > 0\}|$$

Which the d is the same hyperparameter delta as before, the denominator is the frequency/count of the n-gram without the final word (or we can take the ngram_minus_one counter) and the term to the right means the number (not the frequency) of different final word types succeeding the n-gram without the final word. So, is not the frequency, but we count each distinct word that follows the n-gram without the final word. In our function as lambda we have implemented alpha_weight and is as follows:

$$\frac{\texttt{(delta * prefixes\_counter[(context)] + epsilon)}}{\texttt{(ngram\_minus\_one\_count + epsilon)}}$$

`epsilon = 1e-10` : used for the math division error

For example, prefixes_counter for a bigram model is (context is the ngram without the final word):

```
# Convert list of n-grams to a list of tuples
ngram_tuples_bi = [tuple(ng) for ng in bigram_counter]

# Create a Counter for the prefixes
prefixes_counter_bi = Counter(ng[:-1] for ng in ngram_tuples_bi)
```

For Pcont term the equation is:

$$P_{\text{CONTINUATION}}(w_i) = \frac{|\{w_{i-1} : C(w_{i-1} w_i) > 0\}|}{\sum_{w'_i} |\{w'_{i-1} : c(w'_{i-1} w'_i) > 0\}|}$$

in which the numerator means the number of different string types preceding the final word, and the denominator means the number of different possible n-gram types (or we can put the length of the n-gram table). In our function Pcont is as follows:

$$\frac{\texttt{continuation\_counts[continuation\_token]}}{\texttt{len(ngram\_counter)}}$$

For example, continuation counts for a bigram model are:

```
# Calculate continuation counts
continuation_counts_bi = Counter([bigram[1] for bigram in bigram_counter])
```

Cross-entropy and perplexity for bigram model via Kneser-Ney smoothing technique:

```
100%|██████████| 37774/37774 [00:02<00:00, 13130.79it/s]
The total Cross-Entropy of bigram model via Kneser-Ney smoothing for our Test set is:  6.924
Perplexity of bigram model for Test Set: 121.427
```

Cross-entropy and perplexity for trigram model via Kneser-Ney smoothing technique:

In this case we tried to fine-tune our trigram model by testing plenty of delta hyperparameters between 0.1 – 0.9 (added each time 0.01). We ended up choosing **delta = 0.75.**

```
100%|██████████| 89/89 [05:09<00:00,  3.47s/it]
The best hyperparameter delta for Kneser-Ney smoothing technique is:  0.750
The total Cross-Entropy of trigram model for our Test set is:  6.985
Perplexity of trigram model for Test Set: 126.667
```

Results:

As we can observe the results by using Kneser-Ney smoothing technique are much better. The perplexity for trigram model using Kneser-Ney is eight times (8) better than Laplace smoothing which is very effective.

Moreover, we should note that for the Kneser-Ney function some calculations must be implemented outside the function, for the reason that for large corpus will slow up the process very inefficiently. We observed that if the calculation of lambda parameter for the numerator (prefixes_counter) is calculated for each ngram inside the function the total time for running the technique on a test corpus (not very large) will last at least five (5) hours, thus we can calculate the prefixes_counter outside the function for all the corpus and just import it as a parameter in the function. With this slight change the total time of running this technique is just seconds (3-4 secs). For more information, the different implementations of Kneser-Ney technique are in the file ***Kneser-Ney.ipynb***.

Part (III): Autocomplete an incomplete sentence

In this part of the project we are using our models in order to autocomplete a random incomplete sentence using beam search.

The algorithm starts with the initial state (the whole sentence at a given point) and starts to generate the candidate next words using function generate_candidates. For example, in the below example we gave the sentence "The report" (initial state). The first step for the algorithm is to find the candidate next words after "report". After that the algorithm assign a probability with score function to these candidates, using Kneyser-Ney smoothing and keeps the top N most probable (beam search).

Repeating this until the depth we set to arrive. In the end of the process we choose the one, most probable sequence of words. Lets see an example:

```
The 10 best words for autocomplete the sentence "The report claims" is:
The report claims of the LORD , 000 vs loss 1 , and
```

In the same way we do the trigram model. Lets see the difference in the fluency of our model.

```
The 10 best words for autocomplete the sentence "The report says that" is:
The report says that he had not been able to get a good deal of money supply rose a seasonally adjusted unemployment rate was
```

Notes:

- Firstly, if most probable next word appears to be <UNK>, which happens a lot of times, we ignore it. So there is not chance to print <UNK>.

- Secondly, If the previous word is unknown (not <UNK>) the model will not recognise it and will not assign any next words. So in that case we change the previous word to be <UNK> and predict the most probable word after <UNK>, but still it will returnthe original word, as part of the sentence (see example below).

```
The 10 best words for autocomplete the sentence "The unknown_word" is:
The unknown_word and , behold , there is no reason to believe
```

- If there is no way to find candidates based on trigram, it will use the bigram model.

Part (IV): Context-aware spelling corrector

In this part of our project we need to implement a context-aware spelling corrector.

Firstly, we need to calculate the distance between the misspelled word and the candidate words.

To achieve this we are using Demerau-Levenhtein[2] distance, which is

[2] *Damerau–Levenshtein distance between two words is the minimum number of operations (consisting of insertions, deletions or substitutions of a single character, or transposition of two adjacent characters) required to change one word into the other." -* https://www.geeksforgeeks.org/damerau-levenshtein-distance/

better than simple Levenstein distance for spelling correction, because it also takes account the transposition of two characters.

We will use almost the same structure as before with some differences. Now we use the function generate_candidates_with_distance, which returns N words of the vocab that has the lowest Demarau-Levenhstein distance. N is hyperparameter.

After having the closest N words, we now need to give the a probability using Kneser Ney smoothing and a score based on distance. We use L1, L2 as hypeparameters, in order to weight the probability and the distance respectively.

Now beam search try to find the best candidates using the model and the distance together.

Lets see an example of its use. In the first picture, the word “reprts” becomes reports. With L1=0.1, L2=0.9. As we see the most close worth with respect to demarau_levenshtein distance is the word “report”.

```
The misspelled sentece is The deprment reprts
The corrected sentence is: The department reports
```

```
[('reports', 1), ('report', 2), ('hearts', 2), ('regrets', 2), ('rests', 2), ('certs', 2), ('remote', 3), ('years', 3),
Candidate words for 'reprts' is:
reports | report | hearts | regrets | rests | certs | remote | years | reported | rupees |
```

In the second example we change L1=0.9 and L2=0.1. And the result appear to be different.

This time we gave bigger weight to our model probabilities. Thats why it preferred years instead of reports.

```
The misspelled sentece is The deprment reprts
The corrected sentence is: The department years
```

Part (V): Building a test set with falsely spelled sentences originating from the starting test set

This code defines a Python function, replace characters, that modifies a given corpus of sentences by randomly replacing characters within words based on a specified probability. The function iterates through each word in the sentence, deciding whether to replace each non-space character according to the provided probability. The replacement involves selecting a visually or acoustically similar character, as guided by the exercise. The get_similar_char function offers a basic mapping(python dictionary) of characters for this purpose.

The modify_corpus function extends this process to the entire corpus, replacing characters in each sentence, and collecting the modified sentences into a new corpus. In the example usage, the function is applied to a test corpus (test_sents) with a 10% probability of character replacement. The resulting modified corpus (modified_test_corpus) is then printed, showcasing the first 5 sentences for illustration.

Notebook output for our artificial portion of the dataset:

Figure 1:Artificially Modified Corpus (portion)

(Note: A table is located also at our notebook file!)

Part (VI): Evaluation of our custom auto-spelling corrector based on two metrics

- *Word Error Rate (WER)*
- *Character Error Rate (CER)*

1) Word Error Rate (WER) [3]

Word Error Rate is a technique widely used in the field of Automatic Speech Recognition (ASR) systems for decades. The main purpose of such a technique is to evaluate or estimate the performance of any system that operates in a

[3] More about WER at: 🤗 https://huggingface.co/spaces/evaluate-metric/wer

word- to word level. The most common systems of such types are speech recognition APIs used to power interactive voice-based technology, like Siri or the Amazon Echo.

Before explaining the terms displayed in our equation it should be mentioned that our references are the sentences from the test set, we created, and the predictions are the corrected sentences. Also, the falsely spelled sentences originate from the part (PART(V)).

In our context, we are trying to evaluate the performance of a n-gram based auto spelling corrector. The basic formula that computes the WER score is:

$$Word\ Error\ Rate = \frac{Subtitutions + Deletions + Insertions}{Number\ of\ Words}$$

But the Hugging Face implementation we are utilizing has a formula:

$$WER = \frac{S + D + I}{S + D + C}$$

Where:

S: is the number of substitutions,

D: is the number of deletions,

I: is the number of insertions,

C: is the number of correct words,

N: is the number of words in the reference (N= S+D+C)

After using the corrector to spell correct our false sentences, we store the corrected sentences to evaluate our spell-corrector. Here Levenshtein distance comes into play and the distance of the corrected words(predictions) to the original words(references) are computed.

The terms substitutions(S), insertions(I), deletions(D) are essential operations that contribute to the computation of Levenstein distances. These terms are resulting cost operations needed for each misspelled word to be transformed to the original one. Naturally the distance of a corrected word that is close (spelled correctly) to the original word is bound to be very small and vice-versa having a large Leveshstein distance (sum (S, I, D) large).

2) Character Error Rate (CER)[4]

Character error rate (CER) can be considered like WER, but it operates on a character level instead of words as we did previously. This metric also utilizes the Levenshtein distances to compute, or rather estimate the error rate regarding each character of each word.

[4] More about CER: 🤗 https://huggingface.co/spaces/evaluate-metric/cer/discussions/1

The formula for the metric CER is:

$$CharErrorRate = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}$$

With S, D, I, N, C being the same terms as we saw at the evaluation metric WER, but the difference comes down to the character level that the CER operates on.

As discussed previously the sets that are being used as references and predictions are the same sets, we utilized to estimate the WER of our spelling corrector.

Word Error Rate and Character Error Rates (both metrics applied on a portion of 50 sentences):

	Corrected Vs Original	Artificial Vs Original
Word Error Rate	**0.0106**	**0.227**
Character Error Rate	**0.029**	**0.069**

Table 1: WER and CER for Original Corrected and Artificial sets

Results:

Upon analyzing the scores, it is evident that the Word Error Rate (WER) for corrected words significantly outperforms the artificial WER score. This suggests that our spell corrector effectively rectifies approximately 15% (14.7% exactly) of the total misspelled words. As for the Character Error Rate (CER) metric, drawing conclusions is challenging due to the initially low error rate. This outcome is expected, given that we introduce artificial errors by changing each character with only a 10% probability during the creation of the artificial set. Consequently, the initial error rate was not excessive.

Participation in the Assignment

For this assignment each member of the team contributed equally. In more detail:

- Rafail Mpalis: parts (i) & (ii)
- Dionysios Voulgarakis: parts (iii) & (iv)
- Dimitrios Stathopoulos: parts (v) & (vi)

More specifically, for the implementation of Kneser-Ney smoothing technique each member of the team participated equally and the final result was a team effort.

Link for Google Colab notebook:

nGram_models_final_V2.ipynb - Colaboratory (google.com)

Link for Kneser-Ney implementations:

Kneser-Ney.ipynb - Colaboratory (google.com)