Introduction

The aim of this assignment is to develop a bigram and a trigram language model in order to estimate the probabilities of sentences. In particular, the goal is to assign higher probabilities to sentences that 'make sense', and a lower probability to sentences with, e.g., shuffled words. The trigram model that was developed estimates probabilities using Add-a smoothing. The bigram model has two available smoothers: Add-a and the Interpolated Kneser-Ney smoother. The models that were created were evaluated using the Cross-Entropy and Perplexity criteria. The Interpolated Kneser-Ney bigram model had the best scores, followed by the Add-a bigram model and then the Add-a trigram model. Lastly, a context-aware spelling corrector was developed, where the probability estimated by the bigram model and the distance of the sentence's words from the model's vocabulary words were weighted and combined, in an effort to correct false sentences.

Corpus and Preprocessing

The corpus that was used to develop the models consists of 29,059 sentences (27,652 unique words out of the total 766,811 words) from the Australian Broadcasting Commission 2006[1], provided by NLTK[2]. The content of the sentences is about rural and science news.

Each of these sentences was cleaned and padded before being fed to the models using the static method *sent_preprocessing()*. For the cleaning part, all the punctuation was removed, and all the words were lowercased. Then, each sentence was padded according to the model needs. In particular, for the bigram model, each sentence begins with the special token **start** and ends with the special token **end*,* whereas for the trigram model, each sentence begins with two special tokens, **start1** and **start2** and ends with the special token **end*.*

Models

A class named LM (from language model) was created. In order to create an instance of a bigram model, the *n_type* parameter must be set equal to *'bigram'*. If *n_type* is set equal to '*trigram*' then an instance of a trigram model is created. After the instantiation of the chosen model, two phases follow: the training phase and the probability estimation phase.

[1] https://www.abc.net.au/
[2] https://www.nltk.org/

Training Phase

In order to train a model, the method *train()* must be called, passing a list of 'raw' sentences as an argument. All the cleaning and padding is handled by the model. After the *train()* method is called, three dictionaries containing the unigram, bigram and trigram[3] counts, and two dictionaries containing the number of unique words that follow and precede each vocabulary word (these two will be used to implement the Interpolated Kneser-Ney smoother) are created. These dictionaries are stored as attributes which can be called any time after the training is complete. Furthermore, any word that has a count lower than 10 was replaced by the special token **UNK**.

Probability Estimation Phase

Once a model is trained, one can start estimating the probabilities of n-grams and sentences. By calling the *estimate_sent_prob()* method, a list of the log-transformed probabilities of the sentences that were passed as an argument is returned. The log-transformation was used in order to prevent underflow (as well as loss of precision) when multiplying the individual probabilities of each n-gram.

If the model that was created is a bigram model, one can choose between Add-a and Interpolated Kneser-Ney smoothing by setting the parameter *smoothing* equal to '*add_a*' or '*kn*'. The trigram model can only use Add-a smoothing.

The Add-a smoothing is calculated as follows:

$$P_{Laplace}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + \alpha}{c(w_{k-2}, w_{k-1}) + \alpha \cdot |V|}$$

where the numerator consists of the counts of the higher-order n-gram plus a constant value *a*, and the denominator consists of the counts of the lower-order n-gram plus the length of the training vocabulary multiplied by the constant *a*. The constant *a*, which moves probability mass from seen to unseen events, can take a value from 0 to 1 and can be tuned in order to achieve the best evaluation scores (more on that later).

[3] The trigram-counts dictionary is created only when an instance of a trigram model is created.

The interpolated Kneser-Ney smoothing is calculated as follows:

$$P_{\text{KN}}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1}w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{\text{CONTINUATION}}(w_i)$$

The first term is the discounted bigram probability. The constant *d*, just like *a*, moves probability mass from seen to unseen events. It is set equal to 0.5 for bigrams with a count of 1, and 0.75 for the rest, as suggested by Jurafsky & Martin (2001)[4]. The λ term consists of the normalized discount multiplied by the total number of vocabulary words that follow w_{i-1} and is calculated as follows:

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)} |\{w : C(w_{i-1}w) > 0\}|$$

Finally, the $P_{continuation}(w_i)$ term, is the probability of seeing the word w_i as a novel continuation in a new unseen context. It is calculated as the number of vocabulary words that precede w_i divided by the total number of bigrams:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u', w') : C(u'w') > 0\}|}$$

Concluding, the *estimate_sent_prob()* method, will clean and pad the given sentences according to the model type, then it will transform each sentence to bigrams (or trigrams if a trigram model is created) and then it will sum the log-transformed probabilities of each n-gram (estimated by the method *estimate_ngram_prob()* or *kn_prob()*, depending on the chosen smoother) and return the total log-transformed probability of each sentence.

Model Evaluation

The two criteria that were used to evaluate the models' performance are: *cross-entropy* and *perplexity*.

The total sentences of the corpus were split into three different sets: the *training set*, the *development set* and the *test set.* The *training set* consists of 70% percent of the sentences, the *development set* consists of 10% of the sentences, and the remaining 20% was assigned to the *test set*.

[4] https://web.stanford.edu/~jurafsky/slp3/

A trigram and a bigram model were trained using the training set. Using the *development set*, the hyperparameter *a* (of the Add-a smoother) was tuned and lastly, the models were evaluated using the test set. The results that occurred are the following:

- Bigram model using Add-a smoothing, with a=0.01:
  - Cross-Entropy: 6.89
  - Perplexity: 118.85
- Bigram model using Interpolated Kneser-Ney smoothing:
  - Cross-Entropy: 6.52
  - Perplexity: 92.36
- Trigram model using Add-a smoothing, with a=0.007:
  - Cross-Entropy: 8.72
  - Perplexity: 424.00

It seems that the Bigram model using the Interpolated Kneser-Ney smoothing scored the lowest cross-entropy and perplexity!

However, these scores stand-alone cannot tell us whether the models are working or they just assign arbitrary probabilities to each sentence. Thus, the words of each individual sentence of the test set were shuffled, and the scores were recalculated:

- Bigram model using Add-a smoothing, with a=0.01:
  - Cross-Entropy: 9.55
  - Perplexity: 753.23
- Bigram model using Interpolated Kneser-Ney smoothing:
  - Cross-Entropy: 8.38
  - Perplexity: 334.72
- Trigram model using Add-a smoothing, with a=0.007:
  - Cross-Entropy: 10.89
  - Perplexity: 1909.98

The scores are significantly larger, which means that the models assign lower probabilities to ‘non-sense’ sentences. Thus, it can be safely inferred that all the models are working!

Lastly, using the language models and a word of choice, one can also find the word’s most probable continuation. For example, using the Interpolated Kneser-Ney model and the words ‘good’, ‘he’ and ‘make’:

good	Probability
news	0.097393
for	0.032351
at	0.027960
to	0.027091
and	0.024719
as	0.017084
prices	0.015524
enough	0.015372
thing	0.015266
on	0.013823

he	Probability
said	0.393450
says	0.334239
is	0.042016
has	0.019870
was	0.014306
will	0.012981
s	0.012233
had	0.008396
and	0.007996
also	0.007358

make	Probability
a	0.126875
the	0.121068
it	0.114708
sure	0.077510
up	0.046265
them	0.030279
sense	0.016443
any	0.014325
an	0.012470
people	0.012261

A feature like this could be proven really useful for the development of a context-aware spelling corrector!

BEAM SEARCH

The beam search algorithm is used as a spelling corrector while being context aware. This means that not only is it able to identify and correct spelling mistakes, but it can also produce a sentence that makes sense in case the one given does not. In order to achieve that the `close_words` method is called first. This method, given a sentence, finds the N closest words in terms of spelling, where N: [1, length of vocabulary]. The extraction of vocabulary words is mentioned previously. Given a sentence the 3 closest words for every token according to this method are:

```
Given sentece :  this is a test
this  :  [('this', 1.0), ('his', 0.8571428571428571), ('things', 0.8)]
is  :  [('is', 1.0), ('its', 0.8), ('his', 0.8)]
a  :  [('a', 1.0), ('at', 0.6666666666666666), ('as', 0.6666666666666666)]
test  :  [('test', 1.0), ('tests', 0.8888888888888888), ('latest', 0.8)]
```

The number next to the words corresponds to the matching ratio with the initial word. The closer the words are to each other the higher the number (max = 1).

Regarding the distance between words while the Levenstein distance was trialed a different method was used that yielded better results. The method is called gestalt pattern matching.

Given a sequence a ration of similarity is returned, higher meaning closer. As it is a ratio it maintains probabilistic properties (0<=p<=1) and thus was determined to fit better in the algorithm. The results seem to agree. For completion reason the Levenstein distance was included but not used.

The implementation of the beam search itself, combines knowledge gained from the corresponding lecture and some recursive principles. More specifically, the main idea revolves around the partition of the given sentence. This means, that as the beams search tries to find the optimal path for a sentence, the recursion logic states that the best path is the optimal one up until the second from last word, plus the best path to the last word. This applies to every 'sub-sentence' up until that point. Maintaining that logic every word is tokenized until there is only the start token and the first word left. This is achieved by the beam search method calling itself for a part of the initial sentence. Having the start token and the first word the `prob_calc` method is called to calculate the best possible combinations. For every combination of the start token and the closest words a score is given according to the next formula:

$$\hat{t}_1^k = \underset{t_1^k}{\operatorname{argmax}}\, \lambda_1 \log P\left(t_1^k\right) + \lambda_2 \log P(w_1^k | t_1^k)$$

The first part of the formula is calculated by using the already implemented bigram calculator while the second one takes into account the closest words to the given sentence.

The 2 combinations with the highest score are kept and the result is returned to the last recursion call which now has to find the best combination of the two previous words with a new one using the same methods as before. The method ends when a combination is returned using all the words from the sentence. The last word is the *end* token so the last sentence to be returned (first recursion) is the same in both cases. A sample execution of the method is presented below.

```
Given sentece :  thsi i as tets

['*start2*', 'thsi', 'i', 'as', 'tets', '*end*']
['*start2*', 'thsi', 'i', 'as', 'tets']
['*start2*', 'thsi', 'i', 'as']
['*start2*', 'thsi', 'i']
['*start2*', 'thsi']
fin
['*start2* this', '*start2* the']
['*start2* this is', '*start2* this i']
['*start2* this is as', '*start2* this is a']
['*start2* this is as the', '*start2* this is a test']
['*start2* this is a test *', '*start2* this is a test *']

Did you mean :   this is a test
```

We can see that given a sentence the tokens are split up until the last two, followed by the fin token. This means that each word was split and the last recursion was executed. Starting with

the last two words the algorithm tries to find the best combination of the bigrams starting with *start* token as described. The two highest rated bigrams are *start* this, *start* the. The last recursion having these results will try to find the best combination between these two bigrams and a new word and so on until all tokens have been examined.

In order to test the results as well as tune the 3 hyperparameters involved, a set of sentences was used. These sentences stem from the development set with a twist. A random letter was excluded or changed from every word. This is shown in the end result. The tree hyperparameter regard the number of the closest to the original words to include, and the values of λ1, λ2 from the formula above. The number of words was selected to be the size of the vocabulary which made the algorithm more accurate but also slower depending on the training set. For the λ1, λ2 the best combination was determined to be the 0.1 – 0.9 combination which gives more weight to the spelling of the word rather that its context. Nevertheless, context is present and does play an important role as proven by a following test
For the presentation of the method's capabilities the sentences used stem from the test set after the previous changes. This is the case with the following examples as well.

For example, one of the sentences used is the following:

```
Correct sentence :  *start1* *start2* we can apply this method to many other mammals it is very simple *end*
Wrong sentence :  start2* w cn appl his metod t may ther mammal tt s vry imple *ed*

Given sentece :  w cn appl his metod t may ther mammal tt s vry imple

['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal', 'tt', 's', 'vry', 'imple', '*ed*']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal', 'tt', 's', 'vry', 'imple']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal', 'tt', 's', 'vry']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal', 'tt', 's']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal', 'tt']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther', 'mammal']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'ther']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't']
['start2*', 'w', 'cn', 'appl', 'his', 'metod']
['start2*', 'w', 'cn', 'appl', 'his']
['start2*', 'w', 'cn', 'appl']
['start2*', 'w', 'cn']
['start2*', 'w']
fin
['start2* we', 'start2* wa']
['start2* we can', 'start2* wa can']
['start2* we can apply', 'start2* we can apple']
['start2* we can apply his', 'start2* we can apply this']
['start2* we can apply this method', 'start2* we can apply his method']
['start2* we can apply this method to', 'start2* we can apply his method to']
['start2* we can apply this method to many', 'start2* we can apply his method to many']
['start2* we can apply this method to many other', 'start2* we can apply his method to many other']
['start2* we can apply this method to many other mammals', 'start2* we can apply his method to many other mammals']
['start2* we can apply this method to many other mammals that', 'start2* we can apply his method to many other mammals that']
['start2* we can apply this method to many other mammals that is', 'start2* we can apply his method to many other mammals that is']
['start2* we can apply this method to many other mammals that is very', 'start2* we can apply his method to many other mammals that is very']
['start2* we can apply this method to many other mammals that is very simple', 'start2* we can apply his method to many other mammals that is very simple']
['start2* we can apply this method to many other mammals that is very simple *', 'start2* we can apply this method to many other mammals that is very simple *']

Did you mean :  we can apply this method to many other mammals that is very simple
```

From this example we can see the behavior of the method in a real sentence. With the exception of the word that the sentence is the same with the actual correct one. Even with the addition of ‘that’ instead of ‘it’, grammatically and syntactically the sentence is correct. A noteworthy result of the algorithm is the choice of the word mammals instead of mammal which is obviously closer to the given word. This is attributed to being context aware meaning that the algorithm decided that the word “other” preceding a noun means that the noun is plural. To prove its behavior the same sentence was used but the word “other” was substituted with the word “a” which clearly implies singularity of the noun. This is the result:

```
Correct sentence :  *start1* *start2* we can apply this method to many a mammals it is very simple *end*
Wrong sentence :  start2* w cn appl his metod t may a mammal tt s vry imple *ed*

Given sentece :  w cn appl his metod t may a mammal tt s vry imple

['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal', 'tt', 's', 'vry', 'imple', '*ed*']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal', 'tt', 's', 'vry', 'imple']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal', 'tt', 's', 'vry']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal', 'tt', 's']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal', 'tt']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a', 'mammal']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may', 'a']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't', 'may']
['start2*', 'w', 'cn', 'appl', 'his', 'metod', 't']
['start2*', 'w', 'cn', 'appl', 'his', 'metod']
['start2*', 'w', 'cn', 'appl', 'his']
['start2*', 'w', 'cn', 'appl']
['start2*', 'w', 'cn']
['start2*', 'w']
fin
['start2* we', 'start2* wa']
['start2* we can', 'start2* wa can']
['start2* we can apply', 'start2* we can apple']
['start2* we can apply his', 'start2* we can apply this']
['start2* we can apply this method', 'start2* we can apply his method']
['start2* we can apply this method to', 'start2* we can apply his method to']
['start2* we can apply this method to many', 'start2* we can apply his method to many']
['start2* we can apply this method to many a', 'start2* we can apply his method to many a']
['start2* we can apply this method to many a mammal', 'start2* we can apply his method to many a mammal']
['start2* we can apply this method to many a mammal that', 'start2* we can apply his method to many a mammal that']
['start2* we can apply this method to many a mammal that is', 'start2* we can apply his method to many a mammal that is']
['start2* we can apply this method to many a mammal that is very', 'start2* we can apply his method to many a mammal that is very']
['start2* we can apply this method to many a mammal that is very simple', 'start2* we can apply his method to many a mammal that is very simple']
['start2* we can apply this method to many a mammal that is very simple *', 'start2* we can apply this method to many a mammal that is very simple *']

Did you mean :  we can apply this method to many a mammal that is very simple
```

In this case the word “mammal” is kept as it is.

By no means is this a perfect algorithm. Depending on the context of the sentence and the mistakes made, the end result may vary. For instance, if the sentence from the lectures is given, which is clearly out of context regarding the training corpus the result is the following:

```
Given sentece :  he pls gd ftball

['*start2*', 'he', 'pls', 'gd', 'ftball', '*end*']
['*start2*', 'he', 'pls', 'gd', 'ftball']
['*start2*', 'he', 'pls', 'gd']
['*start2*', 'he', 'pls']
['*start2*', 'he']
fin
['*start2* he', '*start2* the']
['*start2* he plans', '*start2* he plus']
['*start2* he plans good', '*start2* he plans gold']
['*start2* he plans good falls', '*start2* he plans gold ball']
['*start2* he plans good falls *', '*start2* he plans good falls *']

Did you mean :   he plans good falls
```

With a corpus revolving around sports, the result would have been far more accurate.

Conclusions

All in all, it seems that both the language models and the context-aware spelling corrector are working as intended. Out of all the language models, the bigram one using the Interpolated Kneser-Ney smoothing performed the best, that is, it produced the lowest Cross-Entropy and Perplexity scores. As for the context-aware spelling corrector, the results are satisfactory when the sentence that is being corrected has a similar theme to the training corpus.

Some possible improvements could be the development of the Interpolated Kneser-Ney smoothing for the trigram model, and its use on the beam search algorithm. Furthermore, the performance of both the spelling corrector and the language models could be improved if a larger dataset containing various themes was used. In this way, the corpus' vocabulary would be bigger, the probabilities estimated by the language would be more accurate and, thus, the spelling corrector would have produced better corrections.

Code:
https://colab.research.google.com/drive/1DSpeO89y_gVqCBHKCMJJ3sN-uURKrWvy?usp=sharing