# BUILDING A LANGUAGE MODEL
# ICS 2203

Ryan Camilleri

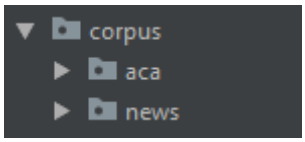
ID: 328400L  Bsc IT A.I. 2nd Year

# Contents

# 1. Pre-processing

## 1.1 Building the lexicon

The corpus selected for this assignment is the British National Corpus, using the *2553.zip* folder attached. This folder contained four other directories, all of which had multiple xml files denoting several different texts. Instead of choosing all directories for building the lexicon, it was decided that only two were used, so that pre-processing is done faster. Note that, should the user wish to append more data, the program is designed in a way as to dynamically go through each folder within the corpus directory, so that the lexicon is generated using both the previous and newly appended corpora. The two directories chosen were the aca and news folders. These folders have a combined size of 89.2 MB storing a combined corpus of a total of 89,529 sentences.



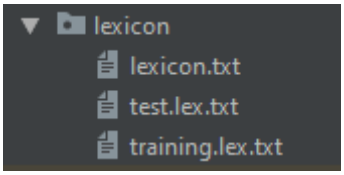

Using the Beautiful Soup library [1] along with the Python programming language, the above mentioned directories were parsed. Beautiful Soup was used for reading the xml tags within each file stored in the directories such that sentences could be easily found, processed, and included in the lexicon text file. Processing of the sentences using the Regex library [2], included that of removing punctuation, converting every character to lower case and making sure that words have a single whitespace in between them. Regarding numbers, these weren't removed since when generating text, the generated sequence made more sense when numbers were present. The parsed sentence is then appended with a start token before it's first word and an end token after it's last word. These tokens are used when generating the models and denote the start of a sentence by <s> (start token) and the end of a sentence by </s> (end token). Thus, by doing this for every sentence in the corpus, such sentences were written to the lexicon text file which stores each sentence in a new line.

## 1.2 Training set

Keeping in mind that with greater data, models have more history to use when generating text and finding probabilities, an 80:20 split was chosen to split the lexicon accordingly. This meant that, sentences within the lexicon are given an 80% chance to be included within the training set. This set will be used for building the separate language models and would hence require a large number of sentences when compared to the test set.

## 1.3 Test set

The test set would thus have all those sentences which weren't included in the training set. This set is used to find probabilities of the sentences using a specific language model. The values returned are then used to calculate the percentage of sentences in the test set that get a zero probability; Denoting how efficient some models are, when compared to others.



# 2. General Implementation

In order to enforce efficient and organised coding practices, an object oriented approach was taken to implement the language models. This meant that each unique model has its own class so that it may be implemented differently depending on the version it is being built for. Taking the example of a laplace bigram model, this model would have to be generated by finding the bigram class through the laplace python script. Each class model was also designed to have its own text generation and probability calculation functions, which make use of the model that they are used in, to output type-specific results. The below are some of the main implementations that are critical throughout the running of the program.

## 2.1 Enumerations

Two enumerator classes were built to store the different versions and types of the models. One enumerator denotes the type of model which includes four options, Unigram, Bigram, Trigram or Interpolation. The other denotes the version of the model, whether it is Vanilla, Laplace or UNK. Combined, the values of these two enumerators define each model able to be generated within this project.

## 2.2 Initial models

By applying an object-oriented approach as mentioned above, this meant that each model had to access the training set through file operations and go through every sentence to generate the model itself. This proved to be very inefficient and time consuming to say the least. However, this was avoided by building the initial default models (Unigram, Bigram, Trigram) once, which solely contain the counts of the words/sequences found within the training set. These model counts are then passed through each model generating function, so that instead of generating the model from the ground up, type specific models could now copy the counts and manipulate them according to their type. This not only reduces runtime to build models, but it also eliminates a lot of repeated code and unnecessary loops for model generation.

## 2.3 Default Dictionaries

A decision needed to be taken on how models were to be stored. The dictionary data structure was chosen instead of a matrix since matrices waste a lot of memory and have higher chances of producing memory leaks when dealing with large amounts of data. Thus, each model is represented by a default dictionary of probabilities. Default dictionaries are data structures used when implementing the collections library [3]. These work exactly like normal dictionaries, but instead, are initialized with a function that takes no arguments and provides the default value for a non-existent key. A default dictionary will hence never raise a Key Error when a key doesn't exist and instead return a count of 0 (non-existent within the training data). This proved to be very beneficial when calculating probabilities since, instead of creating checks to determine whether keys exist within the dictionary or not, it automatically outputs a probability of 0, signifying that the word or sequence doesn't occur in the trained model.

## 2.4 Start and End tokens

The addition of the start and end tokens was mostly done for the bigram and trigram models. Since unigram models calculate probabilities on words in isolation, it doesn't matter to the model whether a specific word is usually used in the beginning of a sentence or at the end. On the other hand, when looking at how bigrams and trigrams calculate probabilities, these tokens are necessary. The following example denotes the importance of such tokens.

Given sequence: “This is a test.”

Pre-processed sequence: “this is a test”

Token appended sequence: “<s> this is a test </s>”

**Unigram Models**

P( sequence ) =

P( <s> ) *
P( this ) *
P( is ) *
P ( a ) *
P ( test) *
P ( </s> )

**Bigram Models**

P( sequence ) =

P( this | <s> ) *
P( is | this ) *
P ( a | is ) *
P ( test | a) *
P ( </s> | test )

**Trigram Models**

P( sequence ) =

P( is | <s>, this) *
P ( a | this, is) *
P ( test | is, a) *
P ( </s> | a, test )

As shown above, a greater n-gram would need to account for more history before a given word. In the case that such tokens weren’t implemented, two problems would occur. Not only would terms like ‘this’ (bigram) and ‘this is’ (trigram) be disregarded as starting terms but also, an error would occur when generating text in trigrams. Given only a single word as input to the text generation function, since the trigram requires a history of two words to generate the next word, this wouldn’t be possible without the start token. Hence, when the token is appended to the inputted word, the trigram would then have two terms to use when generating the next word.

## 2.4 Text Generation

As described, each model has its own text generation function. These functions are all different in some form or another since they are accustomed to the model, they are used in. However, these all follow a general implementation. Each function is tasked to append the given sequence (1 or more words) with the start token <s>. After this is done, the program generates a random number between 0 and 1 and an accumulator, initially set to zero, is incremented the probability values of the words being considered. Once the accumulator reaches a number greater than or equal to the randomly generated number, the last word that satisfied this condition is generated and appended to the text. The algorithm continues this process of randomly selecting words until the end token </s> is generated, signifying an end of sentence. Sentences can be as long as a hundred words to as short as single word. This implementation was done like so since when testing with a word limit, the last word generated didn’t always imply an end of sentence. Hence, by generating text this way, the sentence would most likely have an end-word which makes sense. As a precaution, given no word is generated by the model due to an error or an unknown word in the sequence passed, the program warns the user of an invalid input that he may have done and instructs him to try again.

## 2.5 Finding Probability

Similar to text generation functions, each model has its own probability calculating function which varies in implementation within each model. The general implementation however first checks whether the sequence given is from the test set or otherwise. If the sequence is not from the test set, the start <s> and end </s> tokens are appended to the sequence so that probabilities can be calculated, using the tokens for reference of starting words and end words. The probability of each sequence/word within the given sentence is multiplied to a probability variable initially set to 1. Thus, once all terms have been covered, the variable would store the product of all probabilities, which is the probability that the given sentence is generated by the model. In the case that the sentence given is provided from the test set, the sentence would already have the tokens appended to it and so could instantly begin probability calculation.

# 3. Vanilla Version

## 3.1 Unigram Model

The unigram model is generated by copying the unigram counts passed to it which takes the form of a dictionary whereby keys are terms in the training set and the values are the number of times the term appears in the set. These values are then divided by the total number of words in the model. By doing this, each word in the model has a value which refers to the probability of it occurring within a sentence.

### *3.1.1 Text Generation*

The text generation function for vanilla unigram follows closely the simplified implementation described in section 2.4. The below is an example of generated text by using the model.

```
---------------------- Vanilla-Unigram Generated Text ----------------------
Sequence Given : the
Generated Text : the last form that a sold hard a did terms.
----------------------------------------------------------------------------
```

### *3.1.2 Probability Calculation*

The probability calculation function for vanilla unigram follows closely the simplified implementation described in section 2.5. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Vanilla-Unigram Sequence Probability --------------------
Sequence : this is a test
Probability : 4.2154800067067996e-13
----------------------------------------------------------------------------
```

## 3.2 Bigram Model

The bigram model is generated by copying the bigram counts passed to it which takes the form of a 2D dictionary whereby one term (key) has a dictionary of other terms as its value. These nested terms all have a value equal to their count that they occur after the primary key-term. The value for such terms is then divided by the total number of words that the primary term has, following it. By doing this, each word in the nested dictionary would have a value which refers to the probability of it occurring within a sentence after the primary term.

### *3.2.1 Text Generation*

The text generation function for vanilla bigram follows closely the simplified implementation described in section 2.4. The below is an example of generated text by using the model.

```
---------------------- Vanilla-Bigram Generated Text ----------------------
Sequence Given : the
Generated Text : the abysmal european angling times are growing around but the light.
---------------------------------------------------------------------------
```

### *3.2.2 Probability Calculation*

The probability calculation function for vanilla bigram follows closely the simplified implementation described in section 2.5. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Vanilla-Bigram Sequence Probability --------------------
Sequence : this is a test
Probability : 1.6236291599429673e-08
----------------------------------------------------------------------------
```

## 3.3 Trigram Model

The trigram model is generated by copying the trigram counts passed to it which takes the form of a 2D dictionary whereby one tuple (key with 2 terms) has a dictionary of other terms as its value. These nested terms all have a value equal to their count that they occur after the primary tuple. The value for such terms is then divided by the total number of words that the tuple has, following it. By doing this, each word in the nested dictionary would have a value which refers to the probability of it occurring within a sentence after the primary tuple.

### *3.3.1 Text Generation*

The text generation function for vanilla trigram follows closely the simplified implementation described in section 2.4. The below is an example of generated text by using the model.

```
----------------------- Vanilla-Trigram Generated Text -----------------------
Sequence Given : the
Generated Text : the return is actually quieter than a problem or issue.
------------------------------------------------------------------------------
```

### *3.3.2 Probability Calculation*

The probability calculation function for vanilla trigram follows closely the simplified implementation described in section 2.5. The below is an example of the probability of a specific sequence to be generated by using the model. As can be seen, a probability of 0 is produced which denotes that the sequence given cannot be generated using the vanilla trigram model.

```
------------------- Vanilla-Trigram Sequence Probability -------------------
Sequence : this is a test
Probability : 0.0
----------------------------------------------------------------------------
```

# 4. Laplace Version

## 4.1 Unigram Model

The unigram model in this case is implemented similar to the vanilla unigram model (section 3.1) but instead of dividing with total word count, the laplace equation is applied to each word value, such that 1 is incremented to every word count and this value is then divided by the total word count added with the number of distinct terms in the vocabulary.

### *4.1.1 Text Generation*

The text generation function for laplace unigram follows closely the simplified implementation described in section 2.4. The below is an example of generated text by using the model.

```
---------------------- Laplace-Unigram Generated Text ----------------------
Sequence Given : the
Generated Text : the at randomize etc going argues d change eye.
----------------------------------------------------------------------------
```

### *4.1.2 Probability Calculation*

The probability calculation function for vanilla unigram follows closely the simplified implementation described in section 2.5. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Laplace-Unigram Sequence Probability --------------------
Sequence : this is a test
Probability : 3.43836291583853e-13
----------------------------------------------------------------------------
```

## 4.2 Bigram Model

Similar to the laplace unigram model, the same laplace equation has to be used in this model with some alterations to account for the fact that bigrams do not consider words in isolation. However, since bigrams are generated using a 2D dictionary, the terms stored in the nested dictionary are those terms which occur in the training set only. This means that terms with a count of zero (those which don't occur in the training data) aren't visible within the nested dictionary. The problem with laplace is that such non-occurring words are needed to be given a chance to be generated. Thus, instead of adding all terms in the dataset which weren't in the nested dictionary of a specific term, the terms are ignored, and a default value is found for such terms when needed in the text generation and probability functions. The 'default' laplace value is the value found by applying the following equation:

$$Laplace\ value\ for\ terms\ with\ a\ count\ of\ 0 = \frac{1}{previous\_term\_count\ +\ num\_distinct\_terms}$$

instead of:

$$Laplace\ value\ for\ terms\ = \frac{1\ +\ bigram\_count}{previous\_term\_count\ +\ num\_distinct\_terms}$$

This greatly reduces the need for large amounts of memory use and eliminates the time needed to build such a large dictionary which would technically represent a 2D matrix in itself; something which shouldn't be used in such implementations. Terms which were already in the nested dictionary are applied to the laplace function, such that their new value represents their laplace value equivalent.

### *4.2.1 Text Generation*

When generating text, all possible words in the unigram model are considered for generation. Each word is checked for whether it exists within the bigram model or not. If the term doesn't exist, the accumulator is appended the default laplace values for the term. Otherwise, the bigram model value is appended. This process is repeated until the accumulator is greater than or equal to the randomly generated number. Once this condition is satisfied, the word which last satisfied the condition is appended to the generated text and the algorithm continues to generate the next word.

It was noted that in this case, a word limit had to be induced due to the Laplacian model in itself. Since each and every word is being given a chance to be generated, the chances for the end token to be chosen is greatly reduced. Thus, the model was designed to generate up to 10 words in total, given that the end token hasn't been generated before the word limit is reached. The below is an example of generated text by using the model.

```
---------------------- Laplace-Bigram Generated Text ----------------------
Sequence Given : the
Generated Text : the experience erratic meon collage standstill enticed twofold anybody phoney geraldine.
---------------------------------------------------------------------------
```

### *4.2.2 Probability Calculation*

When finding the probability of a sequence, terms in it are checked for whether they have a bigram count of 0 and if they exist in the dataset. If this condition is satisfied, the default laplace value is found and inserted within the equation; Otherwise, the value from the bigram model is used. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Laplace-Bigram Sequence Probability --------------------
Sequence : this is a test
Probability : 5.999094509954131e-13
----------------------------------------------------------------------------
```

## 4.3 Trigram Model

The laplace trigram model is built similar to the bigram model since both models deal with 2D dictionaries. The only difference lies with the fact that trigrams deal with tuples when considering previous terms, such that the new laplace equation would be:

$$Laplace\ value\ for\ terms\ = \frac{1\ +\ trigram\_count}{previous\_tuple\_count\ +\ num\_distinct\_terms}$$

Laplace default values for terms which do not occur after a specific tuple are found by:

$$Laplace\ value\ for\ terms\ with\ count\ 0\ = \frac{1}{previous\_tuple\_count\ +\ num\_distinct\_terms}$$

### *4.3.1 Text Generation*

The text generation function for laplace trigram follows closely the implementation described in the previous section 4.2.1. The below is an example of generated text by using the model.

```
---------------------- Laplace-Trigram Generated Text ----------------------
Sequence Given : the
Generated Text : the atlantis crossing hallowed avilla inferred £144 threenight haj print geordie.
----------------------------------------------------------------------------
```

### *4.3.2 Probability Calculation*

The probability calculation function for laplace trigram follows closely the implementation described in section 4.2.2. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Laplace-Trigram Sequence Probability --------------------
Sequence : this is a test
Probability : 1.7252396462757242e-14
-----------------------------------------------------------------------------
```

# 5. UNK Version

## 5.1 Unigram Model

To generate the UNK unigram model, unlike other models, the training lexicon has to be used once again. Generating the model could have been done in two ways. One possible solution would have been to generate the model from the already generated unigram counts as described in section 2.2. This however proved to be very inefficient when trying to convert words with a count of one, to the UNK token. This is because, in order to do so, a copy of the unigram counts had to be generated and once a word with a count of 1 is found, this word is popped from the original model and the UNK token value within the model is incremented by 1. The alternate implementation didn't require for the initial counts model to be copied. Instead, it uses this model to build another model from the lexicon. This was done, by checking the count of each word from the lexicon by applying the unigram counts model. If the total count of the word is 1, the new model's UNK token value is immediately incremented by 1 without the need for a pop procedure, thus saving some time in the end. Hence, this implementation was chosen over the other. In the end, once all counts have been generated, these are divided by the total count of terms to generate the probability values for each term.

### *5.1.1 Text Generation*

The text generation function for UNK unigram follows closely the simplified implementation described in section 2.4. The only difference is that each word within the given sequence is checked for whether it occurs in the model or not. If it occurs and its count is 1, the word is replaced by the UNK token. If the word doesn't occur (count of 0), this is also replaced by the token. Otherwise, the word is left untouched. The below is an example of generated text by using the model.

```
----------------------- Unk-Unigram Generated Text -----------------------
Sequence Given : the
Generated Text : the business for within.
--------------------------------------------------------------------------
```

### *5.1.2 Probability Calculation*

The probability calculation function for UNK unigram follows closely the simplified implementation described in section 2.5. The only difference is that words in the given sequence are replaced with the UNK token if the same conditions are met, as those described in the previous section 5.1.1. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Unk-Unigram Sequence Probability --------------------
Sequence : this is a test
Probability : 1.2271257963021328e-11
-------------------------------------------------------------------------
```

## 5.2 Bigram Model

The UNK bigram model is generated in a similar way as the laplace unigram model. The benefits of using the lexicon to generate the new model from scratch, can be better seen in the bigram and trigram implementations. In the case that the previous bigram counts model was used, many checks would have had to be performed to convert words in nested dictionaries to UNK tokens, whilst also building the UNK nested dictionary with terms that follow the UNK token. This implementation would not only have been complicated to perform but would also take a lot of time to finish. Thus, by using the lexicon and the unigram counts model, checks are performed on the current term considered and the previous term, to check whether terms occur once. If so, the term is replaced by the UNK token and the bigram model is incremented the value according to the space it is being directed to by the key-terms. This is done for all sentences in the lexicon. The value for such terms is then divided by the total number of words that the previous term has, following it. By doing this, each word in the nested dictionary would have a value which refers to the probability of it occurring within a sentence after the previous term.

### *5.2.1 Text Generation*

The text generation function for UNK bigram follows closely the simplified implementation described in section 2.4, with the difference that words in the given sequence are replaced by the UNK token if need be, as described in section 5.1.1. The below is an example of generated text by using the model.

```
---------------------- Unk-Bigram Generated Text ----------------------
Sequence Given : the
Generated Text : the consideration of plots produced <UNK> <UNK> from wimbledon.
-----------------------------------------------------------------------
```

### *5.2.2 Probability Calculation*

The probability calculation function for UNK bigram follows closely the simplified implementation described in section 2.5, with the same difference as that described in the previous section. The below is an example of the probability of a specific sequence to be generated by using the model.

```
------------------- Unk-Bigram Sequence Probability --------------------
Sequence : this is a test
Probability : 1.6236291599429673e-08
------------------------------------------------------------------------
```

## 5.3 Trigram Model

The trigram model is generated entirely the same way as the bigram model with the only difference being that the trigram deals with the previous tuple (two terms) instead of just the previous term, when considering a specific word.

### *5.3.1 Text Generation*

The text generation function for UNK trigram follows closely the simplified implementation described in section 2.4, with the difference that words in the given sequence are replaced by the UNK token if need be, as described in section 5.1.1. The below is an example of generated text by using the model.

```
---------------------- Unk-Trigram Generated Text ----------------------
Sequence Given : the
Generated Text : the projected £4 million grand slam event.
------------------------------------------------------------------------
```

### *5.3.2 Probability Calculation*

The probability calculation function for UNK trigram follows closely the simplified implementation described in section 2.5, with the same difference as that described in the previous section. The below is an example of the probability of a specific sequence to be generated by using the model. As can be seen, a probability of 0 is produced which denotes that the sequence given cannot be generated using the UNK trigram model.

```
------------------- Unk-Trigram Sequence Probability --------------------
Sequence : this is a test
Probability : 0.0
------------------------------------------------------------------------
```

# 6. Interpolation

As described in the assignment specifications, interpolation was implemented as a class in itself which accepts the flavour of the interpolation method, along with the three other models that would have been generated beforehand. These three models would have the same version type as that denoted by the interpolation flavour.

## 6.1 Text Generation

When generating text, all words within the unigram model are considered and the interpolation formula is applied to retrieve the probability values of such words occurring within each model. These values are multiplied to the model's respective weight values (0.6 for Trigram, 0.3 for Bigram, 0.1 for Unigram) and then added together to form the final interpolation result for the whole word. This result is appended to the accumulator and if the accumulator is greater than or equal to the randomly generated number, the word is appended to the generated text sequence. This process is repeated until the end token is generated by the interpolation function.

When generating text, it was observed that when applying the interpolation formula to all possible words, words weren't always being generated. This issue was present when using the Vanilla and UNK versions of interpolation. The problem here was that the random number being generated was never being reached by the accumulator and thus, no word was being generated. This meant that the value of the sum of all interpolation probabilities for each word did not always add up to 1, but why was this the case? Upon further testing, it was observed that the maximum value of the accumulator was always around the 0.4 decimal value. This meant that values from the bigram and unigram models were being successfully calculated except for those of the trigram. What was really happening was that when a new word was being generated, the tuple at the end of the generated sequence which was newly created might not have been present in the trigram model beforehand. Thus, when looping to generate a new word using this tuple history, the trigram would always output a probability of 0, since no entry of such tuple existed within the training set.

To account for such issue, a function to calculate the max range (between 0 and 1) that a random number could generate, was implemented. This function would sum all the interpolation results and return that result as a max range so that the random number generation could be done between 0 and the newly generated range. By doing this, generation of words is always ensured, thus avoiding unwanted errors.

### *6.1.1 Vanilla*

The below is an example of generated text by using vanilla interpolation.

```
---------------------- Vanilla-Interpolation Generated Text ----------------------
Sequence Given : the
Generated Text : the basic offence in spite of view 3 and drive to.
----------------------------------------------------------------------------------
```

### *6.1.2 Laplace*

The below is an example of generated text by using laplace interpolation.

```
---------------------- Laplace-Interpolation Generated Text ----------------------
Sequence Given : the
Generated Text : the kickable anteriority.
----------------------------------------------------------------------------------
```

### *6.1.3 UNK*

The below is an example of generated text by using UNK interpolation.

```
---------------------- Unk-Interpolation Generated Text ----------------------
Sequence Given : the
Generated Text : the basis for people having a qv x which we picked at longchamp yesterday.
------------------------------------------------------------------------------
```

## 6.2 Probability Calculation

When calculating probabilities, the same implementations as those described in the previous sections were adopted for each version of the interpolation algorithm. In the Vanilla version, the probability of the sentence is found by calculating the product of interpolation values for each term in the given sequence. This value is then multiplied to the probability value, initially set to 1, to find the probability of the sequence. In the Laplace version, the same process is done with the only exception that words within the given sequence are checked for whether they exist in the training set and if their count is 0. This check is performed when trying to find the probability value for both bigram and trigram models and if the condition is satisfied, the default laplace value is found for both models. This value is then multiplied to the model's respective weight and the answer is used as the interpolation result of the specific model. In the UNK version, the given sequence is checked for any words that are unknown in the training set or that have a count of 1. Given this condition is satisfied, these words are replaced with the unknown token and the probability is found the same way as that described for the vanilla version.

### *6.2.1 Vanilla*

The below is an example of the probability of a specific sequence to be generated using vanilla interpolation.

```
------------------- Vanilla-Interpolation Sequence Probability --------------------
Sequence : this is a test
Probability : 7.134755621926593e-07
-----------------------------------------------------------------------------------
```

### *6.2.2 Laplace*

The below is an example of the probability of a specific sequence to be generated using laplace interpolation.

```
------------------- Laplace-Interpolation Sequence Probability --------------------
Sequence : this is a test
Probability : 5.455153692478033e-11
-----------------------------------------------------------------------------
```

### *6.2.3 UNK*

The below is an example of the probability of a specific sequence to be generated using UNK interpolation.

```
-------------------- Unk-Interpolation Sequence Probability ---------------------
Sequence : this is a test
Probability : 7.134755621926593e-07
-----------------------------------------------------------------------------
```

# 7. Testing the models

## 7.1 Time to generate models

```
------------------------- Generating Language Models -------------------------

--- 0.12 seconds to generate Vanilla Unigram ---
--- 1.49 seconds to generate Vanilla Bigram ---
--- 7.48 seconds to generate Vanilla Trigram ---

--- 0.09 seconds to generate Laplace Unigram ---
--- 1.52 seconds to generate Laplace Bigram ---
--- 9.11 seconds to generate Laplace Trigram ---

--- 0.9 seconds to generate UNK Unigram ---
--- 1.95 seconds to generate UNK Bigram ---
--- 3.9 seconds to generate UNK Trigram ---

--- 1.33 seconds to generate Vanilla Interpolation ---
--- 1.2 seconds to generate Laplace Interpolation ---
--- 1.27 seconds to generate UNK Interpolation ---


--- 34.52 seconds to generate all models and necessities ---
```

The image above shows the rough amount of time needed to generate each and every model within the project. This time is heavily dependent on how large the dataset used to train the models, is. Time is also dependent on which machine the program is computed on. These results were found from my personal computer which is relatively powerful in terms of specifications. When using devices such as laptops with lower-end specs, the time needed to generate the models would hence be slightly more. One should note that when considering a larger history, the time needed to generate n-grams is substantially increased. For example, when considering trigrams which are n-grams of the highest order within this project, the time required for them to be generated is much larger, when compared to the unigram or bigram equivalents. Thus, when using n-grams of higher orders one should determine how feasible it is to create such models and if the text produced by them compensates for the time needed to generate such models.

## 7.2 Testing using the test set

### *7.2.1 Vanilla Models*

```
--------------------------------------------------------------------------------

Model percentage history (percentage of sequences which have zero probability)

Vanilla-Unigram: 24.7%
Vanilla-Bigram: 91.57%
Vanilla-Trigram: 94.62%
Vanilla-Interpolation: 24.05%

--------------------------------------------------------------------------------
```

When finding the probability of each sentence within the given test set, the above percentages where found for each vanilla model, which denote the percentage of sentences that had a zero probability of being generated by the respective model. As shown above, the vanilla unigram and interpolation models are by far more likely to generate more sentences found in the test set, other than the bigram and trigram models.

### *7.2.2 Laplace Models*

```
--------------------------------------------------------------------------------

Model percentage history (percentage of sequences which have zero probability)

Laplace-Unigram: 24.7%
Laplace-Bigram: 23.04%
Laplace-Trigram: 24.21%
Laplace-Interpolation: 0.07%

--------------------------------------------------------------------------------
```

When finding the probability of each sentence within the given test set, the above percentages where found for each laplace model, which denote the percentage of sentences that had a zero probability of being generated by the respective model. As shown above, the laplace unigram, bigram and trigram models were observed to be very similar in percentage values. The three models are observed to only have around an estimated 24% chance of not generating the sentences found in the given test set. However, when compared to the laplace interpolation method which outputted a percentage of 0.07%, these models are made inferior. The percentage calculated by interpolation shows how generalised the laplace interpolation method is and how it can be applied to generate multiple other texts which other models are unable to do.

*7.2.3 UNK Models*

```
--------------------------------------------------------------------------------

Model percentage history (percentage of sequences which have zero probability)

Unk-Unigram: 0.0%
Unk-Bigram: 90.5%
Unk-Trigram: 94.37%
Unk-Interpolation: 0.01%

--------------------------------------------------------------------------------
```

When finding the probability of each sentence within the given test set, the above percentages where found for each UNK model, which denote the percentage of sentences that had a zero probability of being generated by the respective model. As shown above, the UNK unigram and interpolation models are by far superior to the bigram and trigram UNK models. Both of these models generated a percentage equal/close to zero which means that they are both able to generate almost all of the sentences within the test set. Similar to the laplace interpolation model, these results show how efficient both models are for generating a wide variety of texts. However, when generating texts using such models, one has to consider the fact that UNK tokens have a probability to be displayed within the text rather than just English literals.

# 8. Limitations and Improvements

## 8.1 Limitations

- Having larger a dataset offers a higher generalisation of the models. With more sentences that the n-grams can be trained on, the less likely the chances of providing a zero probability when finding the probability of a specific sequence within the test set. However, the amount of data that could be used was heavily limited due to machine specifications. Given that the program would have been run on a super computer, more data could have been used and the time need to generate the models and perform operations would still be much less. However, given this limitation, the dataset had to be reduced to account for this issue.

- Due to the use of the beautiful soup library, errors having to do with Unicode decoding when printing and writing text, might occur. This was solved when writing to files by specifying the encoding type to UTF-8. However, when generating text for example, a term might still produce such decoding errors. When trying to solve this issue by specifying the word's encoding, the term was always printed within comments and was appended by a 'b' at the front. This wasn't ideal especially since further formatting of terms was needed, and hence was disregarded.

## 8.2 Improvements

- At its current state, the program takes around 30 seconds to generate all models from scratch when run. As a future improvement, given that the same training data would be used, it would be ideal to store the models within JSON files thus saving the models for future use. This would remove the need to generate the model dictionaries from scratch each time the program is run, thus saving a lot of time.

# References

[1] "Beautiful Soup Documentation — Beautiful Soup 4.4.0 Documentation". *Crummy.Com*, 2020, https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[2] "Re — Regular Expression Operations — Python 3.8.2 Documentation". *Docs.Python.Org*, 2020, https://docs.python.org/3/library/re.html.

[3] "8.3. Collections — High-Performance Container Datatypes — Python 2.7.17 Documentation". *Docs.Python.Org*, 2020, https://docs.python.org/2/library/collections.html.