L-Università ta' Malta

Statistical Natural Language Processing

Building a Language Model

ICS2203

Programmed using Python

By Mark Dingli
20703H

Table of contents

Introduction

In this documentation, I will be presenting three distinct N-gram models created using the (Baby) British National Corpus as the basis. The models include unigram, bigram, and trigram, implemented using a vanilla language model, a Laplace smoothing language model, and a language model with unknown word handling. Additionally, the documentation will provide a detailed explanation of the evaluation metrics used to compare the models' performances, including perplexity and accuracy scores. Furthermore, the advantages, testing and limitations of each model will be discussed, along with suggestions for possible improvements or modifications. The primary aim of this documentation is to provide an overview of the implementation process for each model, as well as to test and compare their respective performances on the given corpus.

Imports

Various Python libraries were imported and utilized. The Natural Language Toolkit (*nltk*) library was imported for text processing, including tokenization, lemmatization. The *RegexpTokenizer* class from the *nltk.tokenize* module was used to tokenize the text. The *WordNetLemmatizer* class from the *nltk.stem* module was used to perform lemmatization. The *sklearn.model_selection* library was used to split the data into training and testing sets. Additionally, *psutil* library was imported to monitor the system's memory usage during the execution of the program.

Extracting and Pre-processing the Selected Corpus

Description

The corpus used in this project is the (Baby) British National Corpus, which is a collection of written and spoken texts from a variety of sources [1].

Initially, a single XML file from the corpus was used to develop and test the pre-processing steps. Once the pre-processing steps were finalized, the entire corpus was used for N-gram modelling.

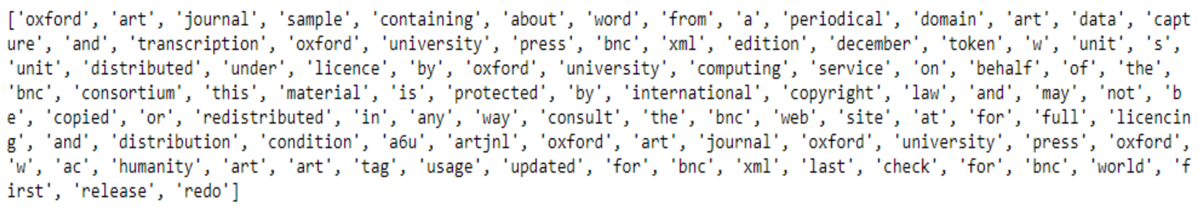
Before the corpus was used for N-gram modelling, it underwent several pre-processing steps to ensure that it was ready for analysis. Firstly, all XML markup was removed from the corpus using the *ElementTree* module [2]. Secondly, all non-alphabetic characters, such as numbers and punctuation, were removed using regular expressions [3]. Lastly, the corpus was tokenized using the *RegexpTokenizer* class from the *nltk.tokenize* module [4]. Additionally, the corpus text was converted to lowercase and lemmatized using the *WordNetLemmatizer* class from the *nltk.stem* module, which maps words to their base form.

It should be noted that the **entire corpus** was used for this project, which consisted of 4,003,540 words in total. The size of the corpus allows for a comprehensive analysis of the language patterns within it, which is essential for the creation of effective N-gram models.

Output

The output of this code is a list of tokens, which are the individual words or phrases that make up the text data. These tokens have been pre-processed to remove URLs, convert text to lowercase, expand contractions, remove punctuation and numbers, and lemmatize the remaining tokens [5]. The list of tokens is printed to the console for the first 100 tokens. The total size of the extracted corpus (the number of tokens) is also outputted.

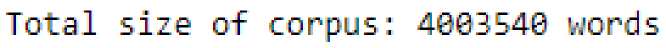
Example output of list of tokens:



```
['oxford', 'art', 'journal', 'sample', 'containing', 'about', 'word', 'from', 'a', 'periodical', 'domain', 'art', 'data', 'capt
ure', 'and', 'transcription', 'oxford', 'university', 'press', 'bnc', 'xml', 'edition', 'december', 'token', 'w', 'unit', 's',
'unit', 'distributed', 'under', 'licence', 'by', 'oxford', 'university', 'computing', 'service', 'on', 'behalf', 'of', 'the',
'bnc', 'consortium', 'this', 'material', 'is', 'protected', 'by', 'international', 'copyright', 'law', 'and', 'may', 'not', 'b
e', 'copied', 'or', 'redistributed', 'in', 'any', 'way', 'consult', 'the', 'bnc', 'web', 'site', 'at', 'for', 'full', 'licencin
g', 'and', 'distribution', 'condition', 'a6u', 'artjnl', 'oxford', 'art', 'journal', 'oxford', 'university', 'press', 'oxford',
'w', 'ac', 'humanity', 'art', 'art', 'tag', 'usage', 'updated', 'for', 'bnc', 'xml', 'last', 'check', 'for', 'bnc', 'world', 'f
irst', 'release', 'redo']
```

Figure 1 - Output showing the list of the first 100 tokens

Example output of corpus size:



```
Total size of corpus: 4003540 words
```

Figure 2 - Output showing the total size of the extracted corpus

Testing

It is important to note that the pre-processing steps used in this code were selected after testing and evaluating different techniques for tokenizing and cleaning the corpus. These steps were chosen as they produced the most effective results for N-gram modelling.

It was decided not to remove stop words during the pre-processing steps because they can provide useful contextual information for N-gram modelling. Removing stop words can result in a loss of meaning and can make it more difficult to accurately capture language patterns in the corpus. By keeping the stop words in the corpus, the resulting N-gram models can be more comprehensive and effective [14].

The code performs several checks to ensure that; the specified directory exists, it contains XML files, and that each XML file can be parsed before tokenizing its text.

Limitations

While this code provides an effective way to pre-process the corpus, it may be slow for very large corpora, as it needs to traverse all files in the specified directory and subdirectories. Additionally, the pre-processing steps used in this code may not be suitable for all types of text data, and may need to be modified depending on the nature of the corpus. It should also be noted that the corpus was tokenized into words, not sentences, which may impact the accuracy of certain language models.

Splitting the corpus

Description

In order to evaluate the performance of the N-gram models, it is necessary to split the corpus into a train set and a test set. The train set will be used to train the model, while the test set will be used to evaluate its performance. After pre-processing, the corpus was split into training and testing sets using the *train_test_split* method from the *sklearn.model_selection* library [8]. The test size parameter is set to 0.2, meaning that 20% of the tokens will be used for testing and the remaining 80% will be used for training.

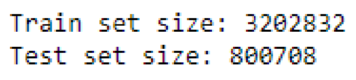
The split of 80% training data and 20% testing data is a common split in machine learning tasks, including N-gram language modelling. This split is generally a good starting point, as it provides enough data for the model to learn from while also allowing for a sufficient evaluation of the model's performance [6][7].

After splitting the tokens, the code defines the vocabulary from the training set. This is done by iterating over each sentence in the train set and adding each word to a set called *vocab*. The *vocab* set is then converted to a list and sorted. The vocabulary is defined by creating a set of all the unique words that appear in the training set. Then, the set is converted to a list and sorted alphabetically. This list of words is used to define the vocabulary for the N-gram models. The purpose of creating the vocabulary is to identify the set of words that the model can use to generate text. The vocabulary is used during the training process to generate the probability distribution of words in the corpus. During the training phase of the N-gram models, the vocabulary is used to compute the probabilities of each word in the corpus. This probability distribution is then used to generate the most probable next word in the sequence during text generation [9].

Output

The code prints the size of the train set and the test set using the *len()* function.

Example output of corpus split:



```
Train set size: 3202832
Test set size: 800708
```

Figure 3 - Output showing the size of the two split sets

Testing

It is important to note that the size of the train and test sets can be adjusted depending on the size of the corpus and the desired level of accuracy. In this case, the train set was set to 80% of the corpus and the test set was set to 20%. The train set size is 3,202,832 tokens and the test set size is 800,708 tokens.

Limitations

The size of the corpus and the desired level of accuracy can affect the size of the train and test sets. In addition, if the corpus is very large, the code may take longer to run when splitting the corpus [7].

Vanilla Language model

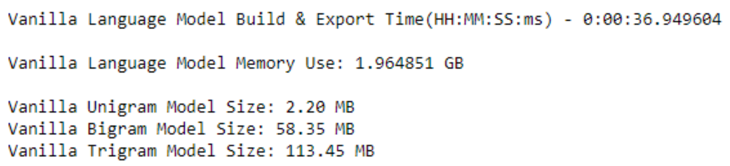
Description

To build the model, the code first defines a function *create_vanilla_ngrams()* that takes a list of tokens and an integer *n* as inputs and returns a list of n-grams of size *n* from the input list. Then, the code creates a vanilla unigram model by iterating over the training data and counting the frequency of each word. The frequency of each word is divided by the total number of unigrams to calculate the probability of each word occurring. Next, the code creates a vanilla bigram model by creating bigrams from the training data and counting the frequency of each bigram [18]. The frequency of each bigram is divided by the frequency of its first word to calculate the probability of each bigram occurring. Finally, the code creates a vanilla trigram model by creating trigrams from the training data and counting the frequency of each trigram. The frequency of each trigram is divided by the frequency of its first two words to calculate the probability of each trigram occurring.

The resulting vanilla unigram, bigram, and trigram probabilities are combined into a single dictionary and saved in JSON files to be exported. Using the *RAMusage()*, *datetime.now()* and the *os.path.getsize()* functions, the size of the resulting JSON files are calculated, and the time taken to build and export the models, as well as the memory used by the model, are printed as output.

Output

The output (figure 4) shows the time taken to build and export the Vanilla Language Model, which is approximately 36.94 seconds. It also shows the amount of memory used by the model during the process, which is around 1.96 GB. Finally, it displays the size of the resulting JSON files that contains the different Vanilla Language Model. The size of the Vanilla Unigram Model was approximately 2.20 MB, the size of the Vanilla Bigram Model was around 58.35 MB, and the size of the Vanilla Trigram Model was about 113.45 MB. These metrics can be useful in evaluating the efficiency and effectiveness of the Vanilla Language Model implementation [15-17].

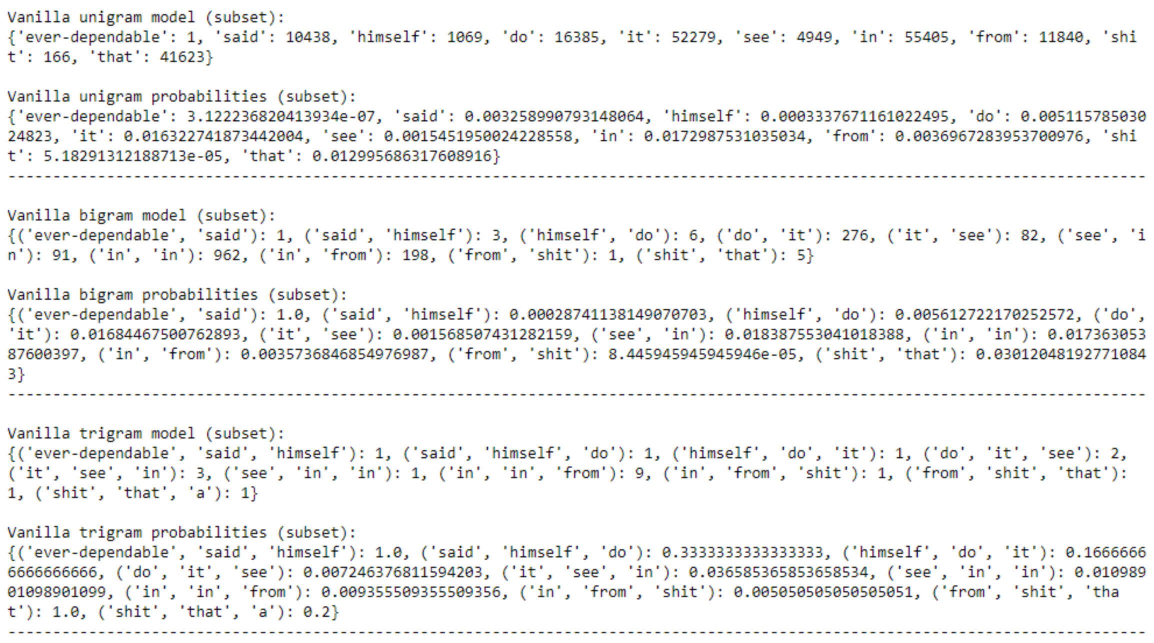


```
Vanilla Language Model Build & Export Time(HH:MM:SS:ms) - 0:00:36.949604

Vanilla Language Model Memory Use: 1.964851 GB

Vanilla Unigram Model Size: 2.20 MB
Vanilla Bigram Model Size: 58.35 MB
Vanilla Trigram Model Size: 113.45 MB
```

Figure 4 - Output showing the time, memory and file size of the Vanilla Language Models

Testing

After building and exporting the Vanilla Language Models, it is important to test the model to ensure that it has been created correctly. In the code testing section, subsets of the model and probability dictionaries are printed to verify the accuracy of the model. The code prints the frequency of the first 10 unigrams, bigrams, and trigrams in the model, as well as their corresponding probabilities (Figure 5). The subsets of the model can be modified by changing the range of the slices in the list() function. The output can be used to verify that the frequency and probability calculations have been performed correctly and to identify any potential issues with the model, such as low-frequency or low-probability n-grams. This can be useful in assessing the effectiveness of the language model for its intended purpose and in making decisions about its refinement or improvement.



```
Vanilla unigram model (subset):
{'ever-dependable': 1, 'said': 10438, 'himself': 1069, 'do': 16385, 'it': 52279, 'see': 4949, 'in': 55405, 'from': 11840, 'shi
t': 166, 'that': 41623}

Vanilla unigram probabilities (subset):
{'ever-dependable': 3.122236820413934e-07, 'said': 0.003258990793148064, 'himself': 0.0003337671161022495, 'do': 0.005115785030
24823, 'it': 0.016322741873442004, 'see': 0.001545195002422858, 'in': 0.0172987531035034, 'from': 0.003696728395370976, 'shi
t': 5.18291312188713e-05, 'that': 0.012995686317608916}
------------------------------------------------------------------------------------------------------------------------------

Vanilla bigram model (subset):
{('ever-dependable', 'said'): 1, ('said', 'himself'): 3, ('himself', 'do'): 6, ('do', 'it'): 276, ('it', 'see'): 82, ('see', 'i
n'): 91, ('in', 'in'): 962, ('in', 'from'): 198, ('from', 'shit'): 1, ('shit', 'that'): 5}

Vanilla bigram probabilities (subset):
{('ever-dependable', 'said'): 1.0, ('said', 'himself'): 0.00028741138149070703, ('himself', 'do'): 0.005612722170252572, ('do',
'it'): 0.01684467500762893, ('it', 'see'): 0.001568507431282159, ('see', 'in'): 0.018387553041018388, ('in', 'in'): 0.017363053
87600397, ('in', 'from'): 0.0035736846854976987, ('from', 'shit'): 8.445945945945946e-05, ('shit', 'that'): 0.03012048192771084
3}
------------------------------------------------------------------------------------------------------------------------------

Vanilla trigram model (subset):
{('ever-dependable', 'said', 'himself'): 1, ('said', 'himself', 'do'): 1, ('himself', 'do', 'it'): 1, ('do', 'it', 'see'): 2,
('it', 'see', 'in'): 3, ('see', 'in', 'in'): 1, ('in', 'in', 'from'): 9, ('in', 'from', 'shit'): 1, ('from', 'shit', 'that'):
1, ('shit', 'that', 'a'): 1}

Vanilla trigram probabilities (subset):
{('ever-dependable', 'said', 'himself'): 1.0, ('said', 'himself', 'do'): 0.3333333333333333, ('himself', 'do', 'it'): 0.1666666
6666666666, ('do', 'it', 'see'): 0.007246376811594203, ('it', 'see', 'in'): 0.036585365853658534, ('see', 'in', 'in'): 0.010989
01098901099, ('in', 'in', 'from'): 0.009355509355509356, ('in', 'from', 'shit'): 0.005050505050505051, ('from', 'shit', 'tha
t'): 1.0, ('shit', 'that', 'a'): 0.2}
------------------------------------------------------------------------------------------------------------------------------
```

Figure 5 - Output showing the subsets of the Vanilla Model (testing)

Limitations

The code requires a significant amount of memory to build the language model, especially for larger datasets. This can be a limiting factor for some systems with limited memory capacity.

Additional Notes

It's important to note that the JSON files created by the Vanilla Language Model code are not used for the subsequent code that performs linear interpolation, calculates perplexity and generates text. Instead, the probability dictionaries are loaded directly into memory to perform these tasks. One advantage of this approach is that it can be faster to access the probability dictionaries directly in memory rather than reading from a file. However, a disadvantage is that it can consume more memory, particularly for larger datasets.

The JSON files can still be used to share the Vanilla Language Model or in other applications that require language modelling. This can be particularly useful when working with multiple programming languages or when collaborating with other researchers who may not have access to the original code or corpus used to build the model. Furthermore, the JSON files can be used as a backup or archive of the model, which can be reloaded into memory in case of system failures or other issues that may affect the performance of the language model. The disadvantage is that the model dictionaries will be lost once the program is terminated, requiring the model to be rebuilt from scratch the next time the program is run.

Laplace Language model

Description

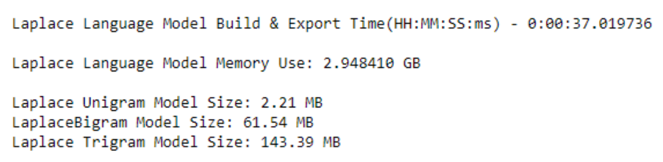
The Laplace Language Model code is similar to the Vanilla Language Model code, but with the addition of Laplace smoothing. The Laplace model takes the Vanilla model as its basis. Laplace smoothing is used to account for unknown words and to avoid zero probabilities when computing n-gram probabilities. The *create_laplace_ngrams()* function is used to create n-grams of size *n* from the input list of tokens. Then, the code creates a Laplace unigram model by iterating over the training data and counting the frequency of each word. The frequency of each word is divided by the total number of unigrams plus the vocabulary size, and one is added to the numerator and the vocabulary size in the denominator to account for unknown words. The Laplace bigram and trigram models are created similarly, with the frequency of each n-gram divided by the frequency of its first (n-1) words plus the vocabulary size, and one added to the numerator and the vocabulary size in the denominator to account for unknown words. The code for the Laplace Language Model was inspired from [22] and [23].

The resulting Laplace unigram, bigram, and trigram probabilities are combined into a single dictionary and saved in JSON files to be exported. Using the *RAMusage()*, *datetime.now()* and the *os.path.getsize()* functions, the size of the resulting JSON files are calculated, and the time taken to build and export the models, as well as the memory used by the model, are printed as output.

Output

The output (figure 6) shows the time taken to build and export the Laplace Language Model, which is approximately 37.01 seconds. It also shows the amount of memory used by the model during the process, which is around 2.95 GB. Finally, it displays the size of the resulting JSON files that contains the different Laplace Language Model. The size of the Laplace Unigram Model was approximately 2.21 MB, the size of the Laplace Bigram Model was around 61.54 MB, and the size of the Laplace Trigram Model was about 143.39 MB. These metrics can be useful in evaluating the efficiency and effectiveness of the Laplace Language Model implementation, and comparing it to the Vanilla Language Model [15-17].

Additionally, the Laplace Language Model requires more memory compared to the Vanilla Language Model, due to the larger probability dictionaries resulting from the Laplace smoothing technique. However, this trade-off can improve the model's performance, particularly for smaller datasets or datasets with low-frequency words. The resulting JSON file size of the Laplace Language Model is also larger compared to the Vanilla Language Model.

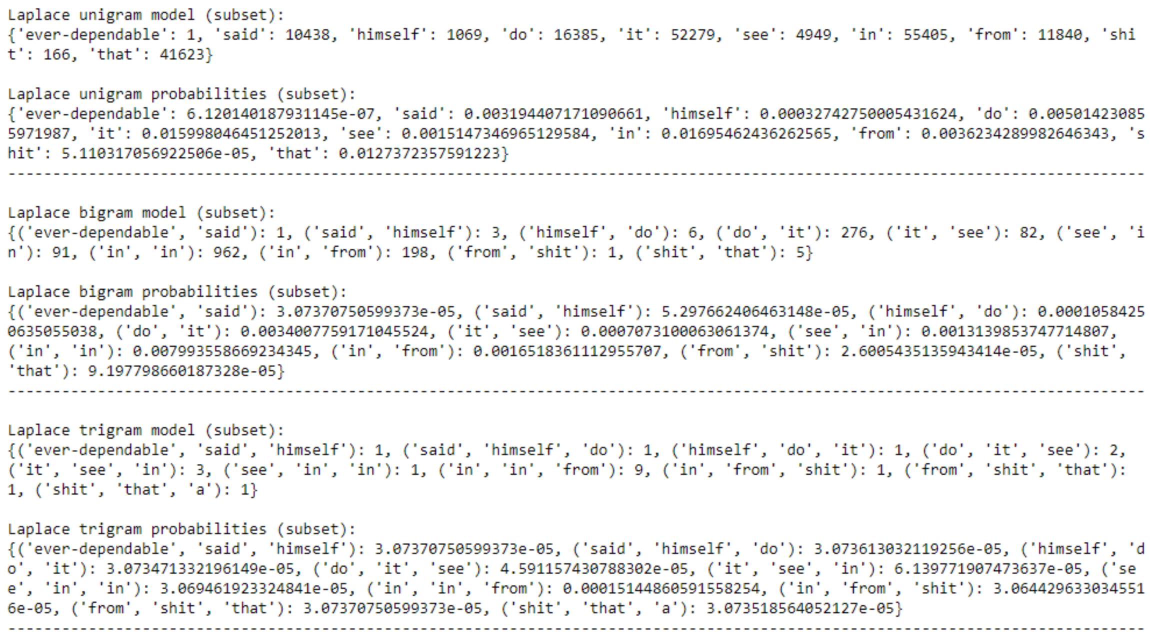


```
Laplace Language Model Build & Export Time(HH:MM:SS:ms) - 0:00:37.019736

Laplace Language Model Memory Use: 2.948410 GB

Laplace Unigram Model Size: 2.21 MB
LaplaceBigram Model Size: 61.54 MB
Laplace Trigram Model Size: 143.39 MB
```

Figure 6 - Output showing the time, memory and file size of the Laplace Language Models

Testing

After building and exporting the Laplace Language Model, it is important to test the model to ensure that it has been created correctly. This is done the same way as the previous model. In the code testing section, subsets of the model and probability dictionaries are printed to verify the accuracy of the model. The code prints the frequency of the first 10 unigrams, bigrams, and trigrams in the model, as well as their corresponding probabilities (Figure 7).



```
Laplace unigram model (subset):
{'ever-dependable': 1, 'said': 10438, 'himself': 1069, 'do': 16385, 'it': 52279, 'see': 4949, 'in': 55405, 'from': 11840, 'shi
t': 166, 'that': 41623}

Laplace unigram probabilities (subset):
{'ever-dependable': 6.120140187931145e-07, 'said': 0.003194407171090661, 'himself': 0.00032742750005431624, 'do': 0.00501423085
5971987, 'it': 0.015998046451252013, 'see': 0.0015147346965129584, 'in': 0.01695462436262565, 'from': 0.0036234289982646343, 's
hit': 5.110317056922506e-05, 'that': 0.0127372357591223}
--------------------------------------------------------------------------------------------------------------------------

Laplace bigram model (subset):
{('ever-dependable', 'said'): 1, ('said', 'himself'): 3, ('himself', 'do'): 6, ('do', 'it'): 276, ('it', 'see'): 82, ('see', 'i
n'): 91, ('in', 'in'): 962, ('in', 'from'): 198, ('from', 'shit'): 1, ('shit', 'that'): 5}

Laplace bigram probabilities (subset):
{('ever-dependable', 'said'): 3.07370750599373e-05, ('said', 'himself'): 5.297662406463148e-05, ('himself', 'do'): 0.0001058425
0635055038, ('do', 'it'): 0.0034007759171045524, ('it', 'see'): 0.0007073100063061374, ('see', 'in'): 0.0013139853747714807,
('in', 'in'): 0.007993558669234345, ('in', 'from'): 0.0016518361112955707, ('from', 'shit'): 2.6005435135943414e-05, ('shit',
'that'): 9.197798660187328e-05}
--------------------------------------------------------------------------------------------------------------------------

Laplace trigram model (subset):
{('ever-dependable', 'said', 'himself'): 1, ('said', 'himself', 'do'): 1, ('himself', 'do', 'it'): 1, ('do', 'it', 'see'): 2,
('it', 'see', 'in'): 3, ('see', 'in', 'in'): 1, ('in', 'in', 'from'): 9, ('in', 'from', 'shit'): 1, ('from', 'shit', 'that'):
1, ('shit', 'that', 'a'): 1}

Laplace trigram probabilities (subset):
{('ever-dependable', 'said', 'himself'): 3.07370750599373e-05, ('said', 'himself', 'do'): 3.073613032119256e-05, ('himself', 'd
o', 'it'): 3.073471332196149e-05, ('do', 'it', 'see'): 4.591157430788302e-05, ('it', 'see', 'in'): 6.139771907473637e-05, ('se
e', 'in', 'in'): 3.069461923324841e-05, ('in', 'in', 'from'): 0.00015144860591558254, ('in', 'from', 'shit'): 3.064429633034551
6e-05, ('from', 'shit', 'that'): 3.07370750599373e-05, ('shit', 'that', 'a'): 3.073518564052127e-05}
--------------------------------------------------------------------------------------------------------------------------
```

Figure 7 - Output showing the subsets of the Laplace Model (testing)

Limitations

Like the previous model, the code requires a significant amount of memory to build the language model, especially for larger datasets. This can be a limiting factor for some systems with limited memory capacity.

Additional Notes

Just like the previous model, the Laplace Language Model code creates JSON files that contains the Laplace unigram, bigram, and trigram probabilities. Similar to the Vanilla Language Model, these files can be used to share the Laplace Language Model or in other applications that require language modelling. However, it's important to note that these JSON files are not used for subsequent tasks such as linear interpolation, perplexity calculation, and text generation.

Instead, the Laplace Language Model code loads the Laplace unigram, bigram, and trigram probability dictionaries directly into memory to perform these tasks. This approach can be faster than reading from a file but may consume more memory, especially for larger datasets.

UNK Language model

Description

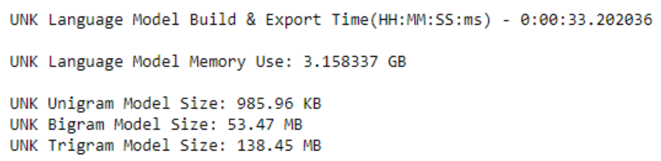
The UNK Language Model code is an extension of the Laplace Language Model code that takes into account unknown words. The UNK model is built on the same principles as the Laplace model, but it creates a new list with <UNK> tokens for words with count less than or equal to 2. Then, the Laplace unigram, bigram, and trigram models are created using the frequency of each n-gram divided by the frequency of its first (n-1) words plus the vocabulary size, and one added to the numerator and the vocabulary size in the denominator to account for unknown words. Removing low-frequency words from both the training and test sets ensures that the model will perform similarly on both the training and test data, as both sets will now contain the same set of vocabulary. This helps to avoid any overfitting that may occur if the model has learned to depend on specific rare words in the training data that do not occur in the test data. Code was inspired by [24] and [25].

The resulting UNK unigram, bigram, and trigram probabilities are combined into a single dictionary and saved in JSON files to be exported. Using the *RAMusage()*, *datetime.now()* and the *os.path.getsize()* functions, the size of the resulting JSON files are calculated, and the time taken to build and export the models, as well as the memory used by the model, are printed as output.

Output

The output (Figure 8) shows the time taken to build and export the UNK Language Model, which is approximately 33.20 seconds. It also shows the amount of memory used by the model during the process, which is around 3.15 GB. Finally, it displays the size of the resulting JSON files that contains the different UNK Language Model. The size of the UNK Unigram Model was approximately 985.96 KB, the size of the UNK Bigram Model was around 53.47 MB, and the size of the UNK Trigram Model was about 138.45 MB. These metrics can be useful in evaluating the efficiency and effectiveness of the UNK Language Model implementation, and comparing it to the Vanilla and Laplace Language Models [15-17].

Additionally, the UNK Language Model requires more memory compared to the Vanilla Language Model, due to the larger probability dictionaries resulting from the Laplace smoothing technique. However, this trade-off can improve the model's performance, particularly for smaller datasets or datasets with low-frequency words. The resulting JSON file size of the UNK Language Model is also larger compared to the Vanilla Language Model.

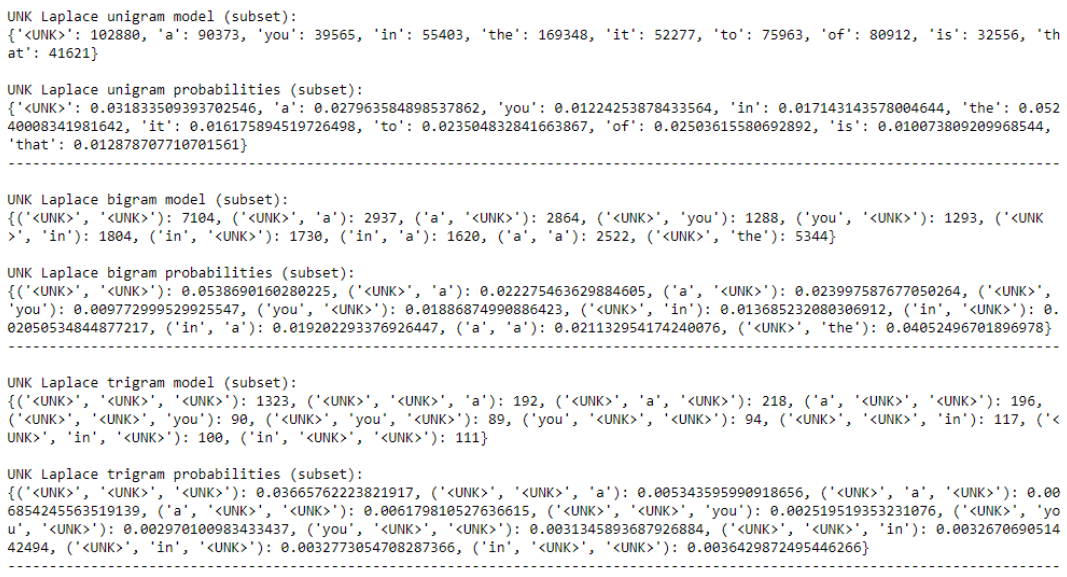


```
UNK Language Model Build & Export Time(HH:MM:SS:ms) - 0:00:33.202036

UNK Language Model Memory Use: 3.158337 GB

UNK Unigram Model Size: 985.96 KB
UNK Bigram Model Size: 53.47 MB
UNK Trigram Model Size: 138.45 MB
```

Figure 8 - Output showing the time, memory and file size of the Laplace Language Models

Testing

After building and exporting the UNK Language Model, it is important to test the model to ensure that it has been created correctly. In the testing code section, subsets of the model and probability dictionaries are printed to verify the accuracy of the model. The code prints the frequency of the first 10 unigrams, bigrams, and trigrams in the model that contain unknown words represented by the <UNK> token, as well as their corresponding probabilities (Figure 9). The subsets of the model can be modified by changing the range of the slices in the list() function.



```
UNK Laplace unigram model (subset):
{'<UNK>': 102880, 'a': 90373, 'you': 39565, 'in': 55403, 'the': 169348, 'it': 52277, 'to': 75963, 'of': 80912, 'is': 32556, 'th
at': 41621}

UNK Laplace unigram probabilities (subset):
{'<UNK>': 0.03183350939702546, 'a': 0.027963584898537862, 'you': 0.01224253878433564, 'in': 0.017143143578004644, 'the': 0.052
40008341981642, 'it': 0.016175894519726498, 'to': 0.023504832841663867, 'of': 0.02503615580692892, 'is': 0.010073809209968544,
'that': 0.012878707710701561}
-----------------------------------------------------------------------------------------------------------------------------

UNK Laplace bigram model (subset):
{('<UNK>', '<UNK>'): 7104, ('<UNK>', 'a'): 2937, ('a', '<UNK>'): 2864, ('<UNK>', 'you'): 1288, ('you', '<UNK>'): 1293, ('<UNK
>', 'in'): 1804, ('in', '<UNK>'): 1730, ('in', 'a'): 1620, ('a', 'a'): 2522, ('<UNK>', 'the'): 5344}

UNK Laplace bigram probabilities (subset):
{('<UNK>', '<UNK>'): 0.0538690160280225, ('<UNK>', 'a'): 0.022275463629884605, ('a', '<UNK>'): 0.023997587677050264, ('<UNK>',
'you'): 0.009772999529925547, ('you', '<UNK>'): 0.01886874990886423, ('<UNK>', 'in'): 0.013685232080306912, ('in', '<UNK>'): 0.
02050534844877217, ('in', 'a'): 0.019202293376926447, ('a', 'a'): 0.021132954174240076, ('<UNK>', 'the'): 0.04052496701896978}
-----------------------------------------------------------------------------------------------------------------------------

UNK Laplace trigram model (subset):
{('<UNK>', '<UNK>', '<UNK>'): 1323, ('<UNK>', '<UNK>', 'a'): 192, ('<UNK>', 'a', '<UNK>'): 218, ('a', '<UNK>', '<UNK>'): 196,
('<UNK>', '<UNK>', 'you'): 90, ('<UNK>', 'you', '<UNK>'): 89, ('you', '<UNK>', '<UNK>'): 94, ('<UNK>', '<UNK>', 'in'): 117, ('<
UNK>', 'in', '<UNK>'): 100, ('in', '<UNK>', '<UNK>'): 111}

UNK Laplace trigram probabilities (subset):
{('<UNK>', '<UNK>', '<UNK>'): 0.0366576223821917, ('<UNK>', '<UNK>', 'a'): 0.005343595990918656, ('<UNK>', 'a', '<UNK>'): 0.00
6854245563519139, ('a', '<UNK>', '<UNK>'): 0.006179810527636615, ('<UNK>', '<UNK>', 'you'): 0.002519519353231076, ('<UNK>', 'yo
u', '<UNK>'): 0.002970100983433437, ('you', '<UNK>', '<UNK>'): 0.0031345893687926884, ('<UNK>', '<UNK>', 'in'): 0.0032670690514
42494, ('<UNK>', 'in', '<UNK>'): 0.0032773054708287366, ('in', '<UNK>', '<UNK>'): 0.0036429872495446266}
-----------------------------------------------------------------------------------------------------------------------------
```

Figure 9 - Output showing the subsets of the Laplace Model (testing)

Limitations

Like the previous models, the code requires a significant amount of memory to build the language model, especially for larger datasets. This can be a limiting factor for some systems with limited memory capacity.

Additional Notes

Just like the previous models, the UNK Language Model code creates JSON files that contain the UNK unigram, bigram, and trigram probabilities. Similar to the Vanilla Language Model and Laplace Language Model, these files can be used to share the UNK Language Model, or it can be used in other applications that require language modelling. However, it's important to note that these JSON files are not used for subsequent tasks such as linear interpolation, perplexity calculation, and text generation.

Instead, the UNK Language Model code loads the UNK unigram, bigram, and trigram probability dictionaries directly into memory to perform these tasks. This approach can be faster than reading from a file but may consume more memory, especially for larger datasets. Since the UNK Language Model is designed to account for unknown words, it may have a better performance on datasets with many rare or unseen words, compared to the Vanilla Language Model.

It is important to note that while the UNK Language Model improves the performance of the language model on unseen words, it may not be able to capture the full complexity and diversity of language.

Linear Interpolation

Description

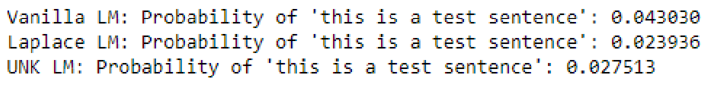
The function first sets the values for lambda_3, lambda_2, and lambda_1, which are the interpolation weights for the trigram, bigram, and unigram models (trigram = 0.6; bigram = 0.3; unigram = 0.1). The sentence is then tokenized using the split() method. The probabilities of each n-gram model are then calculated for the sentence using the corresponding language model specified by *lm_type*. The probabilities are smoothed using Laplace smoothing or UNK handling for unknown words, depending on the specified *lm_type*. The smoothed probabilities are stored in *unigram_probs*, *bigram_probs*, and *trigram_probs* lists [21].

Finally, the probability of the sentence is calculated using linear interpolation [10]. The smoothed probability of each n-gram is used in the interpolation formula to obtain the overall probability of the sentence. The *sentence_prob* variable is initialized to 1, and for each token in the sentence, the smoothed probability of the token is multiplied by the corresponding interpolation weight raised to the power of *lambda_1*. The resulting product is then multiplied by *sentence_prob*.

Output

The output shows the probability of each test sentence using three different types of N-gram models: Vanilla, Laplace, and UNK. The probabilities calculated for each sentence are different for each model, which shows that the model type has an impact on the probability assigned to a given sentence. The Vanilla LM assigned the highest probability to the test sentence, while the Laplace LM assigned the lowest probability. The UNK LM assigned a probability in between the Vanilla and Laplace models. These results also highlight the trade-offs between different types of N-gram models, such as the trade-off between accuracy and robustness when using the Laplace model versus the Vanilla model.

Example output of probability of test sentence with every model:



```
Vanilla LM: Probability of 'this is a test sentence': 0.043030
Laplace LM: Probability of 'this is a test sentence': 0.023936
UNK LM: Probability of 'this is a test sentence': 0.027513
```

Figure 10 - Output showing sentence probability

Testing

It should be noted that the code was tested on other test sentences and the results showed that the probability assigned to each sentence is consistent with the type of N-gram model used. If a word in the sentence is not in the vocabulary of the N-gram model, the code will encounter an error, as it will not be able to calculate the probability of that word. Therefore, it is important to ensure that the vocabulary used for the N-gram model includes all the words that may appear in the test sentences.

Limitations

The linear interpolation method does not take into account the context of the sentence beyond the N-gram model being used. For example, if a sentence contains a rare word that is not present in the N-gram model's vocabulary, the model may assign a low probability to the sentence even if the context of the sentence suggests a higher probability. This can result in inaccurate probability assignments for certain sentences and affect the overall accuracy of the model.

Evaluation & Perplexity

Description

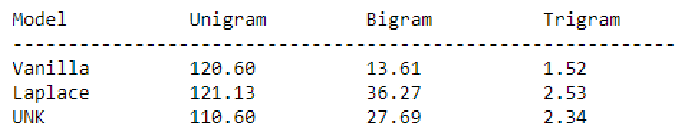
The perplexity values are numerical representations of how well the language model predicts the test set, with lower values indicating better performance [11]. The *calculate_perplexity* function takes in a *test_set*, a dictionary of n-gram language model probabilities, and the type of n-gram model (unigram, bigram, or trigram). It iterates over the *test_set* and calculates the log probability of each n-gram in the test set using the corresponding n-gram model probabilities [12]. It then calculates the entropy and perplexity of the test set using the log probabilities and returns the perplexity. The perplexities for each model and n-gram are printed in a table format.

Output

The output is a table that shows the perplexities for each of the three language models (Vanilla, Laplace, and UNK) and for each n-gram order (Unigram, Bigram, and Trigram).

In this case, the Vanilla language model has the lowest perplexities for the Bigram and Trigram orders, while the UNK language model has the lowest perplexity for the Unigram order. This indicates that the Vanilla language model performs well in predicting the next word given the previous two words, while the UNK model performs well in predicting the next word based on only the previous word [13]. The Laplace model generally has the highest perplexity values, indicating that it is the least accurate of the three models in predicting the test set.

Example output of the perplexities:



```
Model           Unigram         Bigram          Trigram
------------------------------------------------------------
Vanilla         120.60          13.61           1.52
Laplace         121.13          36.27           2.53
UNK             110.60          27.69           2.34
```

Figure 11 – Table output of perplexity

Testing

It should be noted that the code has only been tested on a single test set, and it would be beneficial to test the code on multiple test sets to ensure the accuracy and robustness of the language models. Additionally, the current test set may not be representative of all possible inputs, which could impact the accuracy of the language models. Therefore, it is important to test the code on multiple test sets to ensure that the models are generalizable and can perform well on a variety of inputs [13].

Limitations

One limitation of the perplexity measure is that it assumes that the test set is drawn from the same distribution as the training set. This assumption may not always hold, particularly if the test set contains rare or previously unseen words that were not present in the training set. In such cases, the perplexity measure may not accurately reflect the performance of the language model. Another limitation is that the linear interpolation of n-gram models could not be implemented.

Generation

Description

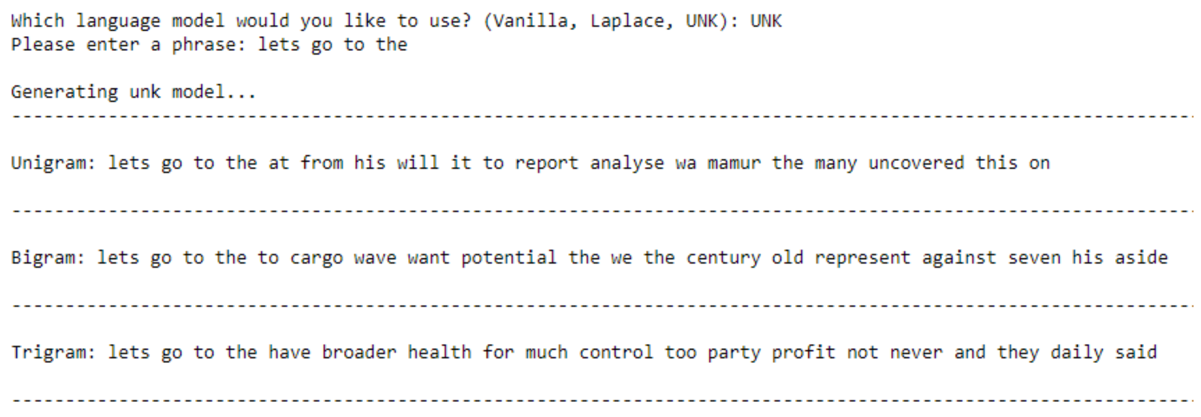
This code defines three functions to generate a sentence using an n-gram language model and a given phrase. The *generate_sentence()* function loops through each n-gram type and calls the *generate_ngram_sentence()* function to generate a sentence using the selected model, phrase, and n-gram type. The *generate_ngram_sentence()* function generates a sentence using a specific n-gram type and language model. It first sets start and end tokens for the sentence, and a maximum length. It then initializes the sentence with the start token and the words in the input phrase. The model probabilities are determined based on the selected model, and the sentence is generated by adding words one at a time until the end token is reached or the maximum length is reached. The code inspiration can be found in [19] and [20].

The *generate_word()* function generates a word given the probability distribution. If no context is provided, it uses the full vocabulary and corresponding probabilities. If a context is provided, it uses only the vocabulary and probabilities corresponding to that context. It should be noted that the generated sentences may not always make sense or be grammatically correct, as they are generated based solely on the language model probabilities and the input phrase.

Output

The output of the code is a generated sentence for each n-gram type (unigram, bigram, and trigram) using the specified language model and input phrase. The input phrase is appended to the start token ("<s>") to initiate the sentence generation. The generated sentence is limited to a maximum length of 20 words and is terminated when either the end token ("</s>") is reached or the maximum length is exceeded. The output demonstrates that the generated sentences can vary widely depending on the selected n-gram and language model.

Example output of sentence generation:



```
Which language model would you like to use? (Vanilla, Laplace, UNK): UNK
Please enter a phrase: lets go to the

Generating unk model...
----------------------------------------------------------------------------------------------------

Unigram: lets go to the at from his will it to report analyse wa mamur the many uncovered this on

----------------------------------------------------------------------------------------------------

Bigram: lets go to the to cargo wave want potential the we the century old represent against seven his aside

----------------------------------------------------------------------------------------------------

Trigram: lets go to the have broader health for much control too party profit not never and they daily said

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

Figure 12 - Output showing sentence generation

Testing

The code below is used for testing the sentence generation function with different language models and phrases. It iterates through a list of models and phrases and generates a sentence for each combination using the *generate_sentence()* function. The generated sentences are then printed in the console along with a separator line. This testing approach allows for a quick evaluation of how well the different language models can generate sentences given various phrases. To test out more phrases, just add a phrase in inverted commas in the *phrases* list.

```
# Test the sentence generation function with different models and phrases
models = ['Vanilla', 'Laplace', 'UNK']
phrases = ['this is a', 'i love', 'the weather is']

for model in models:
    for phrase in phrases:
        generate_sentence(model, phrase)
        print("-" * 125)
```

It is important to note that during the testing process, some of the generated sentences contained nonsensical words or phrases. This is likely due to the probabilistic nature of the language model and the fact that it is generating words based on probabilities derived from the training data.

Limitations

One limitation of the sentence generation function is that it relies solely on the n-gram language models to generate sentences, which may not always result in coherent or grammatically correct sentences. The function uses probabilistic models to generate the next word based on the previous words, and this approach may not always capture the complexity of language and context. Additionally, the function may generate sentences that contain rare or previously unseen words, which may not be familiar to the reader and could impact the overall comprehensibility of the generated sentences.

Sen_Probability

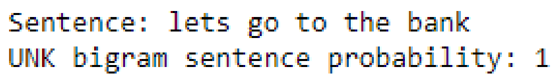
Description

The *sen_probability()* function calculates the probability of a given sentence using a specified language model and n-gram type. The input sentence is first tokenized into words, and then the function calculates the probability of the sentence for each model and n-gram type. The function supports three different models (Vanilla, Laplace, and UNK) and three different n-gram types (unigram, bigram, and trigram). The probability for each model and n-gram type is calculated by iterating through each word in the sentence and multiplying the appropriate probabilities for that word based on the selected model and n-gram type.

Output

The output of the *Sen_Probability()* function for the given sentence "lets go to the bank" and the selected language model "UNK" and n-gram type "bigram" is a probability value of 1. This may indicate that the language model is too confident in its prediction and may not be accurately capturing the complexity of the language.

Example output of the *Sen_Probability()* function:



```
Sentence: lets go to the bank
UNK bigram sentence probability: 1
```

Figure 13 – sen_probability output

Testing

Please note that the implementation of the sen_probability function in this code did not properly handle sentence segmentation. The function tokenizes the input sentence into individual words, which may not reflect the true probabilities of the sentence as a whole. Additionally, the test and train corpora were not split by sentence, which may further affect the accuracy of the language model. Future improvements to this code may involve proper sentence segmentation and splitting the corpora into sentences to improve the accuracy of the model.

Limitations

The function assumes that the sentence is properly tokenized and split into words, which may not always be the case. Finally, the function only considers probabilities based on the selected n-gram type and does not take into account the overall language model performance on the entire corpus. This can lead to inaccuracies if the sentence being evaluated deviates significantly from the training data.

Downloading Created Models

To download the language models created in this project, please follow these steps:

1. Click on the following Google Drive link:
   https://drive.google.com/drive/folders/1aJG04NllTjpjFjK5sAjsKCC_7aSUmEre?usp=share_link
2. You will be taken to a Google Drive folder containing the three folders for the Vanilla, Laplace, and UNK language models.
3. Each folder will contain 3 models, ‘unigram’, ‘bigram’ and ‘trigram’
4. Right-click on the file you wish to download and select "Download" from the dropdown menu.
5. Save the file to your desired location on your local machine.
6. Once you have downloaded the files, you can use them in your own NLP projects as needed.
7. If you have any issues downloading the models or questions about how to use them, please feel free to reach out to me for assistance.

References

[1] Bodleian Libraries, University of Oxford, "OTA: Oxford Text Archive," accessed March 7, 2023, https://ota.bodleian.ox.ac.uk/repository/xmlui/handle/20.500.12024/2553.

[2] Python Software Foundation, "19.7. xml.etree.ElementTree — The ElementTree XML API," Python 3.10.1 documentation, accessed March 7, 2023, https://docs.python.org/3/library/xml.etree.elementtree.html.

[3] Python Software Foundation, "6.2. re — Regular expression operations," Python 3.10.1 documentation, accessed March 7, 2023, https://docs.python.org/3/library/re.html.

[4] NLTK Project, "nltk.stem.wordnet.WordNetLemmatizer," Natural Language Toolkit 3.6.2 documentation, accessed March 7, 2023, https://www.nltk.org/api/nltk.stem.html#nltk.stem.wordnet.WordNetLemmatizer

[5] P. Goyal, "Text Preprocessing in Natural Language Processing using Python," Towards Data Science, November 10, 2019, https://towardsdatascience.com/text-preprocessing-in-natural-language-processing-using-python-6113ff5decd8.

[6] J. Brownlee, "How to Use Train-Test Split for Evaluating Machine Learning Algorithms," Machine Learning Mastery, accessed March 10, 2023, https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/.

[7] scikit-learn developers, "3.1. Cross-validation: evaluating estimator performance," scikit-learn 1.0.2 documentation, accessed March 7, 2023, https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation.

[8] scikit-learn developers, "sklearn.model_selection.train_test_split," scikit-learn 1.0.2 documentation, accessed March 10, 2023, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[9] M. Karpathy, "How to Create a Vocabulary for NLP Tasks in Python," KDnuggets, November 18, 2019, https://www.kdnuggets.com/2019/11/create-vocabulary-nlp-tasks-python.html.

[10] A. User, "Getting smoothed probabilities using linear interpolation for NLP model," Stack Overflow, March 10, 2020, https://stackoverflow.com/questions/60681202/getting-smoothed-probabilities-using-linear-interpolation-for-nlp-model.

[11] A. Raza, "Two Minutes NLP: Perplexity Explained with Simple Probabilities," Medium, accessed March 15, 2023, https://medium.com/nlplanet/two-minutes-nlp-perplexity-explained-with-simple-probabilities-6cdc46884584.

[12] Hugging Face, "Evaluating Language Models: Perplexity," Transformers documentation, accessed March 15, 2023, https://huggingface.co/docs/transformers/perplexity.

[13] Techslang, "Perplexity in NLP: Definition, Pros, and Cons," accessed March 15, 2023, https://www.techslang.com/perplexity-in-nlp-definition-pros-and-cons/#:~:text=The%20dictionary%20defines%20perplexity%20as,Updated%20September%2020%2C%202022.

[14] S. P. Teja, "Stop Words in NLP," Medium, accessed March 7, 2023, https://medium.com/@saitejaponugoti/stop-words-in-nlp-5b248dadad47.

[15] GeeksforGeeks, "Python | DateTime Module," accessed April 1, 2023, https://www.geeksforgeeks.org/python-datetime-module/.

[16] Python Software Foundation, "16.1. os.path — Common pathname manipulations," Python 3.10.1 documentation, accessed April 1, 2023, https://docs.python.org/3/library/os.path.html.

[17] Python Software Foundation, "19.2. json — JSON encoder and decoder," Python 3.10.1 documentation, accessed April 1, 2023, https://docs.python.org/3/library/json.html.

[18] S. S. Saha, "Understand and Building N-gram Model in NLP with Python," Towards AI, accessed March 12, 2023, https://pub.towardsai.net/understand-and-building-n-gram-model-in-nlp-with-python-addddbdb71fc.

[19] P. Goyal, "Text Generation using N-gram Model," Towards Data Science, accessed March 12, 2023, https://towardsdatascience.com/text-generation-using-n-gram-model-8d12d9802aa0.

[20] O. Borisov, "NGram_LanguageModel/text_generator_ngram.py," GitHub, accessed March 15, 2023, https://github.com/olegborisovv/NGram_LanguageModel/blob/main/text_generator_ngram.py.

[21] GeeksforGeeks, "How to Implement Linear Interpolation in Python?" accessed March 13, 2023, https://www.geeksforgeeks.org/how-to-implement-linear-interpolation-in-python/.

[22] A. Chauhan, "Introduction to Naïve Bayes Classifier," Towards Data Science, accessed March 12, 2023, https://towardsdatascience.com/introduction-to-naïve-bayes-classifier-fa59e3e24aaf#:~:text=P(Y%3D0)P,)%2C%20so%20y%3D0.&text=In%20statistics%2C%20Laplace%20Smoothing%20is,the%20problem%20of%20zero%20probability.

[23] J. McCaffrey, "Understanding and Implementing a Naïve Bayes Classifier in C#: Listing 1," Visual Studio Magazine, May 1, 2019, https://visualstudiomagazine.com/articles/2019/05/01/naive-bayes-classification.aspx.

[24] N. H. Vu, "N-gram Language Model," Medium, accessed March 12, 2023, https://medium.com/mti-technology/n-gram-language-model-b7c2fc322799.

[25] D. Jurafsky and J. H. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 3rd ed., Stanford University, accessed April 3, 2023, https://web.stanford.edu/~jurafsky/slp3/.