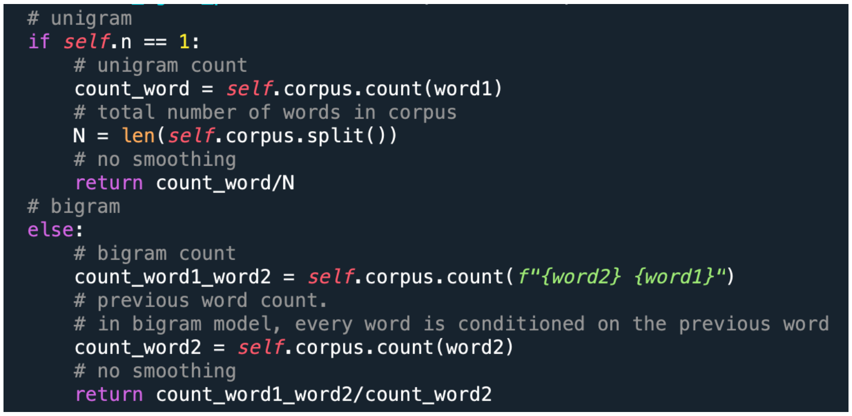
1 Implementation Details

1.1 Unigram and bigram probability computation

The tokens are obtained from the training corpus by using whitespace as the delimiter. Additional preprocessing choices made are converting the corpus to lowercase, removing punctuation, and lemmatizing the words in the corpus. Lowercasing and removing punctuation are two of the most simple and effective preprocessing methods. Lowercasing the text retains the same word and helps with the consistency of n-gram probabilities. For example, “The” and “the” will be treated as the same unigram. Punctuation is removed because it helps clean the dataset by reducing the number of tokens in the corpus by over 10000, and these tokens may not be useful. On the other hand, punctuation marks can be considered as tokens as well since in natural language, punctuation helps with understanding context. For example, “Let ’s eat , grandma” and “Let ’s eat grandma” are the same set of tokens but have different meanings. Lemmatization is a technique used to reduce inflected words to their root word, meaning that it links words with similar meaning to one word. It determines the meaning of a word based on its context by taking into consideration the surrounding sentences or entire document. Due to the effect that punctuation can have, the language models are built with an option to preprocess the dataset or not. The results of this experiment are analyzed in Section 2.



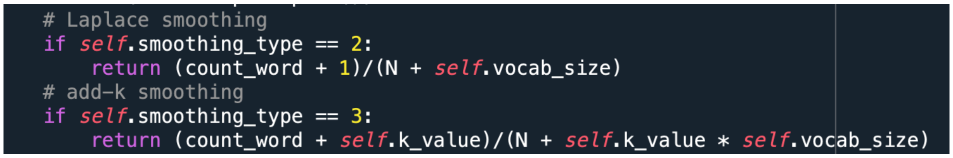
```
# unigram
if self.n == 1:
    # unigram count
    count_word = self.corpus.count(word1)
    # total number of words in corpus
    N = len(self.corpus.split())
    # no smoothing
    return count_word/N
# bigram
else:
    # bigram count
    count_word1_word2 = self.corpus.count(f"{word2} {word1}")
    # previous word count.
    # in bigram model, every word is conditioned on the previous word
    count_word2 = self.corpus.count(word2)
    # no smoothing
    return count_word1_word2/count_word2
```

Figure 1: Implementation of unsmoothed unigram and bigram probability computation.

A unigram is a one-word sequence, and a bigram is a two-word sequence. A unigram language model predicts the probability of a word in a corpus assuming that the occurrence of each word is independent of any previous words. A bigram language model predicts the probability of a word in a corpus based on the previous word. It takes each pair of consecutive words and estimates the probability of encountering a word given the previous word. As shown in Figure 1, the unigram probabilities are computed by getting the count of each word in the training corpus and dividing it by the total number of words in the training corpus. The bigram probabilities are computed by getting the count of each two-word sequence in the training corpus and dividing it by the count of the previous word in that sequence. For example, to calculate the bigram probability of “why not,” the number of times the string “why not” appears in the training corpus is divided by the total number of times the word “why” appears in the same corpus.

A dictionary data structure is used to store the probabilities, and the probabilities are saved in a .json file. For the unigram model, the dictionary stores key value pairs with each word as the key and its unigram probability as the value. For the bigram model, a nested dictionary is used with the outer dictionary storing the previous word as the key and the inner dictionary as the value. The inner dictionary stores each word as the key and its bigram probability of (previous word, word) as the value. In this way, the bigram probability of “word1 word2” can be indexed in the dictionary (say d) as d[word1][word2].

1.2 Smoothing

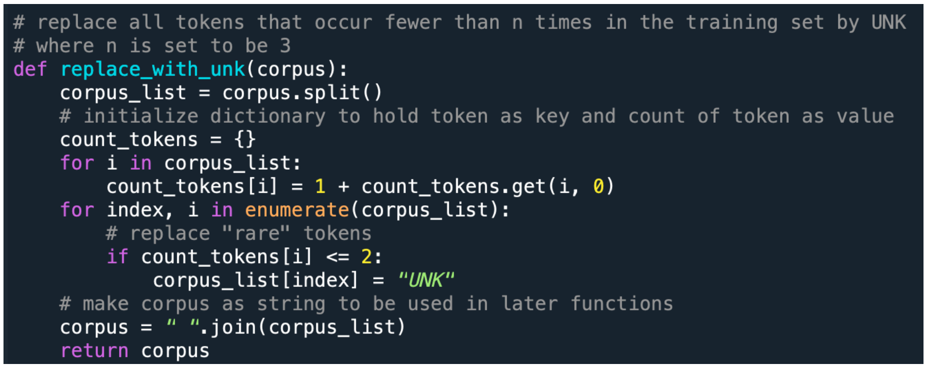


```
    # Laplace smoothing
    if self.smoothing_type == 2:
        return (count_word + 1)/(N + self.vocab_size)
    # add-k smoothing
    if self.smoothing_type == 3:
        return (count_word + self.k_value)/(N + self.k_value * self.vocab_size)
```

Figure 2: Implementation of smoothing unigram probabilities.

The two smoothing methods implemented are Laplace smoothing and add-k smoothing with different k. In Laplace smoothing, 1 is added to all the counts before normalizing them into probabilities. This means that 1 is added in the count of the numerator and the number of words in the vocabulary is added in the count of the denominator, as shown in Figure 2. The vocabulary consists of all unique words in the training set. Adding 1 to the count of each unigram would result in the size of the vocabulary, which is why it is added to the denominator. A more general formulation of Laplace smoothing is add-k smoothing, which adds a fractional count k to all the counts instead of 1. This means that k is added in the count of the numerator and k times the vocabulary size is added in the count of the denominator.

1.3 Unknown word handling



```
# replace all tokens that occur fewer than n times in the training set by UNK
# where n is set to be 3
def replace_with_unk(corpus):
    corpus_list = corpus.split()
    # initialize dictionary to hold token as key and count of token as value
    count_tokens = {}
    for i in corpus_list:
        count_tokens[i] = 1 + count_tokens.get(i, 0)
    for index, i in enumerate(corpus_list):
        # replace "rare" tokens
        if count_tokens[i] <= 2:
            corpus_list[index] = "UNK"
    # make corpus as string to be used in later functions
    corpus = “ ”.join(corpus_list)
    return corpus
```

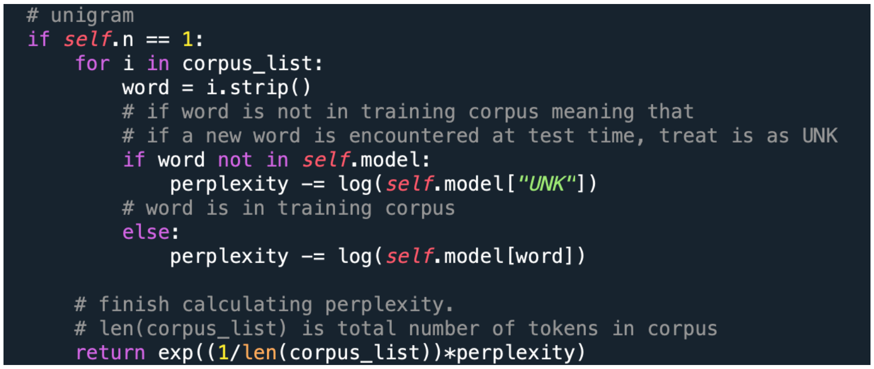
Figure 3: Implementation of handling unknown words.

Since there is no fixed vocabulary given, such a vocabulary is created implicitly by replacing words in the training data by UNK based on their frequency. The method chosen to handle unknown words is to replace all words that occur fewer than 3 times in the training set are by UNK. When a new word is encountered at test time, it is replaced by UNK. The probabilities for UNK are estimated from its counts just like any other word in the training set. As shown in Figure 3, a dictionary is used to store each word as the key and its count as the value. If any key occurs less than 3 times in the training set, that word is replaced by UNK.

For unigram probabilities, the probability of “UNK” is added to the unigram model’s dictionary. For bigram probabilities given a previous word (word1) and a word (word2), there are four cases that need to be considered: probability of “word1 word2,” probability of “word1 UNK,” probability of “UNK word2,” and probability of “UNK UNK.” For each bigram in the training set, these four probabilities are added to the bigram model’s dictionary.

1.4 Implementation of perplexity

The following formula is used to calculate perplexity (PP): $PP = \exp \frac{1}{N}\sum_{i=1}^{N} -\log\ P(w_i)$ for the unigram model and $PP = \exp \frac{1}{N}\sum_{i=1}^{N} -\log\ P(w_i|w_{i-1})$ for the bigram model. Initially, the perplexity variable is set to zero and is created to store the sum of logs of n-gram probability over each word in the dataset.

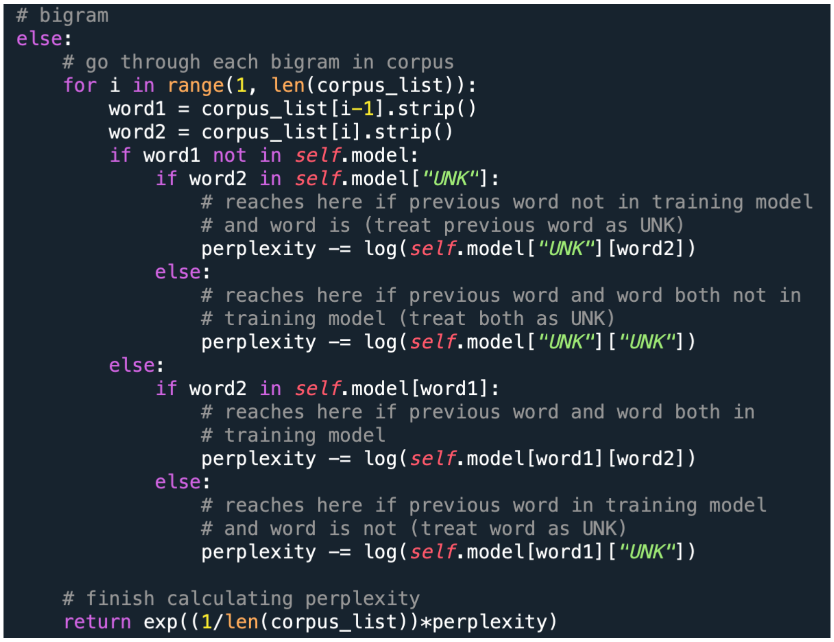


```
# unigram
if self.n == 1:
    for i in corpus_list:
        word = i.strip()
        # if word is not in training corpus meaning that
        # if a new word is encountered at test time, treat is as UNK
        if word not in self.model:
            perplexity -= log(self.model["UNK"])
        # word is in training corpus
        else:
            perplexity -= log(self.model[word])

    # finish calculating perplexity.
    # len(corpus_list) is total number of tokens in corpus
    return exp((1/len(corpus_list))*perplexity)
```

Figure 4: Implementation of unigram model perplexity.

For the unigram model, the negative log of the unigram probability of each word in the dataset is added to the perplexity variable. However, if a word is not in the training corpus, meaning that if a new word is encountered at test time, it is treated as UNK as shown in Figure 4.



```
# bigram
else:
    # go through each bigram in corpus
    for i in range(1, len(corpus_list)):
        word1 = corpus_list[i-1].strip()
        word2 = corpus_list[i].strip()
        if word1 not in self.model:
            if word2 in self.model["UNK"]:
                # reaches here if previous word not in training model
                # and word is (treat previous word as UNK)
                perplexity -= log(self.model["UNK"][word2])
            else:
                # reaches here if previous word and word both not in
                # training model (treat both as UNK)
                perplexity -= log(self.model["UNK"]["UNK"])
        else:
            if word2 in self.model[word1]:
                # reaches here if previous word and word both in
                # training model
                perplexity -= log(self.model[word1][word2])
            else:
                # reaches here if previous word in training model
                # and word is not (treat word as UNK)
                perplexity -= log(self.model[word1]["UNK"])

    # finish calculating perplexity
    return exp((1/len(corpus_list))*perplexity)
```

Figure 5: Implementation of bigram model perplexity.

For the bigram model, as shown in Figure 5, the negative log of the bigram probability of each two-word sequence in the dataset is added to the perplexity variable. However, there are four cases that need to be considered given a previous word (word1) and a word (word2): word1 and word2 are both in the training model, word1 is in the training model and word2 is not (treat word2 as UNK), word1 is not in the training model and word2 is (treat word1 as UNK), and word1 and word2 are both not in the training model (treat both as UNK). The respective bigram probability can then be added to the perplexity variable. Once the log sum has been evaluated, the final perplexity can be calculated as exp(1/N * value in perplexity variable), where N is the total number of tokens in the dataset.

2 Eval, Analysis and Findings

If an n-gram in the validation set has 0 probability (unseen word in the training set), perplexity cannot be computed. To avoid assigning 0 probability to an n-gram in the validation set, handling unknown words and smoothing is required. Smoothing works by decreasing the probability of n-grams seen so that there is a little bit of probability mass left over for unseen n-grams. UNK affects perplexity for the following reason: perplexity captures the effective vocabulary size under the model so it should only be compared across models with the same vocabularies. Since new words in the validation set that have not been encountered in the training set are replaced by UNK, the vocabulary size is kept fixed, and perplexity can be compared across unigram and bigram models.

Run	Preprocessing	Smoothing Type	Smoothing Parameter (k)	Training Set Perplexity	Validation Set Perplexity
1	No	No Smoothing	N/A	217.333	193.448
2	No	Laplace	1	218.428	194.948
3	No	Add-k	0.5	217.817	194.153
4	No	Add-k	0.1	217.418	193.58
5	No	Add-k	0.01	217.342	193.461
6	Yes	No Smoothing	N/A	191.187	176.8
7	Yes	Laplace	1	192.188	178.125
8	Yes	Add-k	0.5	191.636	177.423
9	Yes	Add-k	0.1	191.267	176.917
10	Yes	Add-k	0.01	191.195	176.812

Table 1: Perplexity results for unigram models.

Run	Preprocessing	Smoothing Type	Smoothing Parameter (k)	Training Set Perplexity	Validation Set Perplexity
1	No	Laplace	1	295.283	239.868
2	No	Add-k	0.5	207.795	172.008
3	No	Add-k	0.1	101.118	91.836
4	No	Add-k	0.01	53.401	57.821
5	Yes	Laplace	1	324.163	270.991
6	Yes	Add-k	0.5	232.266	197.459
7	Yes	Add-k	0.1	116.002	107.961
8	Yes	Add-k	0.01	62.936	70.224

Table 2: Perplexity results for bigram models.

Two extra runs for unigram models were conducted for the unsmoothed dataset. Because of UNK handling, no word in the validation set will have zero probability. For bigrams, even with UNK handling, there might not be probabilities for all word2 in the validation set where word1 is UNK given a previous word (word1) and a word (word2), which is why smoothing is required.

After no preprocessing and handling unknown words, the total number of tokens in the training corpus is 89684 and the number of unique tokens in the corpus, or the size of the vocabulary, is 2427. After preprocessing and handling unknown words, the total number of tokens in the training corpus is 79399 and the number of unique tokens in the corpus, or the size of the vocabulary, is 2109.

The perplexity results are shown in Table 1 and Table 2. The smoothing strategy has a slight effect on the unigram model perplexity of the validation set. As the k-value decreases, the perplexity slightly decreases. For bigrams, there is a noticeable decrease in perplexity as the k-value is decreased. When computing the n-gram probabilities of the most frequent unigrams and bigrams, a unigram typically has a higher count in the numerator so adding k does not have much effect on the unigram count. However, for a bigram, the count in the numerator is lower (since there are less occurrences of a two-word sequence than a one-word sequence) so adding k will have a larger effect on the bigram count. Table 2 shows that the bigram model perplexity did change as k changed while in Table 1, the unigram model perplexity only slightly did.

Perplexity measures the degree of uncertainty of a language model in the predictions it makes about new tokens. Typically, the lower the perplexity, the better the model. The reasoning behind this is that a bigram has more context to predict the next word. A bigram has a better idea of what the following word might be, resulting in a lower perplexity. For no preprocessing done on the data, bigram model perplexity is less than unigram model perplexity at k=0.5. For preprocessed data, bigram model perplexity is less than unigram model perplexity at k=0.1.

An additional experiment was performed to determine the effect of preprocessing the corpus. For unigram models, preprocessing resulted in lower perplexity while for bigram models, no preprocessing resulted in lower perplexity. Unigrams are independent of one another so preprocessing made the model simpler with less unigrams to account for. Different bigrams would be formed by choosing to preprocess or not, and the lower perplexities show that upper/lower case of words, punctuation, and inflected forms (tense/plurality) of words are important in understanding the context for predicting the next word. However, the perplexities for each smoothing case comparing no preprocessing and preprocessing are not too different, so preprocessing is not vital to building the language models but not preprocessing can help for bigrams.