# Assignment 1

https://github.com/adibarra/hlt

Group 14

Annette Llanas ajl200006

Ashlee Kang ajk200003

Syed Kabir snk210004

Alec Ibarra adi220000

## 1 Implementation Details

### 1.1 Unigram and Bigram Probability Computation

#### Unigram Probability Computation

The unigram model estimates the probability of each individual word appearing from its training corpus. Unigram models do not have any context and will generate words independently of each other, without taking into account their context or meaning in a sentence.

Unigram models generate the probabilities for each individual word using the Maximum Likelihood Estimation (MLE) formula in Figure 1.

$$\boxed{P(w) = \frac{C(w)}{N}}$$

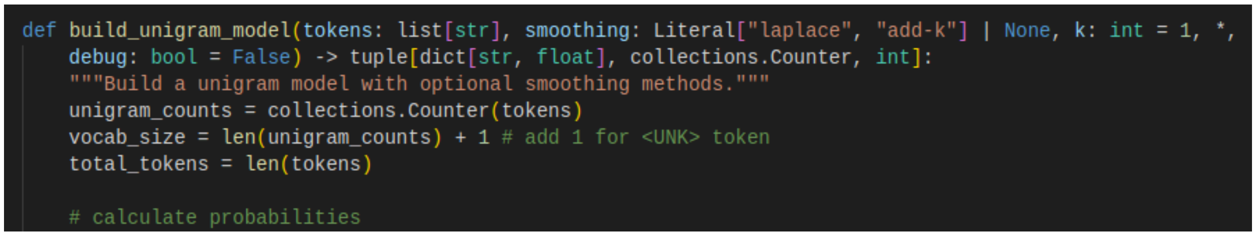
***Figure 1.*** *Maximum Likelihood Estimation Equation.*

In this equation:

**P(*w*)** is the probability of token ***w*** appearing in the corpus.
**C(*w*)** is the number of occurrences of token ***w*** in the corpus.
**N** is the total number of tokens in the corpus.

This essentially means that the calculated probability of each word is just the word's relative frequency in the word in the dataset.

In our implementation of the code in Figure 2, we first pre-process the dataset by tokenizing it, counting the number of occurrences for each token, and then normalizing the token counts to calculate the probabilities.



```
def build_unigram_model(tokens: list[str], smoothing: Literal["laplace", "add-k"] | None, k: int = 1, *,
    debug: bool = False) -> tuple[dict[str, float], collections.Counter, int]:
    """Build a unigram model with optional smoothing methods."""
    unigram_counts = collections.Counter(tokens)
    vocab_size = len(unigram_counts) + 1 # add 1 for <UNK> token
    total_tokens = len(tokens)

    # calculate probabilities
```

***Figure 2.*** *Part of the code implementation for our unigram model.*

## Bigram Probability Computation

The bigram model estimates the probability of every two consecutive words appearing from its training corpus. Unlike the unigram model which generates words independently of each other, the bigram model takes into account the context of a pair of words. Additionally, the bigram model takes into account the start and end of sentences, represented by the tokens **<s>** and **</s>** respectively.

Bigram models generate the probability of a word(Wn) given a previous word(Wn-1) using the Maximum Likelihood Estimation (MLE)

$$\boxed{P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}}$$

***Figure 3.*** *Maximum Likelihood Estimation Equation.*

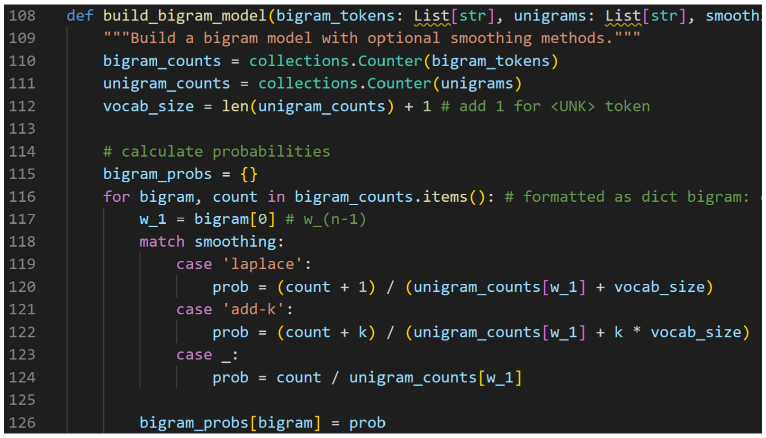
In this equation:

**P(*wn*|*wn*-1)** is the probability of the token ***wn*** appearing given the previous token is wn-1 in the corpus.

**C(*wn*-1*wn*)** is the count of bigram **(*wn*-1*wn*)** in the corpus.

**C(*wn*-1)** is the number of occurrences of ***wn*-1** in the corpus.

This formula essentially calculates the probability of a word occurring given a previous word by getting the frequency of the bigram and normalizing it by the count of the previous word.



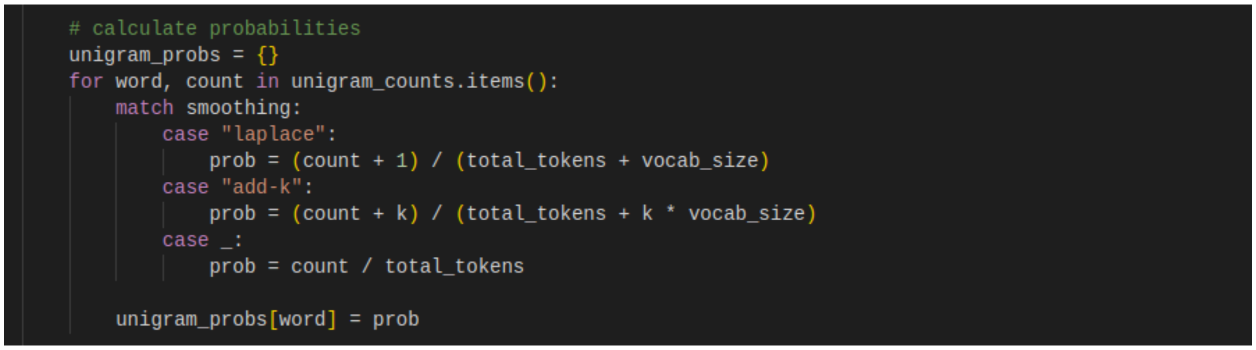
```python
def build_bigram_model(bigram_tokens: List[str], unigrams: List[str], smooth
    """Build a bigram model with optional smoothing methods."""
    bigram_counts = collections.Counter(bigram_tokens)
    unigram_counts = collections.Counter(unigrams)
    vocab_size = len(unigram_counts) + 1 # add 1 for <UNK> token

    # calculate probabilities
    bigram_probs = {}
    for bigram, count in bigram_counts.items(): # formatted as dict bigram:
        w_1 = bigram[0] # w_(n-1)
        match smoothing:
            case 'laplace':
                prob = (count + 1) / (unigram_counts[w_1] + vocab_size)
            case 'add-k':
                prob = (count + k) / (unigram_counts[w_1] + k * vocab_size)
            case _:
                prob = count / unigram_counts[w_1]

        bigram_probs[bigram] = prob
```

***Figure 4*** *Part of code implementation for bigram model.*

## 1.2 Smoothing

In our models, we incorporate several smoothing methods to ensure robustness, especially when handling unseen words. Smoothing addresses the issue of zero probabilities for words not encountered during training, which would otherwise make the model fail when it encounters these words in validation data. Without smoothing, unseen words would receive a probability of zero, making it impossible to calculate meaningful probabilities or perplexities.

We implement three common smoothing techniques: **Laplace Smoothing**, **Add-K Smoothing,** and **Unsmoothed**.



```python
    # calculate probabilities
    unigram_probs = {}
    for word, count in unigram_counts.items():
        match smoothing:
            case "laplace":
                prob = (count + 1) / (total_tokens + vocab_size)
            case "add-k":
                prob = (count + k) / (total_tokens + k * vocab_size)
            case _:
                prob = count / total_tokens

        unigram_probs[word] = prob
```

***Figure 5.*** *Part of the code implementation for unigram probability smoothing.*

### Laplace (Add-One) Smoothing

Laplace smoothing, or Add-One smoothing, is one of the simplest methods. It works by adding 1 to the count of every word in the vocabulary before calculating probabilities. This ensures that no word has a zero-probability, which is essential for handling unseen words. However, while this method prevents zero-probabilities, it tends to overestimate the probability of rare words because it treats all unseen words as equally likely, regardless of their actual frequency.

### Add-k Smoothing

Add-k smoothing extends the idea of Laplace smoothing by allowing a small constant k to be added instead of 1. This offers more control over the smoothing process and can help mitigate the overcorrection issue seen in Laplace smoothing, particularly for large vocabularies. By adjusting k, we can fine-tune how much influence the smoothing has on the probability distribution, leading to a more balanced approach in handling rare and unseen words.

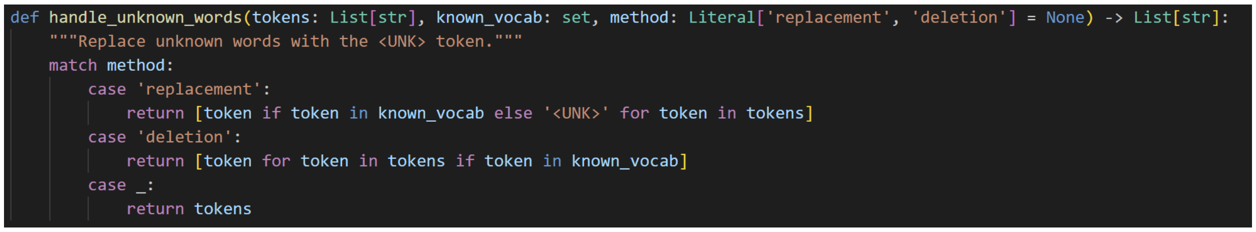
### Unsmoothed

Unsmoothed applies the raw MLE equation without any corrections to the probabilities. As a result, it assigns a probability of zero to any word not present in the training data. This is often problematic because a single unseen word in a sentence will cause the entire probability to collapse to zero, making perplexity undefined. For this reason, Laplace Smoothing or Add-k Smoothing is often preferred.

## 1.3 Unknown word handling

For our unigram and bigram models, we incorporate three methods for handling unknown n-grams. These methods address the issues our models will face when encountering unseen words which would cause it to fail as they would lead to zero probabilities. When we encounter words that were not a part of the training corpus, we utilize these handling methods to ensure our models continue to function properly.

We implement three handling techniques: **Replacement, Deletion,** and **Retain**.



```python
def handle_unknown_words(tokens: List[str], known_vocab: set, method: Literal['replacement', 'deletion'] = None) -> List[str]:
    """Replace unknown words with the <UNK> token."""
    match method:
        case 'replacement':
            return [token if token in known_vocab else '<UNK>' for token in tokens]
        case 'deletion':
            return [token for token in tokens if token in known_vocab]
        case _:
            return tokens
```

***Figure 6.*** *Code for the implementation of handling unknown words*

### Replacement

The replacement method replaces any n-gram not found in the known vocabulary with the token **'<UNK>'**. This allows the model to be able to handle every word in the validation set even if they haven't been seen in the training phase, and assigns them a probability based on the smoothing method. Ensuring zero probabilities don't arise and no words are discarded as all tokens contribute to the model's overall probability.

### Deletion

The deletion method removes any word not present in the known vocabulary. By removing the unknown words, the models only process the tokens seen in its training corpus and do not assign any probabilities to the unseen words. However, this method has its drawbacks as while it ensures the model does not encounter these unseen words, it leads to a loss of information and context if these words were particularly important.

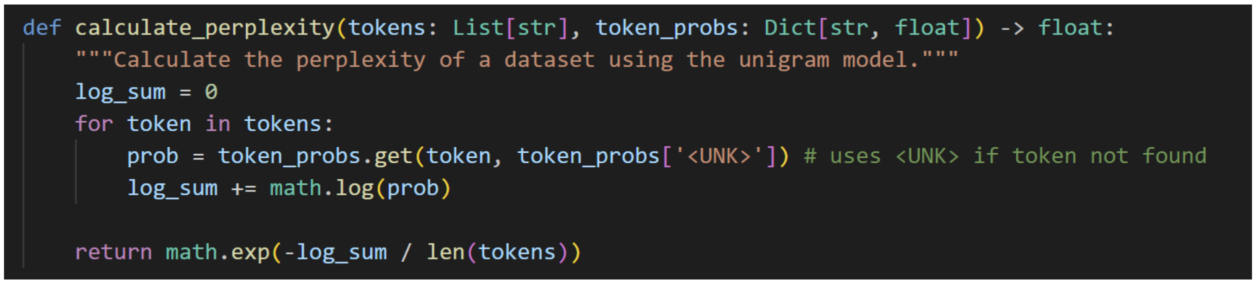
### Retain

The last method's way of handling unknown words is simply to return the list of tokens which will contain the unknown words. Essentially a leave as is approach.

### 1.4 Implementation of perplexity

Perplexity is an evaluation metric to tell us how well our model performs on the test set, or how confused our model is. It's calculated as the inverse probability of the test set, normalized by the number of words. A lower perplexity score is better. As we have a bigram model and unigram model, our perplexity calculations would be different as respectively, one calculates probabilities based on word pairs and the other on individual words.

We implemented this perplexity calculation function for the the two models:



```python
def calculate_perplexity(tokens: List[str], token_probs: Dict[str, float]) -> float:
    """Calculate the perplexity of a dataset using the unigram model."""
    log_sum = 0
    for token in tokens:
        prob = token_probs.get(token, token_probs['<UNK>']) # uses <UNK> if token not found
        log_sum += math.log(prob)

    return math.exp(-log_sum / len(tokens))
```

***Figure 7*** *Code for computing Perplexity Score*

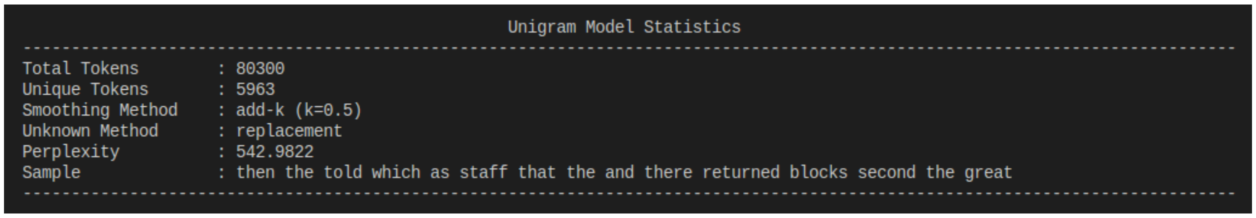
In Figure 7, the **calculate_perplexity** function loops through all the tokens in the dataset and retrieves each token's probability from the unigram model. If the token is an unseen word, we use the **'<UNK>'** token. For calculating the perplexity of our unigram model, we compute the log of the probability of each word generated by the unigram model. This is done to prevent underflow, as calculated probabilities can become very small. Then the average log probability is taken and the negative exponent of that average gives us our perplexity score. We can do the same for the bigram model by simply substituting unigram probabilities with bigram probabilities.

## 2 Eval, Analysis and Findings

For our analysis, we decided to compare the language models using Add-k smoothing with k=0.5, as it provided a balanced approach to handling unseen words. This value was chosen because it effectively reduced perplexity while avoiding the over-smoothing issues observed with higher k values. By maintaining a reasonable probability distribution, k=0.5 allowed for a more accurate evaluation of the unigram and bigram models.

## 2.1 Unigram Findings

After running the unigram language model, we get the following output statistics:



```
                                        Unigram Model Statistics
-----------------------------------------------------------------------------------------------------------------------------
Total Tokens        : 80300
Unique Tokens       : 5963
Smoothing Method    : add-k (k=0.5)
Unknown Method      : replacement
Perplexity          : 542.9822
Sample              : then the told which as staff that the and there returned blocks second the great
-----------------------------------------------------------------------------------------------------------------------------
```

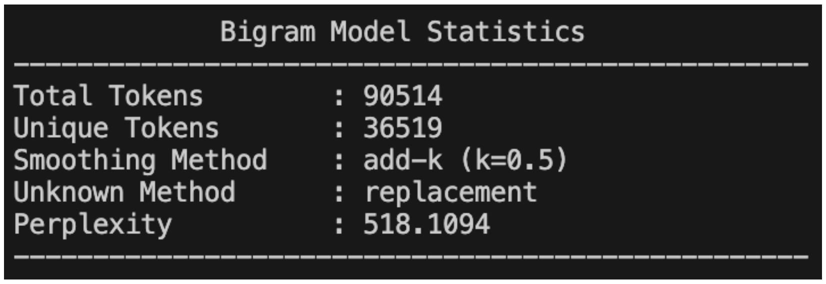
***Figure 8.*** *Unigram Language Model Statistics*

From Figure 8, the unigram language model produced a perplexity result of 542.9822, which is relatively high. This indicates that the model struggles to generate fluent and coherent sequences. However, this result is expected because the unigram model treats words as independent entities, disregarding any contextual relationships.

Additionally, the output includes a smoothing method section. In this case, we applied Add-k smoothing with k = 0.5. This technique helps prevent zero probabilities for unseen words, ensuring that every word in the test set has some assigned probability. The smoothing method balances rare and frequent words, improving robustness. However, while smoothing prevents specific numerical issues, it does not address the fundamental weakness of the unigram model, which is the lack of word dependencies.

## 2.2 Bigram Findings

After running the Bigram language model, we get the following output statistics:



```
             Bigram Model Statistics
--------------------------------------------------
Total Tokens        : 90514
Unique Tokens       : 36519
Smoothing Method    : add-k (k=0.5)
Unknown Method      : replacement
Perplexity          : 518.1094
--------------------------------------------------
```

***Figure 9.*** *Bigram Language Model Statistics*

From Figure 9, the bigram language model produced a perplexity result of 518.1094, which is lower than the unigram model's perplexity of 542.9822. This result confirms that the bigram model performs better at predicting sequences by incorporating context from previous words. The reduction in perplexity suggests that context helps improve prediction accuracy, but the bigram model may still be limited in handling more complex language structures where longer-range dependencies are essential.

Additionally, the output includes a smoothing method section. In this case, we applied Add-k smoothing with k = 0.5. This smoothing method ensures that all bigrams, including unseen word pairs, receive a non-zero probability, preventing model failures due to missing data.

### 2.3 Conclusion

When it comes to choosing a language model, the results show that the bigram model performs better than the unigram model, with a lower perplexity score (518.1094 vs 542.9822). This confirms that incorporating context from previous words improves prediction accuracy.

However, the bigram model still has limitations, particularly with data sparsity due to the large number of unique tokens. While add-k smoothing helped prevent zero probabilities, the model still struggles with longer-range dependencies. Overall, while bigrams outperform unigrams, they are still limited in capturing complex language structures.

## 3 Others

### Details of programming library usage, if any

Standard python library only

### Brief description of the contributions of each group member

Alec - Unigram code, Unigram implementation section, Smoothing section, Others section
Annette - Eval, Analysis and Findings
Ashlee - Bigram code
Syed - Bigram Probability Computation section, Unknown Word handling section, Implementation of Perplexity section

### Feedback for the project

The project was very beneficial in reinforcing the concepts covered in the lectures. It provided us with some valuable hands-on experience with language modeling. One suggestion for improvement could be providing some baseline statistics for the datasets to ensure correct implementation.