Assignment 1

https://github.com/axr2718/nlp-code/tree/main/A1

Group 20

Adrian Rodriguez	Vishwa Pandian	Harsha Addagada	Danish Humair
AXR190042	VRP200001	HXA190029	DAL677646

1 Implementation Details

1.1 Unigram and Bigram Probability Computation

Before we computed the probabilities, we decided on using Byte-Pair Encoding (BPE) or whitespace separation between the words to create our tokens. BPE counts the frequencies of subwords and creates tokens from that while whitespace takes the words separated by whitespace characters, such as space, tabs, and new line characters. This would allow us to compare the performance between word and subword tokens.

For the Unigram model, the probability computation of a 1-gram was how many times that token was encountered in the dataset divided by the number of tokens in the dataset.

$$P(w) = \frac{\text{count}(w)}{\sum_{v \in V} \text{count}(v)}$$

where w is the token in question, and v are all the tokens in the vocabulary. In code, we wrote it as:

```python
def unigram_prob(token: str, unigram_counts: Counter, vocab_size: int, alpha: float) -> float:
    """Calculate smoothed unigram probability with additive smoothing."""
    token_count = unigram_counts.get(token, 0)
    total_tokens = sum(unigram_counts.values())
    return (token_count + alpha) / (total_tokens + alpha * vocab_size)
```

Figure 1: Unigram Probability Python Code

For the Bigram model, the probability computation of a 2-gram was how many times the 2-gram token appeared in the dataset divided by the all 2-grams that started with the conditioned token.

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\sum_{v \in V} \text{count}(w_{i-1}, v)}$$

where w_i is the 2nd gram, w_{i-1} is the previous gram, and v are all the tokens in the vocabulary conditioned on the previous gram. The count function returns the counts of the arguments in order of the arguments. In code, we wrote it as:

```python
def bigram_prob(h: str, w: str, bigram_counts: Counter, context_counts: Counter, vocab_size: int, alpha: float) -> float:
    # Additive smoothing
    count_hw = bigram_counts.get((h, w), 0)
    count_h = context_counts.get(h, 0)
    return (count_hw + alpha) / (count_h + alpha * vocab_size)
```

Figure 2: Bigram Probability Python Code

1.2 Smoothing

We implemented both Laplace smoothing and Add-K smoothing for the probability computation of both models. For the Unigram model, the equation becomes

$$P(w) = \frac{\text{count}(w) + \alpha}{\sum_{v \in V} \text{count}(v) + \alpha|V|}$$

For the Bigram model, the equation becomes

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i) + \alpha}{\sum_{v \in V} \text{count}(w_{i-1}, v) + \alpha|V|}$$

where $\alpha \in \mathbb{R}^+ \cup \{0\}$ and $|V|$ is the total amount of tokens in the vocabulary. When $\alpha = 0$, we get no smoothing and compute the unsmoothed probabilities. When $\alpha = 1$, we get Laplace smoothing, and when α is equal to anything else, we get Add-K smoothing. Figure 1 and Figure 2 allows for any of these cases by specifying the alpha parameter in the code.

1.3 Unknown Word Handling

We introduce the token <UNK> for unknown words for tokens that contain less than a certain threshold or tokens that appear in the validation set that do not appear in the training set. For the former, when we create the tokens for training, we can assign <UNK> to tokens that appear less than a certain threshold if we decided to, but if it is 0, then nothing is set to unknown. During testing, we check the tokens in the validation set and if any are not in the training dataset vocabulary, we set them to unknown. In code:

```
if is_train:
    token_freq = Counter()
    for tokens in linestokens:
        token_freq.update(tokens)
    vocab_set = set(tok for tok, freq in token_freq.items() if freq > threshold)
    vocab_set.add('<UNK>')

    replaced_lines = []
    for tokens in linestokens:
        new_tokens = [tok if tok in vocab_set else '<UNK>' for tok in tokens]
        replaced_lines.append(new_tokens)

    return replaced_lines, sorted(vocab_set)
else:
    if vocab is None:
        raise ValueError("Vocabulary must be provided for evaluation mode")
    vocab_set = set(vocab)
    replaced_lines = []
    for tokens in linestokens:
        new_tokens = [tok if tok in vocab_set else '<UNK>' for tok in tokens]
        replaced_lines.append(new_tokens)
    return replaced_lines, vocab
```

Figure 3: Unknown Word Handling Code in Python

1.4 Implementation of Perplexity

Perplexity was computed on the validation set and was defined by the average of the sum of the negative log probability and then exponentiated.

$$\exp(-\frac{1}{|V|}\sum_{i=1}^{|V|}\log P(\cdot))$$

In code, we simply iterated through the validation set tokens using the Unigram and Bigram models, computed their probabilities, and then applied the perplexity equation.

2 Eval, Analysis, and Findings

In our evaluation, we tested our models on the validation set, using both whitespace and BPE tokenization, different K values for K-Add smoothing, and different <UNK> thresholds, with perplexity being the key metric. Our full results are in this Notebook in the repository.

2.1 Unigram vs. Bigram

We evaluated both the unigram and bigram models under the same conditions (smoothing value and tokenization used). In the case of unigrams, there was no unknown threshold applied as rare words are already handled reasonably. We note that for whitespace, the perplexity of both models is similar, since the vocabulary explores with every single token separated by a whitespace character. For BPE, however, we see that the perplexity quickly drops in the bigram model compared to the unigram model. This shows us that bigram models are better models when tokenization is better, as expected. See figure 4.

2.2 BPE vs Whitespace Tokenization

Between BPE and whitespace tokenization, when keeping the values for smoothing and unknown word threshold clamped at the same values for both methods, we observe that BPE had a much lower perplexity than whitespace. At the highest perplexity, whitespace had 43930 and BPE had 876. At the lowest perplexity, whitespace had 227 and BPE had 120.

This is due to the fact that every sample is a review, which means that people will write it differently, and there will be many different words, which gives higher perplexity per vocabulary since there are more tokens in the vocabulary. Additionally, every word, whether misspelled or rare, is part of the training vocabulary, which may not appear in the validation set.

Unlike whitespace, BPE captures the frequencies of subtokens and can handle rare tokens, which correspond to the different ways people write, and also captures prefixes and suffixes, which whitespace does not. This means that the vocabulary is smaller, leading to a smaller perplexity.

2.3 Smoothing and Unknown Thresholds

We observed that increasing the smoothing parameter, while keeping the other parameters clamped to the same values, would bring perplexity down and then bring it up as it increased. We also note that BPE was impacted less by the choice of the unknown threshold. This is because whitespace can remove words from the vocabulary that were not in the validation set. For this, we used unknown thresholds values of 0, 1, and 2. The smoothing values were 0, 0.5, and 1.0. We note that decreasing the vocabulary and removing rare tokens in whitespace tokenization lowered the perplexity for reasons stated above.

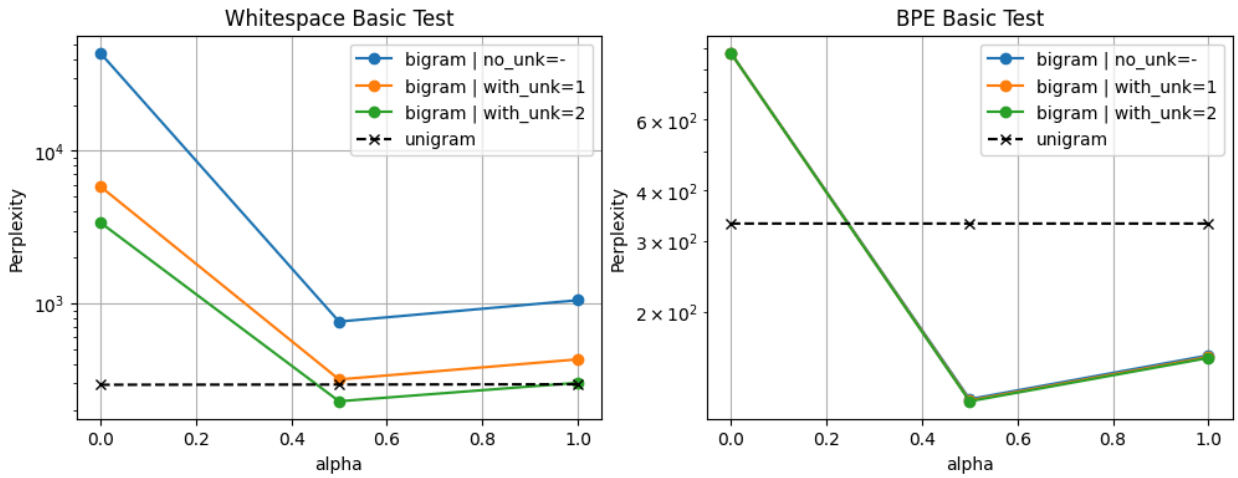

Figure 4: BPE is less impacted by removing words from the vocabulary

For a more fine-grained analysis and evaluation, we decided to vary the smoothing parameter between 0.1 and 0.8 to find the best parameters and kept the unknown threshold at 2 due to having the better results. The trend continued in that higher smoothing values caused the perplexity to rise.

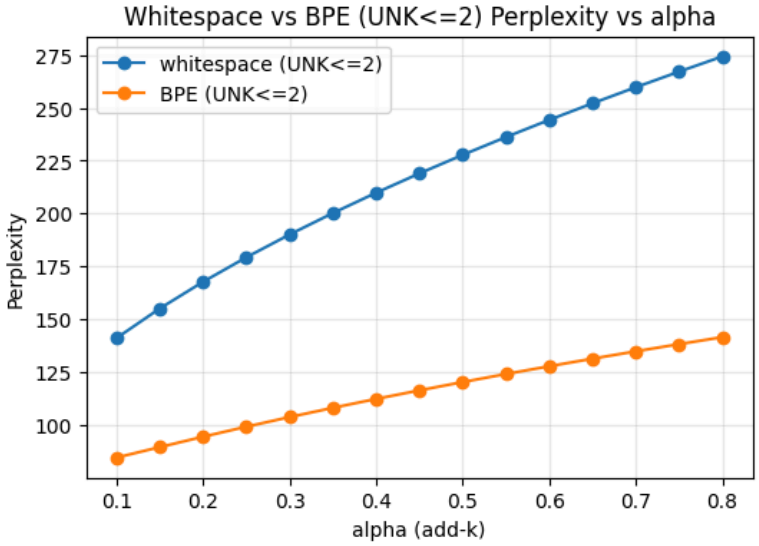

Figure 5: Increasing the smoothing parameter increases perplexity.

The perplexity is monotonically increasing, which tells us that the best values for smoothing lie between 0 and 0.1. We further tested this range and found that the best values for whitespace were unknown threshold of 2 and smoothing value of 0.01. For BPE, the threshold value was 2 and the smoothing value was 0.04.

To validate these choices, we also computed perplexities on both the train and validation set using the best settings. For whitespace tokenization, the optimal parameters resulted in a training perplexity of 43.9 and validation of 115.5. For BPE, it was 46.8 for the training set and 80.1 for the validation set.

3 Contributions

Adrian Rodriguez: Data preprocessing, whitespace tokenization, probability distribution normalization computation for whitespace tokenization, and wrote the report.

Vishwa Pandian: Implemented bigram model (counts, add-k smoothing, perplexity) and extended the system with BPE tokenization and an interactive REPL for text generation.

Danish Humair: Tweaked bigram model implementation, ran evaluations and conducted analysis, tuned hyperparameters, generated visualizations.

Harsha Addagada: Implemented unigram model, probability calculation, and perplexity evaluation. I also added functionality to handle rare and out-of-vocabulary words.

4 Feedback

Adrian Rodriguez: It's a really fun toy project, but it's too simple to capture the essence of NLP. It comes down to computing probabilities of the data. I would have preferred computing n-gram models rather than unigram and bigram models. There was also no requirement for generating text, which I would have enjoyed.

Vishwa Pandian: I'm happy to get hands-on practice to complement the theory learned in class.

Danish Humair: I found this assignment to be a good balance between coding and open-ended analysis. It was fascinating to see ourselves how small design choices had such a large impact on perplexity.

Harsha Addagada: I enjoyed implementing the models, as the hands-on practice helped solidify the theory; would have also liked to apply them to a simple downstream task to see the real-world impact.