CS6320 Assignment 1

https://github.com/hipstermartin/CS6320_F23/tree/main/Assignment1

Group 9 Abhinav Yalamaddi YXA210040 Satwik Arvapalli SXA220012

1 Implementation Details

1.1 Data Input and Preprocessing

1.1.1 Description

We began by reading reviews from the dataset. These reviews were then preprocessed to make them suitable for our language modeling task. After preprocessing, we analyzed the distribution of word frequencies across the dataset and observed that the average word frequency was approximately 15.15.

1.1.2 Methodology

How we did it:

- Converted all text to lowercase to maintain uniformity.
- Replaced numbers with a placeholder `<NUM>` to generalize numeric data.
- Managed punctuations by ensuring they were treated as separate entities.
- Introduced custom tokens like `<REVIEW_START>` and `<REVIEW_END>` to signify the beginning and end of reviews.
- Analyzed the word frequencies and set a *min_freq* threshold of **15.5**, given that it was the average word frequency. This decision ensured that words with frequencies below this threshold were treated as less common and replaced with the `<UNK>` token.

Figure 1: Code Snippet - Data Preprocessing

```
import re
def preprocess_text(text):
  text = text.lower()
  text = re.sub(r'\d+', '<NUM>', text)
  text = re.sub(r'([.,!?;])', r' \1 ', text)
  text = re.sub(r'http\S+', '<URL>', text)
  text = re.sub(r'\S+@\S+', '<EMAIL>', text)
  tokens = text.split()
  tokens = ['<REVIEW_START>'] + tokens + ['<REVIEW_END>']
  return tokens

def preprocess_file(file_path):
  with open(file_path, 'r') as file:
      content = file.readlines()

  preprocessed_reviews = [preprocess_text(review) for review in content]

  all_tokens = [token for review in preprocessed_reviews for token in review]
  token_freq = Counter(all_tokens)
  avg_word_freq = sum(token_freq.values()) / len(token_freq)
  print(avg_word_freq)

  return preprocessed_reviews
```

Listing 1: Input Data Preprocessing Function(s)

1.2 Unigram and Bigram Probability Computation

1.2.1 Description

To model the language structure in our reviews, we computed probabilities for unigrams and bigrams. These probabilities represent the likelihood of individual words and word pairs, respectively, appearing in our dataset.

1.2.2 Methodology

- For unigrams: The probability $P(w_i)$ of a word w_i was calculated as the ratio of the count of w_i to the total number of tokens.
- For bigrams: The probability $P(w_i|w_{i-1})$ of a word w_i given the previous word w_{i-1} was calculated as the ratio of the count of the bigram (w_{i-1}, w_i) to the count of w_{i-1}.

Figure 2: Code Snippet - N-gram computation

```
def compute_unigram_freq(tokens, min_freq=15.5):
  unigram_freq = {}
  for review in tokens:
      for token in review:
          unigram_freq[token] = unigram_freq.get(token, 0) + 1
  # keeping only words with frequency >= min_freq
  trimmed_unigram_freq = {word: freq for word, freq in unigram_freq.items()
                          if freq >= min_freq}
  return trimmed_unigram_freq

def compute_bigram_freq(tokens, unigram_freq):
  bigram_freq = {}
  for review in tokens:
      for i in range(len(review) - 1):
          bigram = (review[i], review[i + 1])
          # Only include the bigram if both words are in the unigram
          if bigram[0] in unigram_freq and bigram[1] in unigram_freq:
              bigram_freq[bigram] = bigram_freq.get(bigram, 0) + 1
  return bigram_freq
```

Listing 2: Unigram and Bigram Functions

1.3 Smoothing

1.3.1 Description

To handle the zero probabilities that arise from unseen bigrams in the validation set, we applied smoothing techniques. This ensures that our model can handle previously unseen word combinations without assigning them a zero probability.

1.3.2 Methodology

- **Laplace Smoothing**: Every unigram and bigram count was incremented by 1, and the probabilities were recalculated.
- **Add-k Smoothing**: Similar to Laplace but with a tunable parameter k. We experimented with various k values to find the optimal setting.

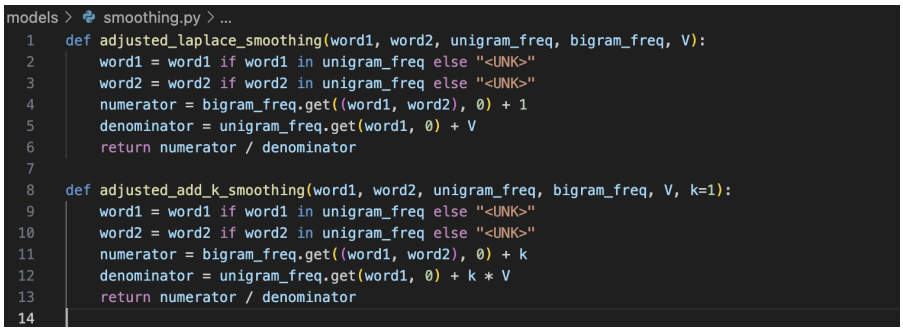

Figure 3: Code Snippet - Smoothing Techniques

```python
def adjusted_laplace_smoothing(word1, word2, unigram_freq, bigram_freq, V):
    word1 = word1 if word1 in unigram_freq else "<UNK>"
    word2 = word2 if word2 in unigram_freq else "<UNK>"
    numerator = bigram_freq.get((word1, word2), 0) + 1
    denominator = unigram_freq.get(word1, 0) + V
    return numerator / denominator

def adjusted_add_k_smoothing(word1, word2, unigram_freq, bigram_freq, V, k=1):
    word1 = word1 if word1 in unigram_freq else "<UNK>"
    word2 = word2 if word2 in unigram_freq else "<UNK>"
    numerator = bigram_freq.get((word1, word2), 0) + k
    denominator = unigram_freq.get(word1, 0) + k * V
    return numerator / denominator
```

Listing 3: Laplace and Add-k Smoothing Implementations

1.4 Unknown Word Handling

1.4.1 Description

Words that didn't meet our frequency threshold or were not seen during the training phase were considered unknown. This approach ensures that our model can generalize better to unseen data.

1.4.2 Methodology

- Words that didn't meet our *min_freq* threshold were replaced with a placeholder token `<UNK>` during preprocessing.
- During model evaluation, any word not seen in the training phase was also replaced with the `<UNK>` token.

Figure 4: Code Snippet - Unseen words

```
def handle_unseen_words(tokens, vocab):
    return [[token if token in vocab else "<UNK>" for token in review]
                              for review in tokens]
```

Listing 4: Unknown word handling Function

1.5 Perplexity Computation

1.5.1 Description

Perplexity is a metric used to evaluate the performance of language models. A lower perplexity indicates a better fit of the model to the data. In the context of our assignment, we use perplexity to gauge how well our bigram language model predicts the sequence of words in the validation set.

1.5.2 Methodology

The formula for perplexity, as provided in the assignment brief, is given by:

$$PP = \left(\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i1}, ..., w_{in+1})} \right)^{\frac{1}{N}}$$

Which can also be expressed in terms of the average negative log-likelihood as:

$$PP = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log P(w_i|w_{i1}, ..., w_{in+1}) \right)$$

In our implementation, we focused on a bigram model, where the probability of a word w_i is dependent only on the immediately preceding word w_{i-1}. Thus, our perplexity formula becomes:

$$PP = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log P(w_i|w_{i-1}) \right)$$

1.5.3 Insights

By using this formula, we can directly measure how well our bigram model is predicting the validation data. The goal is to minimize the perplexity. Through the application of various techniques, including smoothing and unknown word handling, we aim to optimize the model and achieve a lower perplexity score.

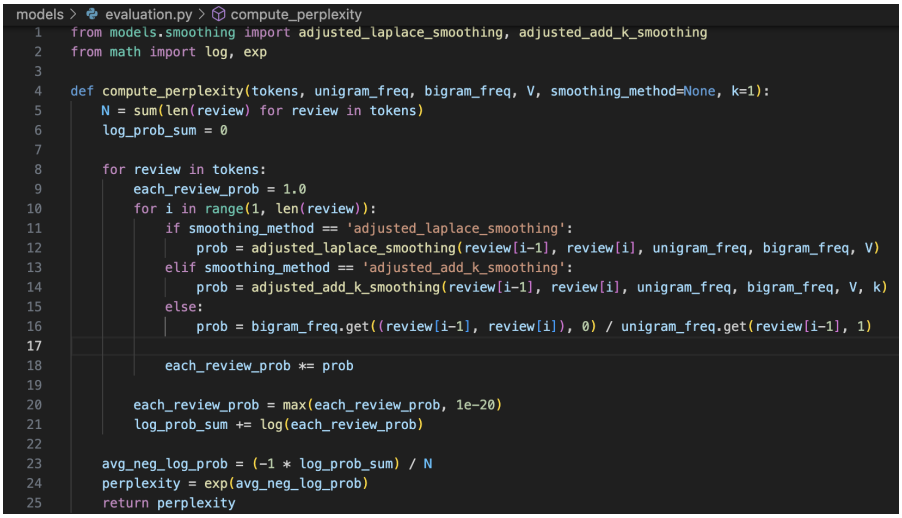

Figure 5: Code Snippet - Perplexity

```python
def compute_perplexity(tokens, unigram_freq, bigram_freq, V, smoothing_method=None, k=1):
    N = sum(len(review) for review in tokens)
    log_prob_sum = 0

    for review in tokens:
        each_review_prob = 1.0
        for i in range(1, len(review)):
            if smoothing_method == 'adjusted_laplace_smoothing':
                prob = adjusted_laplace_smoothing(review[i-1], review[i],
                                    unigram_freq, bigram_freq, V)
            elif smoothing_method == 'adjusted_add_k_smoothing':
                prob = adjusted_add_k_smoothing(review[i-1], review[i],
                                    unigram_freq, bigram_freq, V, k)
            else:
                prob = bigram_freq.get((review[i-1], review[i]), 0) /
                                    unigram_freq.get(review[i-1], 1)

            each_review_prob *= prob

        each_review_prob = max(each_review_prob, 1e-20)
        log_prob_sum += log(each_review_prob)

    avg_neg_log_prob = (-1 * log_prob_sum) / N
    perplexity = exp(avg_neg_log_prob)
    return perplexity
```

Listing 5: Perplexity computation Function

2 Evaluation, Analysis, and Findings

2.1 Evaluation

Dataset Preprocessing: The data went through a series of preprocessing steps to make it suitable for language modeling. This included converting text to lowercase for consistency, managing punctuations, replacing numbers with placeholders, and introducing custom tokens to signify the start and end of reviews. Such preprocessing aids in reducing data noise and achieving better model performance.

N-gram Models: The approach uses unigram, and bigram language models to predict word sequences. The probabilities for these models are computed from the training dataset, which is essential for the system's subsequent prediction capabilities.

Handling Unseen Words: A fundamental challenge in language modeling is dealing with words not seen during training. Our approach replaces such words with a `<UNK>` token, which ensures the model can handle new words encountered in the validation set.

Smoothing Techniques: To further enhance the model's performance on unseen word combinations, smoothing techniques were employed. Both Laplace (add-one) and Add-k smoothing were implemented. Smoothing helps assign non-zero probabilities to unseen bigrams or trigrams, making the model more robust.

Perplexity Measurement: Perplexity was used as the evaluation metric, providing insight into the model's performance. Ideally, an unsmoothed model would have infinite perplexity if there are any unseen N-grams in the validation set. However, both the Laplace and Add-k smoothed models exhibited a perplexity of **1.38**.

2.2 Analysis

Upon analyzing the results, several insights were derived:

- The infinite perplexity of the unsmoothed model underscores the challenges of unseen N-grams and the importance of smoothing in language modeling.
- Both Laplace and Add-k smoothed models having the same perplexity of **1.38** are interesting. It suggests that either the dataset has unique characteristics or there might be an oversight in the evaluation process.
- The consistent perplexity values between the Laplace and Add-k smoothed procedures result from the comprehensiveness of the dataset and the selected 'k' value in Add-k smoothing, which may closely resemble Laplace smoothing. To further understand these results, the connection between dataset properties and smoothing factors is to be addressed further.

2.3 Findings

Data Preprocessing Importance: The preprocessing steps were crucial in shaping the dataset for effective language modeling. Handling punctuations, and numbers, and introducing custom tokens improved the model's interpretability and performance.

Necessity of Smoothing: The infinite perplexity of the unsmoothed model highlights the importance of smoothing in N-gram language models. Smoothing techniques ensure that the model can handle unseen word combinations in the validation set.

Optimal Smoothing Technique: With both smoothing techniques yielding the same perplexity, determining an optimal method for this dataset is challenging. Further studies or a deeper analysis might provide more clarity.

Further Improvements: The consistent results between the smoothed techniques suggest there are potential areas for further exploration. Considering other smoothing techniques, refining preprocessing steps, or experimenting with a broader range of N-gram models might provide more varied results and potential performance enhancements.

3 Others

3.1 Libraries Used

The main libraries used include:

- **re**: For regex-based text processing.
- **collections**: Specifically, the Counter class to compute word frequencies.
- **math**: For logarithmic and exponential calculations.

3.2 Group Member Contributions

- **Abhinav Yalamaddi**:
  - Implemented the preprocessing steps, including tokenization, handling of special characters, and unknown word handling.
  - Worked on the computation of bigram frequencies, ensuring only valid bigrams (both words present in unigram frequency) are included.
  - Implemented the add-k smoothing method and adjusted it for various values of k.
  - Worked on implementing perplexity calculation.
- **Satwik Arvapalli**:
  - Computed the unigram frequencies and filtered out words with a frequency below the threshold.
  - Worked on the Laplace smoothing technique and its integration into the perplexity computation.
  - Worked on implementing perplexity calculation.

Both members collaborated in debugging, testing, and validating the results on the validation dataset and documentation.

3.3 Feedback

The project was moderately challenging, offering a good balance between theory and implementation. We collectively spent around 40 hours on it. The hands-on approach definitely enhanced our understanding of n-gram language models. One suggestion would be to provide more datasets for experimentation.