Assignment 1

https://github.com/AdemOdza/CS6320_Assignment1

Group11 Abhirup Mukherjee AXM240026 Adem Odza AXO180008 Buvana Seshathri BXS240020

Samiha Kuncham SXK240025

1 Implementation Details

1.1 Preprocessing Decisions

We are given two plain-text corpora of reviews where each line is a single review. In order to compute the counts for the unigram and bigram, we need to tokenize the words into two flat Python lists with each word of the corpora being a single list element. We chose a list as our data structure since it is easily iterable and sliceable.

The following preprocessing decisions were made, along with the rationale behind each choice:

1. **Lowercasing** - We converted all the text to lower case to ensure that all the words with the same spelling are treated the same way. This process helped us in reducing the vocabulary size and it also helped the model generalize better.
2. **Removed Punctuation** - All punctuation is removed so that the focus is on the actual words in the text. This decision was taken since punctuation often doesn't contribute significant meaning in bag-of-words or similar models and it can introduce unnecessary noise. We used the regular expression module `re` from the Python library to remove the punctuation in the text.
3. **Tokenization** - Since the data given to us is already tokenized with one review per line and the tokens separated by spaces, we simply split each line into words. This made it straightforward and easy to process each word individually.

The preprocessing steps resulted in two flat Python lists called `train_tokens` and `val_tokens`.

1.2 Unknown word handling

The input to this step is a list of tokens (words), and the output is a modified list in which rare words are replaced with the token `UNK`. This ensures that infrequent words are handled consistently, reducing sparsity in the dataset and improving the robustness of the model.

The steps were carried out are as follows:

1. **Threshold Selection** - We set the threshold for rare words at 2. Since the dataset is relatively small, a threshold of 2 ensures that we do not lose excessive vocabulary information while still handling infrequent words effectively.
2. **Frequency Calculation** - A frequency dictionary was created to count occurrences of each token. For instance, if the token *hotel* appears five times, then `freq['hotel'] = 5`.
3. **Replacement Rule** - If a token appears fewer than 2 times, it is replaced with `UNK`. Otherwise, the original token is retained.

The result is a token list where all rare words are replaced by `UNK`, helping the model handle unseen or low-frequency words better.

```python
# unk handling
def replace_with_unk(tokens, threshold=2):
    freq = Counter(tokens)
    return ['UNK' if freq[token] < threshold else token for token in tokens]
```

Listing 1: Function to replace unknown words

1.3 Unigram Counts

We chose to create a dictionary of the form $\{$`'`w_i`':freq(`w_i`)`$\}$ where w_i is a word in our vocabulary (key) and $freq(w_i)$ is its frequency in the training corpus (value). Python dictionaries offer constant time key insertions and lookups ($O(1)$) and are simple to work with. To create this dictionary, we initialized an empty dictionary `unigram_freq` and iterated through the training tokens. If a word is not in the dictionary, then we add it to the dictionary with a frequency of 0. Otherwise, we update its frequency.

1.4 Bigram Counts

Next, we compute the bigram frequencies into a `bigram_freq` dictionary where each key is a bigram and the value is its frequency. To create this dictionary, we initialize the empty `bigram_freq` dictionary and iterate. We use the `zip` function to combine our `train_tokens_with_stop` list and a copy of this list starting from the second word to create a sequence of tuples, where each tuple is a bigram. We then iterate over each tuple and follow the same counting logic as before: if the tuple is not in `bigram_freq`, we assign its value as 0. Otherwise, we increment its count. This gives us a dictionary of the form $\{$`('`w_{i-1}`','`w_i`'):bigram frequency`$\}$.

1.5 Smoothing

Smoothing techniques are used to handle tokens (for unigrams) and pairs (for bigrams) that were not seen during the train phase. When a word/pair occurs during the test phase that has not been encountered in the train phase, probability of that token becomes zero. Zero probability results in **infinite perplexity**. To avoid this, we use smoothing techniques.

In this assignment, we have explored **add-k smoothing** techniques with different `k` values. `k` values used are `[0.001, 0.01, 0.1, 0.5, 1.0]`. Add-k smoothing with `k=1.0` is called **Laplace smoothing**.

Formulae for calculating Add-k smoothing:

For Unigram:

$$P(w) = \frac{count(w) + k}{N + k * V}$$

For Bigram:

$$P(w, v) = \frac{count(w, v) + k}{count(w) + k * V}$$

where $count(w)$ is the count of token w, $count(w, v)$ is the count for pair (w, v), N is the corpus size, and V is the vocabulary size

```python
def add_k_smoothing_unigram(c_w, N, V, k):
  return (c_w + k) / (N + k * V)

def add_k_smoothing_bigram(c_w_v, c_w, V, k):
  return (c_w_v + k) / (c_w + k * V)
```

Listing 2: Functions to perform smoothing

The above functions were used to calculate smoothing. These functions are called during perplexity calculation and they return the smoothed probabilities.

1.6 Perplexity

We use the following formula for calculating perplexity:

$$l = \frac{1}{N}\sum_{i=1}^{N} \log P(w_i|w_{i-1}, \ldots, w_{i-n+1})$$

$$PP = 2^{-l}$$

where N is the size of the training set, $P(w_i|w_{i-1}, \ldots, w_{i-n+1})$ is the probability of the ith N-Gram

This is implemented using two functions: one for unigrams and one for bigrams. Both functions take the `corpus` (preprocessed corpus), `unigram_counts` (the number of unigrams in the training set), N (training corpus size), V (vocabulary size), and k (smoothing parameter). The bigram perplexity function takes one additional parameter, `bigram_counts` (the number of unigrams in the training set).

```python
def calculate_unigram_perplexity(corpus, unigram_counts, N, V, k):
    # M = test corpus size
    M = len(corpus)
    total_log_prob = 0.0

    for token in corpus:
        # set default value of count to 0
        # if token not found in train set tokens
        c_w = unigram_counts.get(token, 0)
        prob = add_k_smoothing_unigram(c_w, N, V, k)
        total_log_prob += math.log2(prob)

    l = total_log_prob / M
    perplexity = math.exp2(-l) # 2.0 ** -l

    return perplexity
```

Listing 3: The unigram perplexity function

The smoothed probabilities are retrieved using the `add_k_smoothing_unigram` function, which gets the value of the `k` parameter and calculates the smoothed probabilities. We get the smoothed probability of every n-gram in the corpus, find their logarithm, and find their sum. We then divide that sum by M to get the l value. We then raise 2 to the power of $-l$ before returning.

Similarly, when calculating the bigram perplexity, we pair up the words in the corpus using the `zip` tool and iterate through each tuple of words. We fetch the counts of the bigram and the count of the previous word and pass these as arguments to the `add_k_smoothing_bigram` function along with V and k. We get the smoothed probability of every bigram in the corpus, find their logarithm, and find their sum. We then divide that sum by M to get the l value. We then raise 2 to the power of $-l$ before returning.

```
def calculate_bigram_perplexity(corpus, unigram_counts, bigram_counts, V, k):
    # Create bigrams from the validation corpus
    bigram_corpus = zip(corpus, corpus[1:])
    M = len(corpus)
    total_log_prob = 0.0

    for bigram in bigram_corpus:
        previous_word = bigram[0]
        # set default value of count to 0
        # if bigram not found in train set bigrams
        c_w_v = bigram_counts.get(bigram, 0)
        c_w = unigram_counts.get(previous_word, 0)
        prob = add_k_smoothing_bigram(c_w_v, c_w, V, k)
        total_log_prob += math.log2(prob)

    l = total_log_prob / M
    perplexity = math.exp2(-l) # 2.0 ** -l

    return perplexity
```

Listing 4: The bigram perplexity function

2 Evaluation, Analysis, and Findings

For hyperparameter `k` for **add-k smoothing**, upon trying different values, we inferred that while `k=0.001` resulted in lower perplexities in the training set, `k=0.01` produced significantly better results (lower perplexities) for both bigram and unigram during the testing phase. This helped us evaluate model performance and mitigate over-smoothing.

Table 1: Training set Perplexity with different k values in add-k smoothing

	$k = 0.001$	$k = 0.01$	$k = 0.1$	$k = 0.5$	$k = 1.0$
Unigram	391.324	391.324	391.340	391.680	392.592
Bigram	36.4174	51.723	122.885	303.366	459.282

Table 2: Validation set Perplexity with different k values in add-k smoothing

	$k = 0.001$	$k = 0.01$	$k = 0.1$	$k = 0.5$	$k = 1.0$
Unigram	756.657	670.337	594.195	547.696	530.667
Bigram	425.511	293.423	341.753	542.233	703.129

3 Programming Libraries Used

The following modules and libraries were used:

- `re` - This module provides regular expression matching operations similar to those found in Perl.
- `Counter` - We used the `Counter` class from the `collections` module, which is a dict subclass for counting hashable objects.
- `math` - This module provides access to common mathematical functions and constants, including those defined by the C standard.
- `tabulate` - We used the `tabulate` which is a PyPI library that allows pretty-printing of tabular data.

4 Contributions of members

We began by preprocessing the dataset, a step handled by **Samiha**. This involved converting all words to lowercase, removing punctuation, and tokenizing the reviews into words. Unknown or rare words were replaced with `UNK` tokens using a threshold of 2, and the processed tokens were stored in a list data structure.

Next, **Abhirup** computed unigram and bigram frequencies from this preprocessed data, producing two dictionaries: one for unigram frequencies and one for bigram frequencies. He also integrated the smoothing functions that give the smoothed probabilities inside the perplexity functions once they were ready

Buvana applied the add-k smoothing and Laplace smoothing techniques for the training data and the validation data. Additionally, she computed, analyzed, and tabulated the perplexity numbers for different `k` values.

Finally, **Adem** implemented the perplexity calculation formula. Two functions were written, one for unigram perplexity and another for bigram perplexity. He also removed code smells and ensured the end-to-end pipeline worked correctly.

All members put in equal efforts in completing both the source code and report for this assignment and we are thankful for each other's contributions.

5 Feedback

We feel that this assignment gave us a chance to apply the theory that was presented during the lectures as a hands-on exercise. It cleared our misconceptions and deepened our understanding of elementary language models. As such, we feel that the difficulty level of this assignment was appropriate when compared to the material covered in the lectures. We enjoyed working on this assignment and are thankful for our instructor Dr Xinya Du's guidance.