Assignment 1

https://github.com/AdemOdza/CS6320_Assignment1

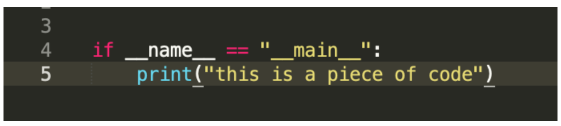
Group11 Abhirup Mukherjee AXM240026 Adem Odza AXO180008 Buvana Seshathri BXS240020

Samiha Kuncham SXK240025

TODO: Fill in your name and details above. If not filled correctly, we will subtract 2pt.

TODO: It is highly suggested to use the template for reports. You are free to use other softwares (e.g. Microsoft Word, Google Doc), but we prefer you follow the structure below.

1 Implementation Details



```
if __name__ == "__main__":
    print("this is a piece of code")
```

Figure 1: Example of a Code Piece.

```
import numpy
print("this is a piece of code")
```

Listing 1: Example of a Code Piece.

1.1 Unigram and bigram probability computation (15%)

1.1.1 Preprocessing Decisions

We are given two plain text corpora of reviews where each line is a single review. In order to compute the counts and probabilities 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 slicable.

These were the preprocessing steps that we performed:

a. Strip the leading and trailing whitespaces such as spaces, tabs, and newlines.

b. If the line is empty after removing whitespaces, move to the next line in the file.

c. Preprocess the current line by converting everything into lower case.

d. Remove the punctuations using the `re` library.

e. If the processed line is not empty, split the line into individual words and add these into the token list.

f. All words are flattened into a single list from the non-empty processed lines.

(Samiha to write the how and the why)

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

1.1.2 Unigram Counts and Probabilities

We begin with our `train_tokens` list and append start and stop tokens to it. The start token gives the bigram context of the first word and the stop token makes the bigram grammar a true probability distribution.[1]

The unigram is the simplest of language models. The unigram probability of a given word is computed using the formula:

$$P(w_i) = \frac{Count(w_i)}{N}$$

where:

- $Count(w_i)$ is the number of times the word w_i appears in the training text.
- N is the total number of words (tokens) in the training text.

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 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.

Next, we initialize another empty dictionary called `unigram_probs`. We iterate through our `unigram_freq` dictionary and keep adding the (key, value) pairs to this new dictionary after dividing the value by N. This gives us our desired dictionary of the form $\{$`'`w_i`':P(`w_i`)`$\}$.

1.1.3 Bigram Counts and Probabilities

The bigram model calculates the probability of a word based on the word that appeared immediately before it. In other words, it computes the conditional probability $P(w_i|w_{i-1})$ of the word w_i, given that the previous word was w_{i-1}. The bigram probability can be computed using the formula:

$$P(w_i|w_{i-1}) = \frac{Count(w_{i-1}, w_i)}{Count(w_{i-1})}$$

where:

- $Count(w_{i-1}, w_i)$ is the number of times the pair of words (the bigram) appears together in that order.
- $Count(w_{i-1})$ is the unigram count of the preceding word.

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`$\}$.

Finally, we calculate the bigram probabilities by initializing an empty dictionary `bigram_probs`. We iterate through our `bigram_freq` dictionary and add the (key, value) pairs to this new dictionary after dividing the value by the unigram frequency of the first word in the bigram.

TODO:

1.2 Smoothing (15%)

1.3 Unknown word handling (15%)

1.4 Implementation of perplexity (15%)

2 Eval, Analysis and Findings

TODO: to add (30%)

3 Others

TODO: to add (10%)

References

[1] Chapter 3, Section 3.1.2, Daniel Jurafsky and James H. Martin. *Speech and Language Processing (3rd ed. draft)*. https://web.stanford.edu/~jurafsky/slp3/. 2021.