CS 4343 - Problem Set 2: Language Models

Nabeel Mahmood

Due Date: **September 30, 2024**

October 3, 2024

Report

Data Structures

Dictionaries: Unigram, bigram, and trigram frequency dictionaries are used to store the frequencies of one-word, two-words, and three-words. In Python, dictionaries are implemented as hash tables, which function similarly to arrays but they allow us to store and access a collection of values efficiently.

Lists: Lists are used to store the tokens that are taken from the text, which makes it simple to go through them for counting frequencies and calculating the probabilities.

Preprocessing

I loaded my Shakespeare.txt file into the program. I opened the dataset and read its contents into a variable. This way, I can easily work with the text for tasks like tokenization and frequency counting.

```
with open("/Users/nabeelmahmood/NLP/Shakespeare.txt", "r") as i:
   doc = i.read()
```

We start by processing the input text to extract tokens. This involves ignoring punctuation and converting everything to lowercase for consistency. We use regular expressions for tokenization, as shown below:

```
doc = re.sub(r'[0-9]', '', doc)

doc = re.sub(r'[^a-z\s!?]', '', doc)

doc_split = re.findall(r'[a-zA-Z!?]+', doc)
```

N-Gram Frequencies

Unigram, bigram, and trigram frequencies are calculated by iterating through the tokenized list of words. For example, the code snippet shows how unigram frequencies are calculated:

Unigram

```
unigram_lm = dict()

for token in doc.split():
#    print(token)
    if token in unigram_lm:
        unigram_lm[token] += 1
    else:
        unigram_lm[token] = 1
```

Bigram

```
bigrams = dict()

tokens = doc.split()
for i in range( 0, len(tokens) -1,  1 ):
    bigram = tokens[i] + " " + tokens[i+1]

    if bigram in bigrams:
        bigrams[bigram] += 1
    else:
        bigrams[bigram] = 1
```

Trigram

```
trigrams = dict()

tokens = doc.split()
for i in range( 0, len(tokens) -2):
    trigram = tokens[i] + " " + tokens[i+1] + " " + tokens[i+2]

    if trigram in trigrams:
        trigrams[trigram] += 1
    else:
        trigrams[trigram] = 1
```

Laplace Smoothing

```
def bigram_prob_smoothing(bigrams, first_word, second_word):
    V = len(unigram_lm)
    word_size = 0

    bigram_count = bigrams[bigram]

    for key in bigrams:
        if key.split()[0] == first_word:
            word_size += bigrams[key]
```

```
return (bigram_count+ 1) / (word_size + V)
```

Space and Time Complexity Analysis

Space Complexity

The space complexity for storing the unigram, bigram, and trigram models is $O(V + N)$, where V is the size of the lexicon and N is the number of tokens in the dataset. Each model needs space based on the number of unique words and their combinations.

Unigram: $O(V)$

Bigram: $O(V^2)$

Trigram: $O(V^3)$

The trigram model uses more memory than the unigram and bigram models because it needs to store three words. This results in a larger number of possible trigrams, making it consume more memory compared to unigram and bigram models, which only store one and two words.

Time Complexity

The time complexity for counting unigrams, bigrams, and trigrams is $O(N)$, where N is the total number of tokens. This is because we iterate through the list of tokens once for each model. The time complexity for unigram, bigram, and trigram all run in $O(N)$ because they all take one for loop to iterate through the list of tokens, but it mainly depends on your operations so trigram can take more operations compared to unigram and bigram.

Testing

To make sure your language model works, there is a lot of testing and debugging involved. For example, when I was tokenizing my trigrams, I had two for loops, and when I ran that in Jupyter Notebook, my computer just turned off. I realized that it is very important to test your code and each section very carefully because memory usage is crucial. I tested multiple functions in Jupyter Notebook to make sure that I was getting the unigrams, bigrams, and trigrams correctly, along with their probabilities. There were times when my Discord bot was predicting ten sentences, and after a minor change in the code, my bot just wouldn't predict anymore. It requires a lot of testing to ensure your program is running correctly.

Below are examples of the output generated by the models:

- **User Input:** "$hello this is"
- **Bigram Model Prediction:** "the 100th etext file presented by project gutenberg, and is"
- **Trigram Model Prediction:** "man that hath a heart as i am not in"
- **User Input:** "$hello how are you"
- **Bigram Model Prediction:** " should read! this is the th etext file presented by"
- **Trigram Model Prediction:** "not to be a man of his own hand did"

$$\log P(X) = \log(p_1) + \log(p_2) + ... + \log(p_n)$$

Comparison of Bigram and Trigram Models

The trigram model did better than the bigram model in generating the ten predicted words. This makes sense since the trigram model looks at the two previous words, providing more context than the bigram model, which only looks at one. This added complexity of the trigram model does lead to higher memory usage and can create issues with the frequency counts. We used Naive Bayes, but it is clear that the trigram model could be even more accurate and faster with other algorithms and models.

Conclusion

Looking back at my implementation, I see that theirs definitely room for improvement. I feel like the probabilities for the bigram and trigram models aren't quite where they should be, and my implementation could use some more work to better its accuracy and probability. In conclusion, there's definitely room for improvement and I'm excited to explore those improvements in the future.

Application Development

```
#********************************#
# Name: Nabeel Mahmood           #
# Username: <if applicable>      #
# Problem Set: PS2               #
# Due Date: October 1st, 2024    #
#********************************#

import re
import array
import random
import math

with open("/Users/nabeelmahmood/NLP/Shakespeare.txt", "r") as i:
    doc = i.read()

doc = doc.lower()

doc = re.sub(r'[0-9]', '', doc)

doc = re.sub(r'[^a-z\s!?]', '', doc)

doc_split = re.findall(r'[a-zA-Z!?]+', doc)


#doc = "this is a test test test test hello nerd this is a test test"


# unigram frequencies
unigram_lm = dict()

for token in doc.split():
#    print(token)
    if token in unigram_lm:
        unigram_lm[token] += 1
    else:
        unigram_lm[token] = 1




# bigram frequencies
bigrams = dict()

tokens = doc.split()
for i in range( 0, len(tokens) -1,  1 ):
    bigram = tokens[i] + " " + tokens[i+1]

    if bigram in bigrams:
        bigrams[bigram] += 1
```

```
    else:
        bigrams[bigram] = 1

#print(f"Bigrams:  {bigrams}")


# trigram frequencies
trigrams = dict()

tokens = doc.split()
for i in range( 0, len(tokens) -2):
    trigram = tokens[i] + " " + tokens[i+1] + " " + tokens[i+2]

    if trigram in trigrams:
        trigrams[trigram] += 1
    else:
        trigrams[trigram] = 1




# laplace smoothing
def bigram_prob_smoothing(bigrams, first_word, second_word):
    V = len(unigram_lm)
    word_size = 0

    bigram_count = bigrams[bigram]

    for key in bigrams:
        if key.split()[0] == first_word:
            word_size += bigrams[key]


    return (bigram_count+ 1) / (word_size + V)




# trigram prob
def trigram_prob(word1, word2):

    best_next_word = None
    highest_prob = 0.0
    V = len(bigrams)

    for trigram in trigrams:
        first_word = trigram.split()[0]

        second_word = trigram.split()[1]
        third_word = trigram.split()[2]
```

```
        if first_word == word1 and second_word == word2:
            trigram_count = trigrams[trigram]

            bigram_key = word1 + " " + word2


            bigram_count = bigrams.get(bigram_key)

            # prob = trigram_count / bigram_count
            prob = (trigram_count + 1) / (bigram_count + V)

            if prob > highest_prob:
                highest_prob = prob
                best_next_word = third_word

                if highest_prob == 0:
                    return bigram_prob_smoothing(bigrams, first_word, second_word)


    if best_next_word != None:
        return best_next_word







#print( f"Bigram with the highest probability: '{highestProbkey}' with probability: {highestProb}")



#bigram log
def calc_bigram_log(w1, w2):
    total_log = 0
    prob = bigram_prob_smoothing(bigrams, w1, w2)


    if prob > 0:
        total_log += math.log(prob)
    else:
        total_log += math.log(1e-10)

    return total_log

#trigram log
def calc_trigram_log(w1, w2):
```

```
    total_log = 0


    prob = trigram_prob(w1, w2)

    if isinstance(prob, (int, float)):
        total_log += math.log(prob)
    else:
        total_log += math.log(1e-10)

    return total_log




def predict_next_ten_words_trigrams(w1, w2, numwords=10):
    predictWords = []
    string = ""

    for i in range(numwords):
        next_word = trigram_prob(w1, w2)
        if next_word:
            predictWords.append(next_word)
            w1, w2 = w2, next_word
        else:
            next_word = predict_next_word_bigram(w2)
            predictWords.append(next_word)
            w1, w2 = w2, next_word


    for triword in predictWords:
        string += triword + " "


    return string.strip()




def predict_next_word_bigram(w2, numwords = 10):
    predictWords = []
    string = ""

    for i in range(numwords):
        next_word = None
        highest_prob = 0

        for bigram in bigrams:
            first_word = bigram.split()[0]
            if first_word == w2:
                second_word = bigram.split()[1]
```

```
            prob = bigram_prob_smoothing(bigrams, first_word, second_word)
            #print(f"Checking bigram: {bigram}, Probability: {prob}")

            if prob > highest_prob:
                highest_prob = prob
                next_word = second_word
                break


    if next_word:
        predictWords.append(next_word)
        w2 = next_word

    else: random.choice(list(unigram_lm))


for biword in predictWords:
    string += biword + " "


return string.strip()
```