Problem 1 (15 pts) - Text Classification with Naïve Bayes

a.

- P(spam) =$\frac{3}{5}$
- P(ham) = $\frac{2}{5}$

b.

Word	$P(Word \mid Ham)$	$P(Word \mid Spam)$
Buy	0	$\frac{1}{3}$
Car	$\frac{1}{2}$	$\frac{1}{3}$
Nigeria	$\frac{1}{2}$	$\frac{2}{3}$
Profit	0	$\frac{2}{3}$
Money	$\frac{1}{2}$	$\frac{1}{3}$
Home	1	$\frac{1}{3}$
Bank	$\frac{1}{2}$	$\frac{2}{3}$
Check	0	$\frac{1}{3}$
Wire	0	$\frac{1}{3}$
Fly	$\frac{1}{2}$	0

c.

$$P(Label \mid X_1, \dots, X_d) = \alpha[P(Label) \prod_i P(X_i \mid Label)]$$

$$y^* = argmax\, P(y) \prod_i P(x_i|y)$$

Sentence: Nigeria

$$P(Ham|Nigeria) = P(ham) * P(Nigeria|Ham) = \frac{2}{5} * \frac{1}{2} = \frac{1}{5}$$

$$P(Spam|Nigeria) = \cdots = \frac{3}{5} * \frac{2}{3} = \frac{2}{5}$$

$$y^* = SPAM$$

Sentence: Nigeria home

$$P(Ham|Nigeria, home) = P(ham) * P(Nigeria|Ham) * P(home|Ham) = \frac{2}{5} * \frac{1}{2} * 1 = \frac{1}{5}$$

$$P(Spam|Nigeria, home) = P(spam) * P(Nigeria|spam) * P(home|Spam) = \frac{3}{5} * \frac{2}{3} * \frac{1}{3}$$

$$= \frac{2}{15}$$

$$y^* = HAM$$

Sentence: home bank money

$$P(Ham|home, bank, money)$$

$$= P(ham) * P(home|Ham) * P(bank|Ham) * P(money|Ham)$$

$$= \frac{2}{5} * 1 * \frac{1}{2} * \frac{1}{2} = \frac{1}{10}$$

$$P(Spam|Nigeria, home)$$

$$= P(spam) * P(home|spam) * P(bank|Spam) * P(money|Spam)$$

$$= \frac{3}{5} * \frac{1}{3} * \frac{2}{3} * \frac{1}{3} = \frac{6}{135}$$

$$y^* = HAM$$

Problem 2 (15 pts) – Bigram Models

Show that if you sum up the probabilities of all sentences of length n under a bigram language model, this sum is exactly 1 (i.e. the model defines a proper probability distribution). Assume a vocabulary size of V

$$\sum_{w_1,w_2,\dots,w_n} P(w_1, w_2, \dots, w_n) = \sum_{w_1,w_2,\dots,w_n} P(w_1|start) * P(w_2|w_1) * \cdots * P(w_n|w_{n-1}) = 1$$

Hint: Use induction over the sentence length.

Step 1: Base case

Consider the base case of $n = 1$. We have

$$\sum_{w_1} P(w_1) = P(V) = 1$$

Step 2: Hypothesis

Assume our claim holds for $n = k$. We then want to show that

$$\sum_{w_1,w_2,\dots,w_{k+1}} P(w_1, w_2, \dots, w_{k+1})$$
$$= \sum_{w_1,w_2,\dots,w_{k+1}} P(w_1|start) * P(w_2|w_1) * \cdots * P(w_k|w_{k-1}) * P(w_{k+1}|w_k) = 1$$

We have

$$\sum_{w_1,w_2,\dots,w_{k+1}} P(w_1, w_2, \dots, w_{k+1})$$
$$= \sum_{w_1} \sum_{w_2} \cdots \sum_{w_{k+1}} P(w_1|start) * P(w_2|w_1) * \cdots * P(w_k|w_{k-1}) * P(w_{k+1}|w_k)$$
$$= \sum_{w_1} \sum_{w_2} \cdots \sum_{w_k} \left[P(w_1|start) * P(w_2|w_1) * \cdots * P(w_k|w_{k-1}) \sum_{w_{k+1}} P(w_{k+1}|w_k) \right]$$

Assuming that $P(w_i) > 0$ for $i = 1,2, \dots, |V|$, we have

$$\sum_{w_{k+1}} P(w_{k+1}|w_k) = \sum_{w_{k+1}} \frac{P(w_k, w_{k+1})}{P(w_k)} = \frac{\sum_{w_{k+1}} P(w_k, w_{k+1})}{P(w_k)} = \frac{P(w_k)}{P(w_k)} = 1$$

Thus, we have

$$\begin{aligned}
\sum_{w_1, w_2, \dots, w_{k+1}} & P(w_1, w_2, \dots, w_{k+1}) \\
&= \sum_{w_1} \sum_{w_2} \cdots \sum_{w_k} \left[P(w_1|start) * P(w_2|w_1) * \cdots * P(w_k|w_{k-1}) \sum_{w_{k+1}} P(w_{k+1}|w_k) \right] \\
&= \sum_{w_1} \sum_{w_2} \cdots \sum_{w_k} [P(w_1|start) * P(w_2|w_1) * \cdots * P(w_k|w_{k-1})] \\
&= \sum_{w_1} \sum_{w_2} \cdots \sum_{w_k} P(w_1, \dots, w_k) \\
&= \sum_{w_1, w_2, \dots, w_k} P(w_1, \dots, w_k) \\
&= 1
\end{aligned}$$

where the last equality holds by our induction hypothesis.

Thus, we can conclude by weak induction that if you sum up the probabilities of all sentences of length n under a bigram language model, this sum is exactly 1. That is,

$$\sum_{w_1, w_2, \dots, w_n} P(w_1, w_2, \dots, w_n) = \sum_{w_1, w_2, \dots, w_n} P(w_1|start) * P(w_2|w_1) * \cdots * P(w_n|w_{n-1}) = 1$$

Programming Component – Building a Trigram Language Model

Part 1 – extracting n-grams from a sentence (10 pts)

```
def get_ngrams(sequence, n):
    """
    COMPLETE THIS FUNCTION (PART 1)
    Given a sequence, this function should return a list of n-grams, where
each n-gram is a Python tuple.
    This should work for arbitrary values of 1 <= n < len(sequence).
    """

    #Invalid input, return empty
    if len(sequence) < n or n < 1:
        return []

    startVar = 'START'
    stopVar = 'STOP'
    sequence_copy = sequence.copy()
    sequence_copy.append(stopVar)

    n_grams = list()

    if n == 1:
        n_grams.append(tuple(['START']))
    else:
        #create tuples using start key
        for i in range(-n, 0):
            curr_tuple = tuple(['START'],)
            j = 1
            index = i
            while j < n:
                index += 1
                j += 1
                if index < 0:
                    curr_tuple = curr_tuple + tuple(['START'],)
                else:
                    curr_tuple = curr_tuple + tuple([sequence_copy[index]])
                    if len(curr_tuple) == n:
                        n_grams.append(curr_tuple)




    #create rest of tuples
    for i in range(0, len(sequence_copy)):
        curr_tuple = tuple([sequence_copy[i]])
        tuple_is_valid = True
        for j in range((i+1), min(i+n, len(sequence_copy))):
            if j > len(sequence_copy):
                tuple_is_valid = False
                continue
            curr_tuple = curr_tuple + tuple([sequence_copy[j]])
        if tuple_is_valid and len(curr_tuple) == n:
            n_grams.append(curr_tuple)
    return n_grams
```

Part 2 – counting n-grams in a corpus (10 pts)

```
def count_ngrams(self, corpus):
    """
    COMPLETE THIS METHOD (PART 2)
    Given a corpus iterator, populate dictionaries of unigram, bigram,
    and trigram counts.
    """
    self.unigramcounts = {} # might want to use defaultdict or Counter
instead
    self.bigramcounts = {}
    self.trigramcounts = {}

    countSentences = 0
    for sentence in corpus:
        countSentences += 1
        sent_unigrams = get_ngrams(sentence, 1)
        sent_bigrams = get_ngrams(sentence, 2)
        sent_trigrams = get_ngrams(sentence, 3)

        for unigram in sent_unigrams:
            if unigram not in self.unigramcounts.keys():
                self.unigramcounts[unigram] = 1
            else:
                count = self.unigramcounts[unigram] + 1
                self.unigramcounts[unigram] = count

        for bigram in sent_bigrams:
            if bigram not in self.bigramcounts.keys():
                self.bigramcounts[bigram] = 1
            else:
                count = self.bigramcounts[bigram] + 1
                self.bigramcounts[bigram] = count

        for trigram in sent_trigrams:
            if trigram not in self.trigramcounts.keys():

                self.trigramcounts[trigram] = 1
            else:
                count = self.trigramcounts[trigram] + 1
                self.trigramcounts[trigram] = count

    start_tuple = tuple(['START'],)
    start_tuple = start_tuple + tuple(['START'],)
    self.bigramcounts[start_tuple] = countSentences
    self.trigramtotals = sum(self.trigramcounts.values())
    self.bigramtotals = sum(self.bigramcounts.values())
    self.unigramtotals = sum(self.unigramcounts.values())

    return
```

Part 3 – Raw n-gram probabilities (10 pts)

```
def raw_trigram_probability(self,trigram):
    """
    COMPLETE THIS METHOD (PART 3)
    Returns the raw (unsmoothed) trigram probability
    """

    if trigram not in self.trigramcounts.keys():
        return 0.0

    count = self.trigramcounts[trigram]
    denom = self.bigramcounts[trigram[:2]] #TODO
    if count > denom:
        print("problem")
    return count / denom


def raw_bigram_probability(self, bigram):
    """
    COMPLETE THIS METHOD (PART 3)
    Returns the raw (unsmoothed) bigram probability
    """
    if bigram not in self.bigramcounts.keys():
        return 0.0

    count = self.bigramcounts[bigram]
    denom = self.unigramcounts[(bigram[0],)] #TODO
    if count > denom:
        print("problem")
    return count / denom

def raw_unigram_probability(self, unigram):
    """
    COMPLETE THIS METHOD (PART 3)
    Returns the raw (unsmoothed) unigram probability.
    """

    #hint: recomputing the denominator every time the method is called
    # can be slow! You might want to compute the total number of words once,
    # store in the TrigramModel instance, and then re-use it.

    if unigram not in self.unigramcounts.keys():
        return 0.0

    count = self.unigramcounts[unigram]
    return count / self.unigramtotals
```

Part 4 – Smoothed probabilities (10 pts)

```
def smoothed_trigram_probability(self, trigram):
    """
    COMPLETE THIS METHOD (PART 4)
    Returns the smoothed trigram probability (using linear interpolation).
    """
    lambda1 = 1/3.0
    lambda2 = 1/3.0
    lambda3 = 1/3.0

    trigram_value = lambda3 * self.raw_trigram_probability(trigram)
    bigram_value = lambda2 * self.raw_bigram_probability(trigram[:2])
    unigram_value = lambda1 * self.raw_unigram_probability(trigram[:1])
    value = trigram_value + bigram_value + unigram_value
    return value
```

Part 5 – Computing Sentence Probability (10 pts)

```python
def sentence_logprob(self, sentence):
    """
    COMPLETE THIS METHOD (PART 5)
    Returns the log probability of an entire sequence.
    """
    trigrams = get_ngrams(sentence, 3)
    log_prob = 0
    for trigram in trigrams:
        smoothed_prop = self.smoothed_trigram_probability(trigram)
        curr_log_prob = 0
        if smoothed_prop != 0:
            curr_log_prob = math.log2(smoothed_prop)
        log_prob = log_prob + curr_log_prob
    return log_prob
```

Part 6 – Perplexity (10 pts)

```
def perplexity(self, corpus):
    """
    COMPLETE THIS METHOD (PART 6)
    Returns the log probability of an entire sequence.
    """
    log_prob = 0
    denom = 0
    for sentence in corpus:
        n_grams = get_ngrams(sentence, 3)
        curr_log_prob = self.sentence_logprob(sentence)
        log_prob = log_prob + curr_log_prob
        denom += len(n_grams)
    value = log_prob / denom
    exp_value = 2**-value
    return exp_value
```

Part 7 – Using the Model for Text Classification (10 pts)

```
def essay_scoring_experiment(training_file1, training_file2, testdir1, 
testdir2):

        model1_high = TrigramModel(training_file1)
        model2_low = TrigramModel(training_file2)

        total = 0
        correct = 0

        #testDir1=high
        for f in os.listdir(testdir1):
            pp = model1_high.perplexity(corpus_reader(os.path.join(testdir1, 
f), model1_high.lexicon))
            pp2 = model2_low.perplexity(corpus_reader(os.path.join(testdir1, 
f), model2_low.lexicon))
            if pp < pp2:
                correct += 1
            total += 1

        #testDir2=low
        for f in os.listdir(testdir2):
            pp_low = 
model2_low.perplexity(corpus_reader(os.path.join(testdir2, f), 
model2_low.lexicon))
            pp_high = 
model1_high.perplexity(corpus_reader(os.path.join(testdir2, f), 
model1_high.lexicon))
            if pp_low < pp_high:
                correct += 1
            total += 1
        return correct / total
```