.0081   .0083

NLP HW1

(1)

- prior probality:

$$P(\text{spam}) = \frac{3}{5} = 0.6 \quad P(\text{ham}) = \frac{2}{5} = 0.4$$

- p(word | class)

Spam	Ham
P(buy) = 1/12	P(money) = 1/7
P(car) = 1/12	P(bank) = 1/7
P(Nigeria) = 2/12	P(home) = 1/7
P(profit) = 2/12	P(car) = 2/7
P(money) = 1/12	P(Nigeria) = 1/7
P(home) = 1/12	P(fly) = 1/7
P(bank) = 2/12	
P(check) = 1/12	
P(wire) = 1/12	

- predicted class labels:

$$P(\text{label} \mid X_1, \ldots, X_a) = \frac{P(\text{label}) \prod P(X_i \mid \text{label})}{\prod P(X_i)}$$

$$P(\text{spam} \mid \text{Nigeria}) = \frac{P(\text{spam}) \cdot P(\text{Nigeria} \mid \text{spam})}{P(\text{Nigeria})} = \frac{(3/5)(1/6)}{(3/19)}$$

$$P(\text{ham} \mid \text{Nizeria}) = \frac{P(\text{ham}) \cdot P(\text{Nigeria} \mid \text{ham})}{P(\text{Nigeria})} = \frac{(2/5)(1/7)}{(3/19)}$$

$P(\text{spam} \mid \text{Nigeria}) > P(\text{ham} \mid \text{Nigeria})$

class → spam

$$P(\text{Spam} \mid \text{Nigena home}) = \frac{P(\text{spam}) \cdot P(\text{Nigeria} \mid \text{spam}) P(\text{home} \mid \text{spam})}{P(\text{Nigeria}) P(\text{home})} = \frac{(3/5)(1/6)(1/12)}{(3/19)(2/19)}$$

$$P(\text{ham} \mid \text{Nigeria home}) = \frac{P(\text{ham}) P(\text{Nigeria} \mid \text{ham}) P(\text{home} \mid \text{ham})}{P(\text{Nigeria}) P(\text{home})} = \frac{(2/5)(1/7)(1/7)}{(3/19)(3/19)}$$

class → ham

.00039
.0012

$$P(\text{spam}|\text{home bank money}) = \frac{P(S)P(\text{home}|S)P(\text{bank}|S)P(\text{money}|S)}{P(\text{home})P(\text{bank})P(\text{money})} = \frac{(\frac{3}{5})(\frac{1}{12})(\frac{1}{6})(\frac{1}{12})}{(\frac{2}{19})(\frac{3}{19})(\frac{2}{19})}$$

$$P(\text{ham}|\text{home bank money}) = \frac{P(H)P(\text{home}|H)P(\text{bank}|H)P(\text{money}|H)}{P(\text{home})P(\text{bank})P(\text{money})} = \frac{(\frac{2}{5})(\frac{1}{7})(\frac{1}{7})(\frac{1}{7})}{(\frac{2}{19})(\frac{3}{19})(\frac{2}{19})}$$

$P(\text{ham}|\dots) > P(\text{spam}|\dots)$

class → ham

(2)

Given:

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

We also know that:

$$P(w_n|w_{n-1}) = \frac{c(w_n w_{n-1})}{c(w_{n-1})}$$

We can substitute this into what we're given:

$$\sum_{w_1,\dots,w_n} \frac{c(w_1 \cdot \text{start})}{c(\text{start})} \dots \frac{c(w_n w_{n-1})}{c(w_{n-1})} = 1$$

Given $n=1$, we write:

$$P(w_1|\text{start}) = \sum_{w_1,\dots,w_n} P(w_1|\text{start}) = 1$$

Thus, we have

$$= \sum \frac{c(w_1 \cdot \text{start})}{c(\text{start})} = 1$$

If we expand this further, we have n unique sentences:

$$= \frac{c(w_1 \text{start})}{c(\text{start})} + \dots + \frac{c(w_n \text{start})}{c(\text{start})} = \frac{1}{c(\text{start})}(c(w_1 \text{start}) + \dots + c(w_n \text{start}))$$

Since there are i sentences with only one starting word in the set $\{w_1,\dots,w_n\}$, the sum of the counts of a sentence and its starting word w_i must be equal to the count of the total number of sentences:

$$c(w_1 \text{start}) + \dots + c(w_n \text{start}) = c(\text{start})$$

Thus, when $n=1$, $\sum P(w_1,\dots,w_n) = 1$.

If we assume that the same holds for $n=k$, then we can prove it holds for $n=k+1$!

$$P(W_{k+1}|W_k) = \frac{C(W_{k+1}W_k)}{C(W_k)}$$

Thus, we have:

$$P(W_1,\dots,W_{k+1}) = \sum \frac{C(W,\text{start})}{C(\text{start})} \cdots \frac{C(W_{k+1}W_k)}{C(W_k)} = 1$$

Given that there are only $k+1$ unique words we can factor out the count of these words:

$$= \frac{1}{C(\text{start})\dots C(W_k)} \sum^i C(W,\text{start})\dots C(W_{k+1}W_k) = 1$$

We can now prove:

$$\sum^i C(W,\text{start})\dots C(W_{k+1}W_k) = C(\text{start})\dots C(W_k)$$

From our base case, we can deduce that for each count $C(W_k)$ this is equal to the sum of all counts where W_k is the preceeding i.e for the set $\{1,\dots,k\}$ every instance of the bigram where $k \neq j$ where j is a member of the set. Formally,

$$\sum^j C(W_jW_k) = C(W_k)$$

Given the equation we are trying to prove, we can deduce that our newly defined W_j can be expanded:

$$\sum^j C(W_jW_k) \sum^i C(W_{j+1}W_{k+1})\dots = C(W_k)C(W_{k+1})\dots$$

And finally:

$$\sum^j C(W_jW_k)C(W_{j+1}W_{k+1})\dots = C(W_k)C(W_{k+1})\dots$$

In [57]:

```python
import sys
from collections import defaultdict
import math
import random
import os
import os.path
"""
COMS W4705 - Natural Language Processing
Homework 1 - Programming Component: Trigram Language Models
Yassine Benajiba
"""

def corpus_reader(corpusfile, lexicon=None):
    with open(corpusfile,'r') as corpus:
        for line in corpus:
            if line.strip():
                sequence = line.lower().strip().split()
                if lexicon:
                    yield [word if word in lexicon else "UNK" for word in s
                else:
                    yield sequence

def get_lexicon(corpus):
    word_counts = defaultdict(int)
    for sentence in corpus:
        for word in sentence:
            word_counts[word] += 1
    return set(word for word in word_counts if word_counts[word] > 1)



def get_ngrams(sequence, n):
    """
    COMPLETE THIS FUNCTION (PART 1)
    Given a sequence, this function should return a list of n-grams, where
    This should work for arbitrary values of 1 <= n < len(sequence).
    """
    n_gram_list = []
    sequence.insert(0,'START')

    # adding 'START' values to our sequence
    if (n > 2):
        for i_temp in range(n - 2):
            sequence.insert(0,'START')

    # adding 'STOP' value at the end of our sequence
    sequence.append('STOP')

    # main iteration through sequence
    for i in range(len(sequence) - n + 1):
        # temporary n_gram to be reset after each string in sequence
        temp_gram = ()

        for j in range(n):

            temp_tup = (sequence[i+j],)
            temp_gram = temp_gram + temp_tup

        n_gram_list.append(temp_gram)

    return n_gram_list


class TrigramModel(object):

    def __init__(self, corpusfile):

        # Iterate through the corpus once to build a lexicon
        generator = corpus_reader(corpusfile)
        self.lexicon = get_lexicon(generator)
        self.lexicon.add("UNK")
        self.lexicon.add("START")
        self.lexicon.add("STOP")

        # Now iterate through the corpus again and count ngrams
        generator = corpus_reader(corpusfile, self.lexicon)
        self.count_ngrams(generator)


    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
        self.bigramcounts = {}
        self.trigramcounts = {}

        ##Your code here


        for sentence in corpus:

            unigram_list = get_ngrams(sentence, 1)
            bigram_list = get_ngrams(sentence, 2)
            trigram_list = get_ngrams(sentence, 3)

            # bigram dictionary builder
            for bigram in bigram_list:

                # if key exists, add 1 to its count
                if (bigram in self.bigramcounts):
                    self.bigramcounts[bigram] += 1

                # else, set its count to 1
                else:
                    self.bigramcounts[bigram] = 1

            # unigram dictionary builder
            for unigram in unigram_list:
                # if key exists, add 1 to its count
                if (unigram in self.unigramcounts):
                    self.unigramcounts[unigram] += 1

                # else, set its count to 1
                else:
                    self.unigramcounts[unigram] = 1

            # trigram dictionary builder
            for trigram in trigram_list:

                # if key exists, add 1 to its count
                if (trigram in self.trigramcounts):
                    self.trigramcounts[trigram] += 1

                # else, set its count to 1
                else:
                    self.trigramcounts[trigram] = 1

        return

    def raw_trigram_probability(self, trigram):
        """
        COMPLETE THIS METHOD (PART 3)
        Returns the raw (unsmoothed) trigram probability
        """
        total_grams = sum(self.trigramcounts.values())

        #for key in self.trigramcounts:

            #count = self.trigramcounts[key]
            #total_grams += count

        if (trigram in self.trigramcounts):
            trigram_count = self.trigramcounts[trigram]

        else:
            trigram_count = 0

        raw_prob = trigram_count / total_grams

        return raw_prob

    def raw_bigram_probability(self, bigram):
        """
        COMPLETE THIS METHOD (PART 3)
        Returns the raw (unsmoothed) bigram probability
        """
        total_grams = sum(self.bigramcounts.values())

        # finds total num bigrams
        #for key in self.bigramcounts:

            #count = self.bigramcounts[key]
            #total_grams += count

        # finds bigram count
        if (bigram in self.bigramcounts):
            bigram_count = self.bigramcounts[bigram]

        else:
            bigram_count = 0

        raw_prob = bigram_count / total_grams

        return raw_prob

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

        #for key in self.unigramcounts:

            #count = self.unigramcounts[key]
            #total_grams += count

        if (unigram in self.unigramcounts):
            unigram_count = self.unigramcounts[unigram]

        else:
            unigram_count = 0

        raw_prob = unigram_count / total_grams

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

    def generate_sentence(self,t=20):
        """
        COMPLETE THIS METHOD (OPTIONAL)
        Generate a random sentence from the trigram model. t specifies the
        max length, but the sentence may be shorter if STOP is reached.
        """
        return result

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

        smoothed_prob = lambda1*(self.raw_unigram_probability(trigram[2]))

        return smoothed_prob

    def sentence_logprob(self, sentence):
        """
        COMPLETE THIS METHOD (PART 5)
        Returns the log probability of an entire sequence.
        """
        trigrams_list = get_ngrams(sentence, 3)

        log_probs_list = []

        for i in range(len(trigrams_list)):

            prob = math.log2(self.smoothed_trigram_probability(trigrams_lis
            log_probs_list.append(prob)

        log_prob = sum(log_probs_list)

        return log_prob

    def perplexity(self, corpus):
        """
        COMPLETE THIS METHOD (PART 6)
        Returns the log probability of an entire sequence.
        """
        sum_prob = 0
        total_words = 0

        for sequence in corpus:
            log_prob = self.sentence_logprob(sequence)
            sum_prob += log_prob
            total_words += len(sequence)

        l = sum_prob / total_words
        perplexity_val = pow(2,-1*l)

        return perplexity_val


def essay_scoring_experiment(training_file1, training_file2, testdir1, test

        model1 = TrigramModel(training_file1)
        model2 = TrigramModel(training_file2)

        total = 0
        correct = 0

        for f in os.listdir(testdir1):
            pp_1 = model1.perplexity(corpus_reader(os.path.join(testdir1, f
            pp_2 = model2.perplexity(corpus_reader(os.path.join(testdir1, f

            if (pp_1 < pp_2):
                correct += 1

            total += 1

        for f in os.listdir(testdir2):
            pp2 = model2.perplexity(corpus_reader(os.path.join(testdir2, f)
            pp1 = model1.perplexity(corpus_reader(os.path.join(testdir2, f)
            if (pp_2 < pp_1):
                correct += 1

            total += 1

        accuracy = correct / total

        return accuracy

if __name__ == "__main__":

    model = TrigramModel(sys.argv[1])

    # put test code here...
    # or run the script from the command line with
    # $ python -i trigram_model.py [corpus_file]
    # >>>
    #
    # you can then call methods on the model instance in the interactive
    # Python prompt.


    # Testing perplexity:
    dev_corpus = corpus_reader(sys.argv[2], model.lexicon)
    pp = model.perplexity(dev_corpus)
    print("Perplexity of the test corpus:", pp)


    # Essay scoring experiment:
    acc = essay_scoring_experiment("/Users/jay/Home/Courses/NLP/hw1/hw1_dat
                                   "/Users/jay/Home/Courses/NLP/hw1/hw1_dat
                                   "/Users/jay/Home/Courses/NLP/hw1/hw1_dat
                                   "/Users/jay/Home/Courses/NLP/hw1/hw1_dat

    print("Essay scoring accuracy:", acc)
```

```
-------------------------------------------------------------------------
--
FileNotFoundError                         Traceback (most recent call las
t)
<ipython-input-57-9c40bf0a8639> in <module>
    304 if __name__ == "__main__":
    305
--> 306     model = TrigramModel(sys.argv[1])
    307
    308     # put test code here...

<ipython-input-57-9c40bf0a8639> in __init__(self, corpusfile)
     68         # Iterate through the corpus once to build a lexicon
     69         generator = corpus_reader(corpusfile)
---> 70         self.lexicon = get_lexicon(generator)
     71         self.lexicon.add("UNK")
     72         self.lexicon.add("START")

<ipython-input-57-9c40bf0a8639> in get_lexicon(corpus)
     23 def get_lexicon(corpus):
     24     word_counts = defaultdict(int)
---> 25     for sentence in corpus:
```