NLP HW1

Manxueying Li, UNI:ml4529

Analytical Component:

Problem1

(i)

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

(ii)

	spam	ham	P(word|Spam)	P(word|Ham)
buy	1		0.08333333333	0
car	1	1	0.08333333333	0.1428571429
Nigeria	2	1	0.1666666667	0.1428571429
profit	2		0.1666666667	0
money	1	1	0.08333333333	0.1428571429
home	1	2	0.08333333333	0.2857142857
bank	2	1	0.1666666667	0.1428571429
check	1		0.08333333333	0
wire	1		0.08333333333	0
fly		1	0	0.1428571429
	12	7	1	1

(iii)

$$
\begin{aligned}
y_1 &= argmax_y P(y) \prod_i P(x_i|y) \\
&= \begin{cases} y_{spam} = P(Nigeria|Spam)P(Spam) = \frac{1}{10} \\ y_{ham} = P(Nigeria|Ham)P(Ham)) = \frac{2}{35} \end{cases} \\
&= spam
\end{aligned}
$$

Therefore, predicted label for “Nigeria” is Spam.

$$
\begin{aligned}
y_2 &= argmax_y P(y)\prod_i P(x_i|y)\\
&= \begin{cases} y_{spam} = P(Spam)P(Nigeria|Spam)P(home|Spam) = \frac{1}{10}\cdot\frac{1}{12} = 0.00833\\ y_{ham} = P(Ham)P(Nigeria|Ham)P(home|Ham) = \frac{2}{35}\cdot\frac{2}{7} = 0.016327) \end{cases}\\
&= ham
\end{aligned}
$$

Therefore, predicted label for “Nigeria hom”e is Ham.

$$
\begin{aligned}
y_3 &= argmax_y P(y)\prod_i P(x_i|y)\\
&= \begin{cases} y_{spam} = P(Spam)P(home|Spam)P(bank|Spam)P(money|Spam) = \frac{3}{5}\cdot\frac{1}{12}\cdot\frac{2}{12}\cdot\frac{1}{12} = 0.000694\\ y_{ham} = P(Ham)P(home|Ham)P(bank|Ham)P(money|Ham) = \frac{2}{5}\cdot\frac{1}{7}\cdot\frac{2}{7}\cdot\frac{1}{7} = 0.002332 \end{cases}\\
&= ham
\end{aligned}
$$

Therefore, predicted label for “home bank money” is Ham.

Problem2

$$
\begin{aligned}
\sum_{w_1,w_2,\ldots,w_n} P(w_1,w_2,\ldots,w_n) &= \sum_{w_1,w_2,\ldots,w_n} P(w_1|start)P(w_2|w_1)P(w_3,w_2)P(w_4,w_5)\ldots P(w_n|w_{n-1}) \text{ Chain Rule}\\
&\text{Summing over all possibility of } w_n\\
&= \sum_{w_n} P(w_n|w_{n-1}) \sum_{w_1,w_2,\ldots,w_{n-1}} P(w_1|start)P(w_2|w_1)P(w_3,w_2)P(w_4,w_5)\ldots P(w_{n-1}|w_{n-2})\\
&\text{Marginalize } w_n\\
&= 1\cdot \sum_{w_1,w_2,\ldots,w_{n-1}} P(w_1|start)P(w_2|w_1)P(w_3,w_2)P(w_4,w_5)\ldots P(w_{n-1}|w_{n-2})\\
&\text{Summing over all possibility of } w_{n-1}\\
&= \sum_{w_{n-1}} P(w_{n-1}|w_{n-2}) \sum_{w_1,w_2,\ldots,w_{n-2}} P(w_1|start)P(w_2|w_1)P(w_3,w_2)P(w_4,w_5)\ldots P(w_{n-2}|w_{n-3})\\
&\text{Marginalize } w_{n-1}\\
&= 1\cdot \sum_{w_1,w_2,\ldots,w_{n-2}} P(w_1|start)P(w_2|w_1)P(w_3,w_2)P(w_4,w_5)\ldots P(w_{n-1}|w_{n-3})\\
\\
&\text{do the same marginaliztion for the rest } w_i \in \{w_1,w_2,\ldots,w_{n-2}\}.\\
&\text{Since we sum over all possibility of every w, every term will become 1}\\
\\
&= 1
\end{aligned}
$$

Programming Component:

Part1

```
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).
    """
    n_grams = []
    starts = ['START'] * (n - 1)
    stop = ['STOP']
    sequence = starts + sequence + stop
    for i in range(len(sequence) - n + 1):
        n_grams.append(tuple(sequence[i:i + n]))
    return n_grams
```

Part2

```
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 = {}

        # Your code here
        unigramcounts = []
        bigramcounts = []
        trigramcounts = []

        for sentence in corpus:
            unigramcounts += get_ngrams(sentence, 1)

            bigramcounts += get_ngrams(sentence, 2)

            trigramcounts += get_ngrams(sentence, 3)

        self.unigramcounts = dict(collections.Counter(unigramcounts))
        self.bigramcounts = dict(collections.Counter(bigramcounts))
        self.trigramcounts = dict(collections.Counter(trigramcounts))

        self.unicnttotal = 0

```

```python
        for key, val in self.unigramcounts.items():
            self.unicnttotal += val

        self.unicnt = self.unicnttotal
```

part3

```python
def raw_trigram_probability(self, trigram):
        """
        COMPLETE THIS METHOD (PART 3)
        Returns the raw (unsmoothed) trigram probability
        """
        if trigram in self.trigramcounts and trigram[:-1] in self.bigramcounts:
            return self.trigramcounts[trigram] / self.bigramcounts[trigram[:-1]]
        else:
            return 0

    def raw_bigram_probability(self, bigram):
        """
        COMPLETE THIS METHOD (PART 3)
        Returns the raw (unsmoothed) bigram probability
        """
        if bigram in self.bigramcounts and bigram[0] in self.unigramcounts:
            return self.bigramcounts[bigram] / self.unigramcounts[bigram[0]]
        else:
            return 0

    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 in self.unigramcounts:
            return self.unigramcounts[unigram] / self.unicnttotal
        else:
            return 0
```

part4

```
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
        return lambda1 * self.raw_trigram_probability(trigram)\
            + lambda2 * self.raw_bigram_probability(trigram[1:])\
            + lambda3 * self.raw_unigram_probability(trigram[-1])
```

part5

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

part6

```
def perplexity(self, corpus):
        """
        COMPLETE THIS METHOD (PART 6)
        Returns the log probability of an entire sequence.
        """
        summ = 0
        for si in corpus:
            summ += self.sentence_logprob(si)
        return 2 ** (-summ / self.unicnt)
```

part7

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

    model1 = TrigramModel(training_file1)#high
    model2 = TrigramModel(training_file2)#low

    total = 0
    correct = 0

```

```python
    #high
    for f in os.listdir(testdir1):
        pp1 = model1.perplexity(corpus_reader(
            os.path.join(testdir1, f), model1.lexicon))
        # ..
        pp2 = model2.perplexity(corpus_reader(
            os.path.join(testdir1, f), model2.lexicon))
        if pp1 < pp2:
            correct += 1
        total += 1
    #low
    for f in os.listdir(testdir2):
        pp2 = model2.perplexity(corpus_reader(
            os.path.join(testdir2, f), model2.lexicon))
        # ..
        pp1 = model1.perplexity(corpus_reader(
            os.path.join(testdir2, f), model1.lexicon))
        if pp1 > pp2:
            correct += 1
        total += 1

    return correct / total
```