

# 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		0.083333333333	0
car	1	1	0.083333333333	0.1428571429
Nigeria	2	1	0.1666666667	0.1428571429
profit	2		0.1666666667	0
money	1	1	0.083333333333	0.1428571429
home	1	2	0.083333333333	0.2857142857
bank	2	1	0.1666666667	0.1428571429
check	1		0.083333333333	0
wire	1		0.083333333333	0
fly		1	0	0.1428571429
	12	7	1	1

(iii)

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

Therefore, predicted label for "Nigeria" is Spam.

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

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

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

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

## Problem2

$$\sum_{w_1, w_2, \dots, w_n} P(w_1, w_2, \dots, w_n) = \sum_{w_1, w_2, \dots, w_n} P(w_1|start)P(w_2|w_1)P(w_3, w_2)P(w_4, w_5) \dots P(w_n|w_{n-1}) \text{ Chain Rule}$$

Summing over all possibility of  $w_n$

$$= \sum_{w_n} P(w_n|w_{n-1}) \sum_{w_1, w_2, \dots, w_{n-1}} P(w_1|start)P(w_2|w_1)P(w_3, w_2)P(w_4, w_5) \dots P(w_{n-1}|w_{n-2})$$

Marginalize  $w_n$

$$= 1 \cdot \sum_{w_1, w_2, \dots, w_{n-1}} P(w_1|start)P(w_2|w_1)P(w_3, w_2)P(w_4, w_5) \dots P(w_{n-1}|w_{n-2})$$

Summing over all possibility of  $w_{n-1}$

$$= \sum_{w_{n-1}} P(w_{n-1}|w_{n-2}) \sum_{w_1, w_2, \dots, w_{n-2}} P(w_1|start)P(w_2|w_1)P(w_3, w_2)P(w_4, w_5) \dots P(w_{n-2}|w_{n-3})$$

Marginalize  $w_{n-1}$

$$= 1 \cdot \sum_{w_1, w_2, \dots, w_{n-2}} P(w_1|start)P(w_2|w_1)P(w_3, w_2)P(w_4, w_5) \dots P(w_{n-1}|w_{n-3})$$

do the same marginalization for the rest  $w_i \in \{w_1, w_2, \dots, w_{n-2}\}$ .

Since we sum over all possibility of every w, every term will become 1

$$= 1$$

## Programming Component:

### Part1

```
1 def get_ngrams(sequence, n):
2     """
3     COMPLETE THIS FUNCTION (PART 1)
4     Given a sequence, this function should return a list of n-grams,
5     where each n-gram is a Python tuple.
6     This should work for arbitrary values of 1 <= n < len(sequence).
7     """
8     n_grams = []
9     starts = ['START'] * (n - 1)
10    stop = ['STOP']
11    sequence = starts + sequence + stop
12    for i in range(len(sequence) - n + 1):
13        n_grams.append(tuple(sequence[i:i + n]))
14    return n_grams
```

### Part2

```
1 def count_ngrams(self, corpus):
2     """
3     COMPLETE THIS METHOD (PART 2)
4     Given a corpus iterator, populate dictionaries of unigram, bigram,
5     and trigram counts.
6     """
7
8     self.unigramcounts = {} # might want to use defaultdict or Counter instead
9     self.bigramcounts = {}
10    self.trigramcounts = {}
11
12    # Your code here
13    unigramcounts = []
14    bigramcounts = []
15    trigramcounts = []
16
17    for sentence in corpus:
18        unigramcounts += get_ngrams(sentence, 1)
19
20        bigramcounts += get_ngrams(sentence, 2)
21
22        trigramcounts += get_ngrams(sentence, 3)
23
24    self.unigramcounts = dict(collections.Counter(unigramcounts))
25    self.bigramcounts = dict(collections.Counter(bigramcounts))
26    self.trigramcounts = dict(collections.Counter(trigramcounts))
27
28    self.unicnttotal = 0
```

```
29         for key, val in self.unigramcounts.items():
30             self.unicnttotal += val
31
```

## part3

```
1  def raw_trigram_probability(self, trigram):
2      """
3      COMPLETE THIS METHOD (PART 3)
4      Returns the raw (unsmoothed) trigram probability
5      """
6      if trigram in self.trigramcounts and trigram[:-1] in self.bigramcounts:
7          return self.trigramcounts[trigram] / self.bigramcounts[trigram[:-1]]
8      else:
9          return 0
10
11 def raw_bigram_probability(self, bigram):
12     """
13     COMPLETE THIS METHOD (PART 3)
14     Returns the raw (unsmoothed) bigram probability
15     """
16     if bigram in self.bigramcounts and bigram[0] in self.unigramcounts:
17         return self.bigramcounts[bigram] / self.unigramcounts[bigram[0]]
18     else:
19         return 0
20
21 def raw_unigram_probability(self, unigram):
22     """
23     COMPLETE THIS METHOD (PART 3)
24     Returns the raw (unsmoothed) unigram probability.
25     """
26
27     # hint: recomputing the denominator every time the method is called
28     # can be slow! You might want to compute the total number of words once,
29     # store in the TrigramModel instance, and then re-use it.
30     if unigram in self.unigramcounts:
31         return self.unigramcounts[unigram] / self.unicnttotal
32     else:
33         return 0
```

## part4

```

1 def smoothed_trigram_probability(self, trigram):
2     """
3     COMPLETE THIS METHOD (PART 4)
4     Returns the smoothed trigram probability (using linear interpolation).
5     """
6     lambda1 = 1 / 3.0
7     lambda2 = 1 / 3.0
8     lambda3 = 1 / 3.0
9     return lambda1 * self.raw_trigram_probability(trigram)\
10         + lambda2 * self.raw_bigram_probability(trigram[1:])\
11         + lambda3 * self.raw_unigram_probability(trigram[-1])

```

## part5

```

1 def sentence_logprob(self, sentence):
2     """
3     COMPLETE THIS METHOD (PART 5)
4     Returns the log probability of an entire sequence.
5     """
6     trigrams = get_ngrams(sentence, 3)
7     logprob = 0
8     for i in trigrams:
9         if self.smoothed_trigram_probability(i):
10             logprob += math.log2(self.smoothed_trigram_probability(i))
11     return logprob

```

## part6

```

1 def perplexity(self, corpus):
2     """
3     COMPLETE THIS METHOD (PART 6)
4     Returns the log probability of an entire sequence.
5     """
6     summ = 0
7     for si in corpus:
8         summ += self.sentence_logprob(si)
9     return 2 ** (-summ / self.unicnttotal)

```

## part7

```

1 def essay_scoring_experiment(training_file1, training_file2, testdir1, testdir2):
2
3     model1 = TrigramModel(training_file1)#high
4     model2 = TrigramModel(training_file2)#low
5
6     total = 0
7     correct = 0
8

```

```
9      #high
10     for f in os.listdir(testdir1):
11         pp1 = model1.perplexity(corpus_reader(
12             os.path.join(testdir1, f), model1.lexicon))
13         # ..
14         pp2 = model2.perplexity(corpus_reader(
15             os.path.join(testdir1, f), model2.lexicon))
16         if pp1 < pp2:
17             correct += 1
18         total += 1
19
20     #low
21     for f in os.listdir(testdir2):
22         pp2 = model2.perplexity(corpus_reader(
23             os.path.join(testdir2, f), model2.lexicon))
24         # ..
25         pp1 = model1.perplexity(corpus_reader(
26             os.path.join(testdir2, f), model1.lexicon))
27         if pp1 > pp2:
28             correct += 1
29         total += 1
30
31     return correct / total
```