Homework #3: CMPT-413

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Only submit answers for questions marked with †.

Important! To solve this homework you must read the following chapters of the NLTK Book, available online at *http://www.nltk.org/book*

- Chapter 5. Categorizing and Tagging Words
- Chapter 6. Learning to Classify Text

(1) Write regular expressions to match the following classes of strings:

  a. A single determiner (assume that a, an, and the are the only determiners).

  b. An arithmetic expression using integers, addition, and multiplication, such as 2*3+8.

(2) Using re.findall(), write a regular expression which will extract pairs of values of the form login name, email domain from the following string:

```
>>> str = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu  (internet)  hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```

(3) The following code prints out the first tagged sentence from the Brown corpus only selecting the news reports genre (collected from various U.S. newspapers in 1961).

```
import nltk
print nltk.corpus.brown.tagged_sents(categories='news')[0]
```

The following code prints the first tagged sentence from the Brown corpus section only selecting the news editorial genre.

```
import nltk
print nltk.corpus.brown.tagged_sents(categories='editorial')[0]
```

Write a Python program to print out only the part of speech (pos) tag sequences for the first five sentences from the Brown corpus news reports section. Try to write it as a readable one-line program (not counting import statements).

(4) NLTK provides documentation for each tag, which can be queried using the tag, e.g. `nltk.help.brown_tagset('RB')`, or a regular expression, e.g. `nltk.help.brown_tagset('NN.*')`. The Brown corpus manual available from `http://khnt.hit.uib.no/icame/manuals/brown/INDEX.HTM` Print out the 10 most frequent words that are tagged as proper nouns in the entire tagged Brown corpus.

(5) The following code prints out the most probable tag for the word *run* using the probability Pr(t | run). It also prints out the probability for the most probable tag.

```
from nltk.corpus import brown
from nltk.probability import *
cfd = ConditionalFreqDist()
for sent in brown.tagged_sents():
    for word, tag in sent:
        if word == 'run':
            cfd['run'].inc(tag)
# use the maximum likelihood estimate MLEProbDist to create
# a probability distribution from the observed frequencies
cpd = ConditionalProbDist(cfd, MLEProbDist)
# find the tag with the highest probability
tag = cpd['run'].max()
# cfd['run'].B() reports the number of distinct tags seen with 'run'
# cfd['run'].N() reports the total number of ('run', tag) observations
print tag, 'run', cpd['run'].prob(tag), cfd['run'].B(), cfd['run'].N()
```

There are many noun pos tags, for example, pos tags like `NN, NN$, NP, NPS, ...`; the most common of these have `$` for possessive nouns, `S` for plural nouns (since plural nouns typically end in s), `P` for proper nouns.

For each noun pos tag, print out the most probable word for that tag using the conditional probability Pr($w \mid t$) for noun pos tag t and word w. Print out the noun pos tag t, the word with the highest value for Pr($w \mid t$) and the probability.

(6) Write down regular expressions that can be used to match some part of an input word (e.g. capitalization, suffix of a certain kind, etc.) and provide a pos tag for that word. Use the nltk.RegexpTagger package in order to implement a part of speech (POS) tagger using your regular expressions. Provide the Python program that prints out the accuracy of your POS tagger on news section of the Brown corpus.

Note that before you can start answering this question you will need to read Chapter 5 of the NLTK book which explains how to write the code for POS tagging.

(7) † This question deals with tagging words in a sentence with their parts of speech, e.g. is the word 'can' a verb or a noun in a particular sentence. We will use the Brown corpus and the Brown tagset (use `nltk.help.brown_tagset('.*')` to find out more).
The usage for your program is given below:

```
usage: brown_tagger.py -h -i trainsection -o testsection -m method

    -h help
    -i training section ,e.g. 'news' or 'editorial'
    -o test section ,e.g. 'news' or 'editorial'
    -m method, e.g. 'default', 'regexp', 'lookup',
                    'simple_backoff', 'unigram', 'bigram', 'trigram'

    Do not type in the single quotes at the command line.
```

The training and test data is assumed to come from the Brown corpus, and we provide the section name as the command line argument `-i` or `-o`. The various methods you need to implement are explained below:

default Assign the most frequent tag in the training data to each word.

regexp Write down a regular expression tagger for the most frequent tags using the instructions in the NLTK book. If a regular expression matches a word it is tagged with a specific part of speech tag.

lookup For the 1000 most frequent words in training, store the most frequent tag *per word.* A Python dictionary `table` which returns a tag given a word can be used with:
`nltk.UnigramTagger(model=table)` to create a lookup tagger.

simple_backoff Implement the following backoff strategy: `lookup` → `regexp` → `default`.

unigram Use the word, tag counts collected from training to train a `nltk.UnigramTagger`. Backoff to `default`.

bigram Train a bigram tagger using `nltk.BigramTagger`. Backoff to `unigram`.

trigram Train a trigram tagger using `nltk.TrigramTagger`. Backoff to `bigram`.

(8) † **Smoothing *n*-grams**

For this question we will build bigram model of part of speech (pos) tag sequences. We will ignore the words in the sentence and only use the pos tags associated with each word in the Brown corpus. The following code prints out bigrams of pos tags for each sentence in the news reports section of the Brown corpus:

```
from nltk.corpus import brown
for sent in brown.tagged_sents(categories='news'):
    # print out the pos tag sequence for this sentence
    print " ".join([t[1] for t in sent])
    p = [(None, None)] # empty token/tag pair
    bigrams = zip(p+sent, sent+p)
    for (a,b) in bigrams:
        history = a[1]
        current_tag = b[1]
        print current_tag, history    # print each bigram
    print
```

Note that we introduce an extra pos tag called *None* to start the sentence, and an extra pos tag called *None* to end the sentence. So the tag sequence for the sentence s_i of length $n+1$ will be
$None, t_0, t_1, \ldots, t_n, None$.

Extend the above program and compute the probability $p(t_i \mid t_{i-1})$ for all observed pos tag bigrams (t_{i-1}, t_i). Use the NLTK functions that you used in question (5). Note that unobserved bigrams will get probability of zero. Once we have this bigram probability model, we can compute the probability of any sentence s of length $n+1$ to be:

$$\begin{aligned} P(s) &= p(t_0 \mid t_{-1}) \cdot p(t_1 \mid t_0) \cdot \ldots \cdot p(t_n \mid t_{n-1}) \cdot p(t_{n+1} \mid t_n) \\ &= \prod_{i=0}^{n+1} p(t_i \mid t_{i-1}) \end{aligned} \tag{1}$$

where, $t_{-1} = t_{n+1} = None$.

Let $T = s_0, \ldots, s_m$ represent the test data (data which was not used to create the bigram probability model) with sentences s_0 through s_m.

$$P(T) = \prod_{i=0}^{m} P(s_i) = 2^{\sum_{i=0}^{m} \log_2 P(s_i)}$$

$$\log_2 P(T) = \sum_{i=0}^{m} \log_2 P(s_i)$$

where $\log_2 P(s_i)$ is the log probability assigned by the bigram model to the sentence s_i using equation (1). Let W_T be the length of the text T measured in part of speech tags. The *cross entropy* for T is:

$$H(T) = -\frac{1}{W_T} \log_2 P(T)$$

The cross entropy corresponds to the average number of bits needed to encode each of the W_T words in the *test data*. The *perplexity* of the test data T is defined as:

$$PP(T) = 2^{H(T)}$$

Some other things you should use in your program so that you can match the reference perplexity numbers from the testcases:

- Remember to use `from __future__ import division`
- Use log base 2 using the numpy function log2: `from numpy import log2`.
- In some cases your program will attempt to take a log of probability 0.0 (e.g. when an unseen n-gram has probability 0.0). In these cases, instead of log(0.0) use the value `_NINF` defined as:

  ```
  _NINF = float('-1e300')
  ```

- Use the following definition of perplexity, where H stands for the cross entropy.

  ```
  def perplexity(H):
      try: val = pow(2,H)
      except OverflowError: return 'Inf'
      return "%lf" % (val)
  ```

- Use the following function `logsum` to sum the log values for n-grams collected for a given corpus.

  ```
  def logsum(values):
      sum = 0.0
      for i in values:
          if i == _NINF: return _NINF
          sum += i
      if sum < _NINF: return _NINF
      return sum
  ```

- The training and test data should be loaded using the following commands:

  ```
  train = brown.tagged_sents(categories=trainsection)
  test = islice(brown.tagged_sents(categories=testsection), 300)
  ```

  The variables `trainsection` and `testsection` will be set using the command line.

The usage for your program is given below:

```
usage: answer/smoothing.py -h -i trainsection -o testsection -m method

    -h help
    -i training section ,e.g. 'news' or 'editorial'
    -o test section ,e.g. 'news' or 'editorial'
    -m method, e.g. 'no_smoothing', 'interpolation', 'add_one'
    -l lambda_vector, e.g. "0.5:0.3:0.2"
          for values of \lambda_1, \lambda_2 and \lambda_3.
          It must have 3 elements and sum to 1.0 (only used for interpolation)

    Do not type in the single quotes at the command line.
```

The training and test data is assumed to come from the Brown corpus, and we provide the section name as the command line argument `-i` or `-o`. The various methods you need to implement are explained below:

no_smoothing Provide a Python program that trains a bigram probability model on the training data and then prints out the cross-entropy and perplexity for the training data and test data. No smoothing so unseen events in test should get zero probability.

add_one Implement add-one smoothing to provide counts for every possible bigram (t_{i-1}, t_i) using the NLTK functions `ConditionalProbDist` (pay special attention to the bins argument) and `LaplaceProbDist`.

interpolation Implement the following Jelinek-Mercer style *interpolation* smoothing model:

$$P_{interp}(t_i \mid t_{i-1}) = \lambda_1 P(t_i \mid t_{i-1}) + \lambda_2 P(t_i) + \lambda_3 \frac{1}{|V| + 1}$$

$|V|$ is the size of the vocabulary in the training set. The +1 indicates that we can use this for computing the probability of unknown words (which are assumed to be a single word type UNK). The equation provides a probability only when $\lambda_1 + \lambda_2 + \lambda_3 = 1.0$. You will have to estimate a new unigram probability model from the training data. Do not do add-one smoothing on the bigram or unigram model.

(9) **Hidden Markov models for Decipherment**

Suppose you are given a large amount of text in a language you cannot read (and perhaps you are not even sure that it *is* a language). Faced with a total lack of knowledge about the language and the script used to transcribe it, you try to see if you can discover some very basic knowledge using unsupervised learning. Perhaps it is worth trying to see if the letters in the script can be clustered into meaningful groups.

Hidden Markov models (HMMs) are particularly suited for such a task. The hidden states correspond to the meaningful clusters we hope to find, and the observations are the characters of the text. The text provided to you is taken from the Brown corpus (which is in English, but we will try to "decipher" it anyway). Let us take an HMM with two hidden states, and the observations are to be taken from the list of 26 lowercase English characters plus one extra character for a single space character that separates the words in the text.

We will use the software `umdhmm` which implements the Baum-Welch algorithm for unsupervised training of HMMs. It has been installed in `/usr/shared/CMPT/cmpt413/sw/umdhmm-v1.02` on CSIL Linux machines. You can also install the software yourself. The web page for this software is:
`http://www.kanungo.com/software/software.html`

You can use the `esthmm` program to estimate the parameters of an HMM based on maximizing the likelihood of a training data sequence. The parameters of an HMM with n states and m observation symbols are: the probability of moving from one state to another (an n by n matrix A), the probability of producing a symbol from a state (an n by m matrix B), and the probability of starting a sequence from a state (a vector of size n).

In the notation used by `umdhmm` the parameters for $m = 2$ observation symbols and $n = 3$ states is:

```
M= 2
N= 3
A:
0.333 0.333 0.333
0.333 0.333 0.333
0.333 0.333 0.333
B:
0.5   0.5
0.75  0.25
0.25  0.75
pi:
0.333 0.333 0.333
```

If you save the above format as a text file called `init.hmm` then you can train a HMM using the command `esthmm -v -I init.hmm`. If you don't initialize esthmm, it takes random values for A, B and *pi*.

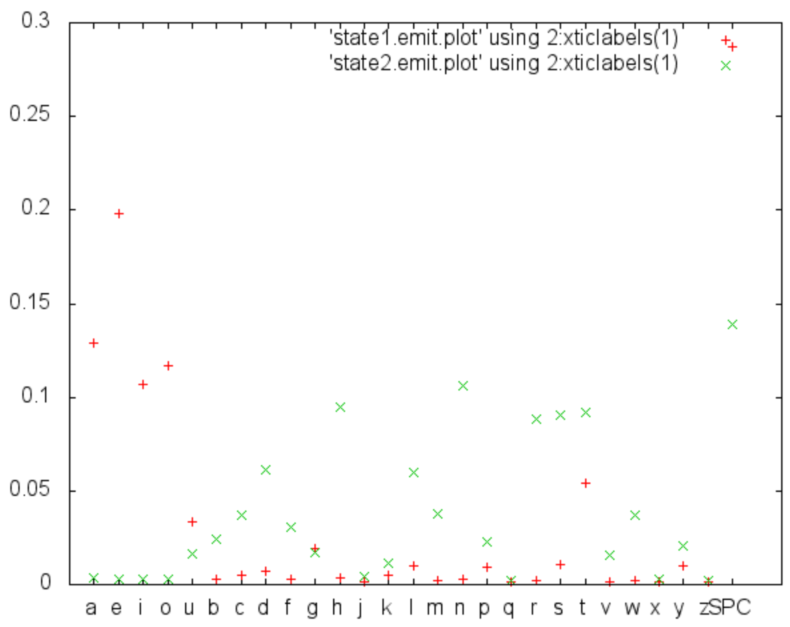

Figure 1: Output graph showing emission distribution in the HMM states after training.

You are provided the training data from the Brown corpus and it is already in the correct format for the `esthmm` program. The file name for the training data is `brownchars_sf.txt` and it contains the first 50000 characters from the science fiction section of the Brown corpus. In this file, the numbers 1 to 26 are an index that specify the ASCII characters 'a' to 'z'. The number 27 is mapped to the space character, and in this data a single space character is used to separate words.

The above `init.hmm` is just an example, you have to think about the number of states required for this question (you need a state that represents vowels and another to represent consonants) and also the emissions from each state (it should be the observations which are limited to 27 character types).

While you do not need to implement the unsupervised training of an HMM for this question, you do need to interpret the HMM that is learned.

a. Use the training corpus `brownchars_sf.txt` to train an HMM using `esthmm`. Provide the HMM that is learned. Try different initializations and see if the learned HMM is different.

b. With careful initialization of the HMM, it is possible to train the HMM to automatically distinguish vowels (let us coarsely define these to be the ASCII characters *a, e, i, o, u*) and consonants (everything else) with the space character being in neither set. Provide the PNG file of a graph that clearly shows (visually) that the trained HMM has learned to distinguish one group (vowels) from the other (consonants). For example, the graph in Fig. 1 shows how the HMM has learned how to distinguish vowels from consonants: state 1 has a higher probability to emit a vowel, when compared with state 2 which has a higher probability to emit a consonant. It isn't perfect: consider the case of the consonant 'g'. Your HMM might look very different since HMMs are very sensitive to their initialization, but make sure that the HMM can mostly classify vowels differently from consonants in the emit distributions of the two states.

c. Experiment with models with 12 hidden states. What distinctions are (or can be) made by a trained HMM with 12 states that could not be made with the HMM with 2 states? Is it better to use 2 or 12 states for this task?