# Homework #3: CMPT-413

Reading: NLTK Tutorial Chp 4;
*http://nltk.org/doc/en/tag.html*;
Distributed on Feb 18; due on Mar 3

Anoop Sarkar – anoop@cs.sfu.ca

Only submit answers for questions marked with †. For the questions marked with a ††; choose one of them and submit its answer.

(1) The following code prints out the first sentence from the Brown corpus Section A (consisting of articles from the Press: Reportage genre). The Brown corpus Section A consists of news articles collected from various newspapers in 1961.

```
import nltk
print nltk.corpus.brown.tagged_sents('a')[1]
```

Print out only the part of speech (pos) tag sequences for the first ten sentences from the Brown corpus Section A. For example, the pos tag sequence for the fifth sentence is:

```
AT NN VBD PPS DOD VB CS AP IN NP$ NN CC NN NNS `` BER JJ CC JJ CC RB JJ '' .
```

(2) The names of the various pos tags might appear mysterious at this point. Look up the section called *List of Tags* from 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 (of all types).

(3) † 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($w$ | $t$) for noun pos tag $t$ and word $w$. Print out the noun pos tag $t$, the word with the highest value for Pr($w$ | $t$) and the probability.

Here are some randomly chosen noun pos tags with their most probable word and associated probability:

```
NN time 0.0101462582803
NNS years 0.0170386499728
NP Mr. 0.0241327300151
NP$ God's 0.0148148148148
NR home 0.19220945083
NR-NC Tuesday 0.75
NRS Sundays 0.5625
```

(4) 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 pos tagger using your regular expressions. Provide the Python program that prints out the accuracy of your pos tagger on Section A of the Brown corpus.

Note that before you can start answering this question you will need to read Chapter 4 of the NLTK tutorial which explains how to write the code for pos tagging.

(5) † Use Section A of the Brown corpus as training data to create various taggers, e.g. Trigram tagger, Bigram tagger, Unigram tagger, a Regexp tagger, a Lookup tagger (tags a token with the most frequently observed pos tag for that word), a Default tagger (tags everything as `NN`), etc.

Note that before you can start answering this question you will need to read Chapter 4 of the NLTK tutorial which explains how to write the code for pos tagging.

The NLTK implementation of these various taggers allows the following strategy:

1. Try tagging with the most accurate tagger you have.
2. If it was unable to find a tag for some token, try the next most accurate tagger using the `backoff` parameter when constructing the tagger.
3. Continue this process of backing off until the Default tagger (the Default tagger does not allow further backoff).

Create a tagger using this strategy on Section A of Brown. Try to build the most accurate tagger you can. Test your accuracy on Section B of Brown. Provide the Python program that trains on Section A of Brown and then prints out the accuracy of your pos tagger on Section B of Brown.

(6) † **Smoothing *n*-grams** (description updated on Feb 23, 2008)

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 Section A of the Brown corpus:

```
from nltk.corpus import brown
for sent in brown.tagged_sents('a'):
    # 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 (3). 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:

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

In the following we will refer to *training data* which is defined to be the pos tag sequences from Section A of the Brown corpus, and *test data* which is defined to be the pos tag sequences from the first 300 sentences of Section B of the Brown corpus (cf. question (1) in this homework).
In some cases your program will attempt to take a log of probability 0 (e.g. when an unseen n-gram has probability 0). In these cases, instead of log(0) use the value `_LOG_NINF` defined as:

```
_NINF = float('1e-300')
_LOG_NINF = log(_NINF,2)
```

a. 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.

   On the test data, when a bigram in unseen, the probability for that bigram is zero. Use `_LOG_NINF` when a bigram $(t_{i-1}, t_i)$ is unseen.

   Remember that cross entropy and perplexity are both positive real numbers, and the lower the values, the better the model over the test data.

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

$$P_{interp}(t_i \mid t_{i-1}) = \lambda P(t_i \mid t_{i-1}) + (1 - \lambda) P(t_i)$$

   Note that you will have to estimate a new unigram probability model from the training data.

   Set $\lambda = 0.8$ and using $P_{interp}$ and print out the cross-entropy and perplexity for the training data and test data. After you run the program for $\lambda = 0.8$, find a value for $\lambda$ that results in a better model of the test data and provide the output of your program that implements interpolation smoothing using this better $\lambda$ value and print out the cross-entropy and perplexity values.

   Use the simplifying assumption that if the unigram $t_i$ is unseen then $\log_2 P(t_i) =$ `_LOG_NINF`.

c. Implement add-one smoothing to provide counts for every possible bigram $(t_{i-1}, t_i)$. Recompute and print out the cross-entropy and perplexity for the training data and the test data. Do not smooth the unigram model $P(t_i)$.

d. After you have done *both* question (6b) and (6c), reduce test set perplexity even further by using add-one smoothing to augment the interpolation model.

(7) †† **(Machine) Translation**

NASA's latest mission to Mars has found some strange tablets. One tablet seems to be a kind of Rosetta stone which has translations from a language we will call MARTIAN-A (sentences 1a to 12a below) to another language we will call MARTIAN-B (sentences 1b to 12b below). The ASCII transcription of the alien script on the Rosetta tablet is given below:

```
1a. ok'sifar zvau hu .

1b. at'sifar somuds geyu .

2a. ok'anko ok'sifar myi pell hu .

2b. at'anko at'sifar ashi erder geyu .

3a. oprashyo hu qebb yuzvo oxloyzo .

3b. diza geyu isvat iwla pown .

4a. ok'sifar myi rig bzayr zu .

4b. at'sifar keerat ashi parq up .

5a. yux druh qebb stovokor .

5b. diza viodaws pai shun .

6a. ked hu qebb zu stovokor .

6b. dimbe geyu keerat pai shun .

7a. ked druh zvau ked hu qebb pnah .

7b. dimbe viodaws somuds dimbe geyu iwla woq .

8a. ked bzayr myi pell eoq .

8b. gakh up ashi erder kvig .

9a. yux eoq qebb zada ok'nefos .

9b. diza kvig pai goli at'nefos .

10a. ked amn eoq kin oxloyzo hom .

10b. dimbe kvig baz iluh ejuo pown .

11a. ked eoq tazih yuzvo kin dabal'ok .

11b. dimbe kvig isvat iluh dabal'at .

12a. ked mina eoq qebb yuzvo amn .

12b. dimbe kvig zeg isvat iwla baz .
```

Due to severe budget cutbacks at NASA, decryption of these tablets has fallen to Canadian undergraduate students, namely you. You can choose to write Python code to solve the problems or do it by hand – it's up to you.

a. Use the above translations to produce a translation dictionary. For each word in MARTIAN-A provide an equivalent word in MARTIAN-B. Provide any Python code used *and* the translation dictionary as a text file with MARTIAN-A words in column one and MARTIAN-B words in column two. If a word in MARTIAN-A has no equivalent in MARTIAN-B then put the entry "(none)" in column two.

b. Using your translation dictionary, provide a word for word translation for the following MARTIAN-B sentences on a new tablet which was found near the Rosetta tablet.

```
13b. gakh up ashi woq pown goli at'nefos .

14b. diza kvig zeg isvat iluh ejuo .

15b. dimbe geyu pai shun hunslob at'anko .
```

The MARTIAN-A sentences you produce will probably appear to be in a different word order from the MARTIAN-A sentences you observed on the Rosetta tablet. Some words might be unseen and so seemingly untranslatable. In those cases insert the word `?` for the unseen word.

Provide any Python code used and the produced MARTIAN-A translation in a text file.

c. The word for word translation can be improved with additional knowledge about MARTIAN-A word order. Luckily another tablet containing some MARTIAN-A sentences (untranslated) was found on the dusty plains of Mars. Use these MARTIAN-A sentences in order to find the most plausible word order for the MARTIAN-A sentences translated from MARTIAN-B sentences in (7b).

```
ok'anko myi oxloyzo druh .
yux mina eoq esky oxloyzo pnah .
ok'anko yolk stovokor koos oprashyo pnah zada ok'nefos yun zu kin hom .
ked hom qebb koos ok'anko .
ok'sifar zvau hu .
ok'anko ok'sifar
myi pell hu .
oprashyo hu qebb yuzvo oxloyzo .
ok'sifar myi rig bzayr zu .
yux druh qebb stovokor .
ked hu qebb zu stovokor .
ked bzayr myi pell eoq .
ked druh zvau ked hu qebb pnah .
yux eoq qebb zada ok'nefos .
ked amn eoq kin oxloyzo hom .
ked eoq tazih yuzvo kin dabal'ok .
ked mina eoq qebb yuzvo amn .
```

Using this additional MARTIAN-A text you can even find a translation for words that are missing from the translation dictionary (although this might be hard to implement in a program, cases that were previously translated as `?` can be translated by manual inspection of the above MARTIAN-A text).

Provide any Python code used and the revised MARTIAN-A translation in a text file.

(8) †† **Machine Translation** (description updated on Feb 23, 2008)

The following pseudo-code provides an algorithm that can learn a translation probability distribution $t(e|f)$ from a set of previously translated sentences. $t(e|f)$ is the probability of translating a given word $f$ in the source language as the word $e$ in the target language.

Implement the psuedo-code as a Python program and apply it to solve Question 7 (please read the description below on finding the most likely Martian-A sentence for a given Martian-B sentence).

```
initialize t(e|f) uniformly
do
    set c(e|f) = 0 for all words e, f
    set total(f) = 0 for all f
    for all sentence pairs (e_s, f_s) in the given translations
        for all word types e in e_s
            n_e = count of e in e_s
            total_s = 0
            for all word types f in f_s
                total_s += t(e|f) · n_e
            for all word types f in f_s
                n_f = count of f in f_s
                rhs = t(e|f) · n_e · n_f / total_s
                c(e|f) += rhs
                total(f) += rhs
        for each f, e
            t(e|f) = c(e|f) / total(f)
until convergence (usually 10-13 iterations)
```

By initializing uniformly, we are stating that each target word $e$ is equally likely to be a translation for given word $f$. Check for convergence by checking if the values for $t(e|f)$ for each $e, f$ do not change much (difference from previous iteration is less than $10^{-4}$, for example).

Once you have the $t(e|f)$ provided by the pseudo-code above you can attempt to solve Q. 7:

a. In Q. 7a you need to find a translation dictionary. To solve this question, you should find a word $e^*$ in Martian-A where $e^* = \text{argmax}_e\, t(e|f)$ for each word $f$ in Martian-B and insert $(e^*, f)$ into your translation dictionary. In cases where a translation does not exist, insert the word ? as the unseen word.

b. Q. 7b asks you to provide a Martian-A translation for given Martian-B sentences. To solve this question, you should assume that the Martian-A translation has the same number of words as the input Martian-B sentence. Let us refer to the input Martian-B sentence as $f_1, f_2, \ldots, f_n$. Then your output Martian-A sentence should be $e_1, e_2, \ldots, e_n$ where each $e_i = \text{argmax}_e\, t(e|f_i)$. In cases where a translation does not exist, insert the word ? as the unseen word into your translated sentence.

c. Q. 7c provides some additional Martian-A sentences. Using this as a language model to improve the translation output in general is beyond the scope of this homework. But you can use these extra Martian-A sentences to deal with cases in Q. 7b where you had to insert word ? as the unseen word. In your translated Martian-A output, check if there is a word $w_1$ that occurs before your unseen word ?. Find the word $w_2$ from the provided Martian-A sentences such that $w_2 = \text{argmax}_w\, p(w|w_1)$ where $p(w_2|w_1)$ is the probability of the bigram $w_1, w_2$. This word $w_2$ can be a good guess for the previously unknown word ?.

The psuedo-code given above is a very simple statistical machine translation model that is called IBM Model 1. More details about how this model works, and the justification for the algorithm is given in the Kevin Knight statistical machine translation workbook (available on the course web page).