Homework #3: CMPT-413

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

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

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:

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

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 `~anoop/cmpt413/sw/linux/umdhmm-v1.02` on fraser.sfu.ca (you will have to copy this directory using scp and use it on a CSIL Linux machine). 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*.

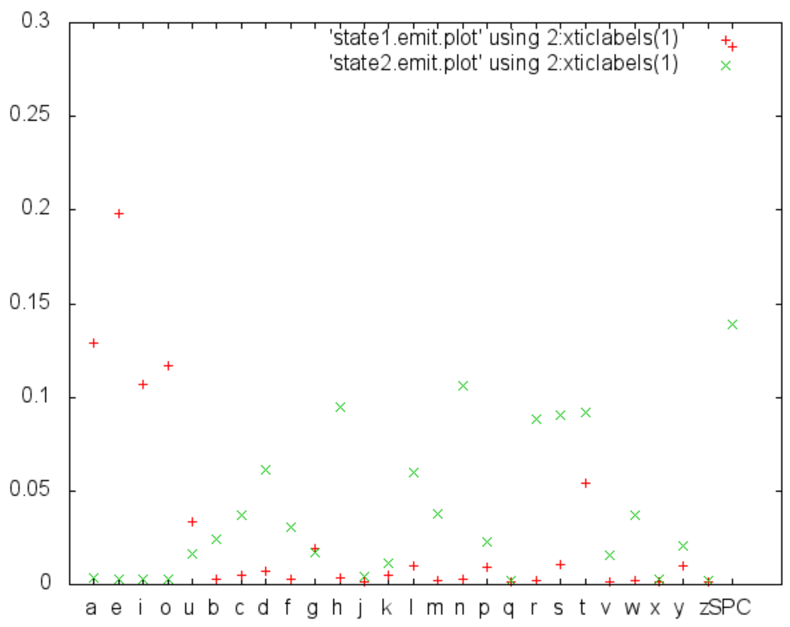

Figure 1: Output graph showing emission distribution in the HMM states after training.

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. *(this sub-part is optional; no extra marks)* 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?

(10) †† **Prepositional phrase attachment ambiguity resolution**

Consider the sentence *Calvin saw the man with the telescope* which has two meanings, one in which Calvin sees a man carrying a telescope and another in which Calvin using a telescope sees the man. This ambiguity between the *noun-attachment* versus the *verb-attachment* is called prepositional phrase attachment ambiguity, or *PP-attachment ambiguity*.

In order to decide which of the two derivations is more likely, a machine needs to know a lot about the world and the context in which the sentence is spoken. Either meaning is plausible for the sentence *Calvin saw a man with the telescope* given the right context. However, in many cases, some of the meanings are more plausible than others even without a particular context. Consider, *Calvin bought a car with anti-lock brakes* which has a more plausible noun-attach reading, while *Calvin bought the car with a low-interest loan* which has a more plausible verb-attach meaning.

We can write a program that can *learn* which attachment is more plausible. In order to keep things simple, we will only use the verb, the final word (as a *head* word) of the noun phrase complement of the verb, the preposition and final word of the noun phrase complement of the preposition. For the sentence *Calvin saw a man with the telescope* we will use the words: *saw, man, with, telescope* in order to predict which attachment is more plausible. We will also consider only the disambiguation between noun vs. verb attachment for sentences or phrases with a single PP, we do not consider the more complex case with more than one PP.
Here is some example NLTK code to read the training dataset for PP-attachment:

```
from nltk.corpus import ppattach
print ppattach.attachments('training')[0]
print ppattach.attachments('devset')[0]
print ppattach.attachments('test')[0]
```

which produces:

```
PPAttachment(sent='0', verb='join', noun1='board', prep='as',
    noun2='director', attachment='V')
PPAttachment(sent='40000', verb='set', noun1='stage', prep='for',
    noun2='increase', attachment='N')
PPAttachment(sent='48000', verb='prepare', noun1='dinner', prep='for',
    noun2='family', attachment='V')
```

Notice that in each case, we have human annotation of which attachment is more plausible. It is convenient to name each component of the training data tuples: $(\#, v, n1, p, n2, attach)$.

a. Learn a model from the training data to predict verb attachment, $\Pr(V \mid p)$. Note that $\Pr(N \mid p) = 1 - \Pr(V \mid p)$. This model is a simple baseline model useful for comparison with the more sophisticated methods below. Provide the accuracy on the dev and test set. While developing your code only evaluate on the dev set to avoid tuning your code to the test set.

b. Learn a model from the training data to predict verb attachment, $\Pr(V \mid v, n1, p, n2)$. Note that $\Pr(N \mid v, n1, p, n2) = 1 - \Pr(V \mid v, n1, p, n2)$. If you use a generative model of $P(V, v, n1, p, n2)$ make sure you smooth this probability to deal with unseen tuples using the dev set. A generative model for this problem is discussed in the lecture notes and you can use `nltk.NaiveBayesClassifier` to implement it (see Chapter 6 of the NLTK book).

   Provide the accuracy on the dev and test set. While developing your code only evaluate on the dev set to avoid tuning your code to the test set.

c. Provide the PNG file of the learning curve for your classifier (plot accuracy on the dev set on the y-axis and size of training data on the x-axis in increments of a 1000 labeled examples). The learning curve using the nltk NaiveBayesClassifier with the LBFGSB algorithm for optimization is given in Fig. 2. Your learning curve could look very different if you use a different learning method.

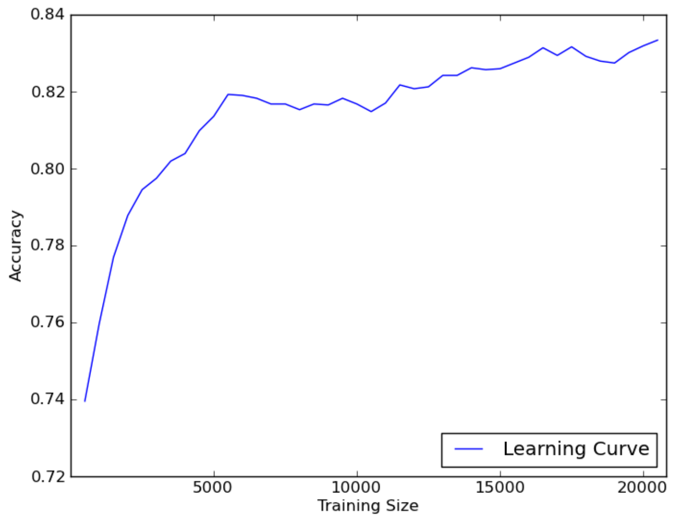

Figure 2: Learning curve for the PP attachment task.

(11) **(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 .
```

We would like to create a translation from the source language which we will take to be Martian-B and produce output in the target language which will be Martian-A. Due to severe budget cutbacks at NASA, decryption of these tablets has fallen to Canadian undergraduate students, namely you. In this question, you should try to solve this task by hand to get some insight into the process of translation.

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 (11b).

```
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.

(12) †† **Statistical Machine Translation**

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 pseudo-code and apply it to solve Question 11 (please read the description below on finding the most likely MARTIAN-A sentence **e** for a given MARTIAN-B sentence **f**).

```
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 $(\mathbf{e}, \mathbf{f})$ in the given translations
        for all word types $e$ in $\mathbf{e}$
            $n_e$ = count of $e$ in $\mathbf{e}$
            total = 0
            for all word types $f$ in $\mathbf{f}$
                total += $t(e|f) \cdot n_e$
            for all word types $f$ in $\mathbf{f}$
                $n_f$ = count of $f$ in $\mathbf{f}$
                rhs = $t(e|f) \cdot n_e \cdot n_f$ / total
                $c(e|f)$ += 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).

The psuedo-code given above is a very simple statistical machine translation model that is called (for historical reasons) 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).

This pseudo-code learns values for probability $t(e|f)$. It is based on a model of sentence translation that is based only on word to word translation probabilities. We define the translation of a source sentence $\mathbf{f}_s = (f_1, \ldots, f_{l_f})$ to a target sentence $\mathbf{e}_s = (e_1, \ldots, e_{l_e})$, with an alignment of each e_j to to some f_i according to an alignment function $a : j \rightarrow i$.

$$\Pr(\mathbf{e}_s, a|\mathbf{f}_s) = \prod_{j=1}^{l_e} t(e_j|f_{a(j)})$$

Consider the example sentence pair below:

$\mathbf{e}_s$	e_1: ok'sifar	e_2: zvau	e_3: hu
$\mathbf{f}_s$	f_1: at'sifar	f_2: somuds	f_3: geyu

For the alignment function $a : \{1 \rightarrow 2, 2 \rightarrow 2, 3 \rightarrow 3\}$ we can derive the probability of this sentence pair alignment to be $\Pr(\mathbf{e}_s, a|\mathbf{f}_s) = t(e_1|f_1) \cdot t(e_2|f_2) \cdot t(e_3|f_3)$. In this simplistic model, we allow any alignment function that maps any word in the source sentence to any word in the target sentence (no matter how far apart they are). However, the alignments are not provided to us, so:

$$\Pr(\mathbf{e}_s|\mathbf{f}_s) = \sum_a \Pr(\mathbf{e}_s, a|\mathbf{f}_s)$$

$$= \sum_{a(0)=1}^{l_f} \cdots \sum_{a(l_e)=1}^{l_f} \prod_{j=1}^{l_e} t(e_j|f_{a(j)}) \text{ (this computes all possible alignments)}$$

$$= \prod_{j=1}^{l_e} \sum_{i=1}^{l_f} t(e_j|f_i) \text{ (converts } l_f^{l_e} \text{ terms into } l_f \cdot l_e \text{ terms)}$$

The pseudo-code provided implements the EM algorithm which learns the parameters $t(\cdot \mid \cdot)$ that maximize the log-likelihood of the training data:

$$L(t) = \underset{t}{\operatorname{argmax}} \sum_s \log \Pr(\mathbf{e}_s|\mathbf{f}_s, t)$$

The best alignment to a target sentence in this model is obtained by simply finding the best translation for each word in the source sentence. For each word f_j in the source sentence the best alignment is given by:

$$a_j = \underset{i}{\operatorname{argmax}}\, t(e_i|f_j)$$

Implement the pseudo-code above to find the distribution $t(e|f)$ from the training data provided in Q. 11. Remember to remove any sentence final punctuation to avoid spurious mappings of words to those markers.

a. In Q. 11a you need to find a translation dictionary. To solve this question, you should find a word e^* in Martian-A where $e^* = \operatorname{argmax}_e t(e|f)$ 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. 11b 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 $e_i = \operatorname{argmax}_e t(e|f_i)$. In cases where a translation does not exist, insert the word ? as the unseen word into your translated sentence.

c. *(this sub-part is optional; no extra marks)*

   Q. 11c provides some additional Martian-A sentences. You can use these extra Martian-A sentences to deal with cases in Q. 11b 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 $w_2 = \operatorname{argmax}_w p(w|w_1)$ 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 ?.

d. *(this sub-part is optional; no extra marks)*

   Use the additional Martian-A sentences in Q. 11c to define a language model and use it to improve the translation output. Due to the small amount of data, a unigram or bigram model with some simple smoothing will suffice. Let **f** be the source sentence of length n:

$$P_{+lm}(\mathbf{e} \mid \mathbf{f}) = \prod_{i=1}^{n} t(e_i|f_i) \cdot p_{lm}(e_i \mid \phi(e_1, \ldots, e_{i-1}))$$

   Your output Martian-A sentence should now be defined as $\mathbf{e}^* = e_1, e_2, \ldots, e_n$ which is defined as

$$\mathbf{e}^* = \underset{\mathbf{e}}{\operatorname{argmax}}\, P_{+lm}(\mathbf{e} \mid \mathbf{f})$$

   ϕ would be empty for the unigram model and would return e_{i-1} for a bigram model, and so on. We assume there is a special start of sentence token defined as e_0 to handle generation of e_1.