Homework #3: CMPT-825

Anoop Sarkar – anoop@cs.sfu.ca

For the programming questions use a `makefile` such that `make` compiles all your programs, and `make test` runs each program, and supplies the necessary input files. For each question, please submit a short readme file describing your approach as a LaTeXor ASCII file.

(1) **Minimum Edit Distance**
The following Python code computes the minimum number of edits: insertions, deletions, or substitutions that can convert an input source string to an input target string. Using the cost of 1, 1 and 2 for insertion, deletion and replacement is traditionally called Levenshtein distance.

```
def distance(target, source, insertcost, deletecost, replacecost):
    n = len(target)+1
    m = len(source)+1
    # set up dist and initialize values
    dist = [ [0 for j in range(m)] for i in range(n) ]
    for i in range(1,n):
        dist[i][0] = dist[i-1][0] + insertcost
    for j in range(1,m):
        dist[0][j] = dist[0][j-1] + deletecost
    # align source and target strings
    for j in range(1,m):
        for i in range(1,n):
            inscost = insertcost + dist[i-1][j]
            delcost = deletecost + dist[i][j-1]
            if (source[j-1] == target[i-1]): add = 0
            else: add = replacecost
            substcost = add + dist[i-1][j-1]
            dist[i][j] = min(inscost, delcost, substcost)
    # return min edit distance
    return dist[n-1][m-1]

if __name__=="__main__":
    from sys import argv
    if len(argv) > 2:
        print "levenshtein distance =", distance(argv[1], argv[2], 1, 1, 2)
```

Let's assume we save this program to the file `distance.py`, then:

```
$ python2.6 distance.py gamble gumbo
levenshtein distance = 5
```

a. Your first task is to produce the following visual display of the best (minimum distance) alignment:

```
$ python2.6 view_distance.py gamble gumbo
levenshtein distance = 5
g a m b l e
|   | |
g u m b _ o
```

```
$ python2.6 view_distance.py "recognize speech" "wreck a nice beach"
levenshtein distance = 14
_ r e c _ _ o g n i z e   s p e e c h
  | | |         | |   | |     |   | |
w r e c k   a   n i c e   _ b e a c h

$ python2.6 view_distance.py execution intention
levenshtein distance = 8
_ e x e c u t i o n
      |     | | | |
i n t e _ n t i o n
```

The 1st line of the visual display shows the *target* word and the 3rd line shows the *source* word. An insertion in the target word is represented as an underscore in the 3rd line aligned with the inserted letter in the 1st line. Deletion from the source word is represented as an underscore '_' in the 1st line aligned with the corresponding deleted character in the source on the 3rd line. Finally, if a letter is unchanged between target and source then a vertical bar (the pipe symbol '|') is printed aligned with the letter in the 2nd line.

You can produce this visual alignment using two different methods:

- Memorize which of the different options: insert, delete or substitute was taken as the entries in the table are computed; or
- Trace back your steps in the table starting from the final distance score by comparing the scores from the predecessor of each table entry and picking the minimum each time.

There can be many different alignments that have exactly the same minimum edit distance. Therefore, for the above examples producing a visual display with a different alignment but which has the same edit distance is also correct.

b. Print out all the valid alignments with the same minimum edit distance. For longer input strings, you should print out only the first N alignments (where N has to be set by the user) because the number of possible alignments is exponential in the size of the input strings.

We can see this by considering a recursive function that prints out all alignments (instead of using the dynamic programming approach). Let us call this function *align*. Let us assume that the two input strings are of length n, m. Then, the number of recursive calls can be written as a recurrence:

$$align(n, m) = align(n, m-1) + align(n-1, m-1) + align(n-1, m)$$

Let us assume $n = m$, then:

$$\begin{aligned} align(n, n) &= align(n, n-1) + align(n-1, n-1) + align(n-1, n) \\ &= 2 \cdot align(n, n-1) + align(n-1, n-1) \\ &= 2\left[align(n, n-2) + align(n-1, n-2) + align(n-1, n-1)\right] + align(n-1, n-1) \\ &> 3 \cdot align(n-1, n-1) \end{aligned}$$

Thus, each call to the function $align(n, n)$ results in three new recursive calls. The number of times the align function will be called is 3^n which is a bound on the total number of distinct alignments.

c. Use the OpenFST toolkit to compute the minimum edit distance between any two strings drawn from the language of the regular expression `[a-zA-Z]*`. Provide the edit distance FST along with the output of the minimum edit distance for the examples in Q 1a.

(2) **Sino-Korean Number Pronunciation**

The Korean language has two different number systems. The native Korean numbering system is used for counting people, things, etc. while the Sino-Korean numbering system is used for prices, phone numbers, dates, etc. (called Sino-Korean because it was borrowed from the Chinese language).

Write a program that takes a number as input and produces the Sino-Korean number pronunciation appropriate for prices using the table provided below. The program should accept any number from 1 to $999,999,999$ and produce a single output pronunciation. The commas are used here to make the numbers easier to read, and can be simply deleted from the input.

You should produce the Korean romanized output (Korean written using the Latin script) as shown in the table, rather than the Hangul script (the official script for the Korean language).

1	il	10	sib	19	sib gu	100	baek
2	i	11	sib il	20	i sib	1,000	cheon
3	sam	12	sib i	30	sam sib	10,000	man
4	sa	13	sib sam	40	sa sib	100,000	sib man
5	o	14	sib sa	50	o sib	1,000,000	baek man
6	yuk	15	sib o	60	yuk sib	10,000,000	cheon man
7	chil	16	sib yuk	70	chil sib	100,000,000	eok
8	pal	17	sib chil	80	pal sib		
9	gu	18	sib pal	90	gu sib		

For example, the input $915,413$ should produce the output *gu sib il man o cheon sa baek sib sam*. Note that as shown the table above 1000 is pronounced *cheon* rather than *il cheon*, and so $111,000$ is pronounced as *sib il man cheon*.

An intermediate representation makes this task much easier. For input $915,413$ consider the intermediate representation of `9[10]1[10^4]5[10^3]4[10^2][10]3#`, where `[10]`, `[10^4]` are symbols in an extended alphabet. The mapping from numbers to intermediate representation and the mapping into Korean pronunciations are both easily implemented as finite-state transducers.

(3) **(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 cheap labor: Canadian graduate students. You can choose to write code (see next question) to solve the problems or do it by hand – it's up to you. I recommend doing it by hand to get a feeling for the problem.

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 `?` 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 programs 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 (3b).

```
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.

(4) **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 3 (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).

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).

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:

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

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

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

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:

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

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:

$$a_j = \operatorname*{argmax}_i t(e_i|f_j)$$

Implement the pseudo-code above to find the distribution $t(e|f)$ from the training data provided in Q. 3. Remember to remove any sentence final punctuation to avoid spurious mappings of words to those markers.

a. In Q. 3a 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. 3b 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. 3c provides some additional Martian-A sentences. You can use these extra Martian-A sentences to deal with cases in Q. 3b 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 `?`.

d. Use the additional Martian-A sentences in Q. 3c 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 $\mathbf{f}$ 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

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

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.

e. The output Martian-A sentences are always the same length as the input Martian-B sentences since we assume that we are producing the Martian-A sentences one word at a time where for each Martian-B word f_i we produce a single Martian-A word e_i. However, there might be another Martian-A sentence $e_0^*, \ldots, e_m^*$ that has a higher probability in the model defined in Q. 4d (notice that in this case m does not have to equal n, the length of the source Martian-B sentence). One way to decode is by first obtaining an initial translation (using our initial decoder from Q. 4d) and then produce alternative translations by using the following change operators for each position i:

if frequency of a word is $> c$ then it is defined as frequent
let there be k words defined as frequent
define the following change operators for position i,
define change(i):
    delete: delete the word at i
    transpose: transpose adjacent words i and $i-1$ if $i > 1$ and $i \leq n$
    insert: insert a frequent word from the vocabulary at position i

For each position i we apply the above change operators giving us $n + (n-1) + kn$ new output translations. This defines a set of alternatives produced from our initial translation. Choose an appropriate value for c based on your experiments. Score each of the alternatives according to the model, and pick the most likely one. If the most likely translation was the one without any changes then we stop and output that as our translation, else we keep the most likely changed translation and then produce a new set of alternative translations using the change operators. This is a hill-climbing algorithm that will stop at some local maximum value of the model score.