COMS 4705 – Natural Language Processing – Spring 2015

Assignment 1

Language Modeling and Part of Speech Tagging

(version 4, February 3, 2015)

Due: Wednesday, February 18, 11:59:59 PM

Tutorial

Before Starting

The Natural Language Tool Kit (NLTK) is a useful API for NLP. It has already been installed on the CLIC (CS) machines, where you will submit your assignment. We will provide more detailed submission instructions at a later time. Note that NLTK is shared between the 4701 and 4705 classes, so some paths may link to ~4701. This should not concern you.

Getting Started

Add NLTK's location to your default path

Add the following line of code to your ~/.bashrc file.

```
export PYTHONPATH=$PYTHONPATH:/home/cs4701/python/lib/python2.7/site-packages
```

After, don't forget to run:

```
source ~/.bashrc
```

Create a link to the NLTK files (equivalent to downloading the files yourself)

Login to your CLIC account and run the following commands in your home directory.

```
% rm -rf nltk_data
% ln -s ~coms4705/nltk_data nltk_data
```

What you need to know

Import the NLTK package

To use NLTK package, you must include the following line at the beginning of your code.

```
import nltk
```

Tokenize

To tokenize means to break text into words, symbols and other meaningful units. In NLTK, text can be tokenized using the word_tokenize() method. It returns a list of tokens that will be the input for many methods in NLTK.

```
sentence = "At eight o'clock on Thursday morning on Thursday morning on Thursday morning."
tokens = nltk.word_tokenize(sentence)
```

N-grams

An n-gram (in the context of this assignment) is a contiguous sequence of n tokens in a sentence. The following code returns a tuple, each of whose elements is a tuple that describes an n-gram in your input.

```
bigram_tuples = tuple(nltk.bigrams(tokens))
trigram_tuples = tuple(nltk.trigrams(tokens))
```

We can calculate the raw count of a particular n-gram using the following code.

```
count = {item : bigram_tuples.count(item) for item in set(bigram_tuples)}
```

Default Tagger (non-statistical)

The most naïve way of tagging is to simply assign the same tag to all the tokens. This is exactly what the NLTK default tagger does. Although inaccurate and arbitrary, it sets a baseline for taggers, and can be used as a default tagger when more sophisticated methods fail.

In NLTK, it's easy to create a default tagger by indicating the tag in the default tagger constructor.

```
default_tagger = nltk.DefaultTagger('NN')
tagged_sentence = default_tagger.tag(tokens)
print tagged_sentence
```

Now we have our first tagger. NLTK can help if you need to understand the meaning of a tag.

```
#Show the description of the tag ' NN'
nltk.help.upenn_tagset('NN')
```

You will find instructions for evaluating taggers (your own and NLTK's) in the assignment section.

Regular Expression Tagger (non-statistical)

A regular expression tagger maintains a list of regular expressions paired with a tag (see the Wikipedia article for more information: http://en.wikipedia.org/wiki/Regular_expression). The tagger tries to match each token to one of the regular expressions in its list; the token receives the tag that is paired with the first matching regular expression. "None" is given to a token that does not match any regular expression.

To create a Regular Expression Tagger in NLTK, we provide a list of pattern-tag pairs to the appropriate constructor.

```
patterns = [(r'.*ing$', 'VBG'),(r'.*ed$', 'VBD'),(r'.*es$', 'VBZ'),(r'.*ed$', 'VB')]
regexp_tagger = nltk.RegexpTagger(patterns)
regexp_tagger.tag(tokens)
```

N-gram tagger (statistical)

Although there are many different kinds of statistical taggers, we will only work with Hidden Markov Model (HMM) taggers.

Like every statistical tagger, n-gram taggers use a set of tagged sentences, known as the training data, to create a model that is used to tag new sentences. In NLTK, the training data must be a set of tagged tokens; NLTK provides tagged sentences in this format.

```
#import the corpus from NLTK and build the training set from sentences in "news"
from nltk.corpus import brown
```

```
training=brown.tagged_sents(categories='news')

#Create Unigram, Bigram, Trigram taggers based on the training set.
unigram_tagger = nltk.UnigramTagger(training)
bigram_tagger = nltk.BigramTagger(training)
trigram_tagger = nltk.TrigramTagger(training)
```

Although we could also build 4-gram, 5-gram, etc. taggers, trigram taggers are the most popular model. This is because a trigram model is an excellent compromise between computational complexity and performance.

Combination of taggers

A tagger fails when it cannot find a best tag sequence for a given sentence. For example, one situation when an n-gram tagger will fail is when it encounters an OOV (out of vocabulary) word not seen in the training data: the tagger will tag the word as ”NONE”. One way to handle tagger failure is to switch to an alternative tagger if the primary one fails. This is called “using back off.” One can easily set a hierarchy of taggers in NLTK as follows.

```
default_tagger = nltk.DefaultTagger('NN')
bigram_tagger = nltk.BigramTagger(training, backoff=default_tagger)
trigram_tagger = nltk.TrigramTagger(training, backoff=bigram_tagger)
```

Tagging low frequency words

Low frequency words are another common source of tagger failure, because an n-gram that contains a low frequency word and is found in the test data might not be found in the training data. One method to resolve this tagger failure is to group low frequency words. For example, we could substitute the token “_RARE_” for all words with frequency lower than 0.05% in the training data. Any words in the development data that were not found in the training data could then be treated instead as the token “_RARE_”, thereby allowing the algorithm to assign a tag. If we wanted to add another group we could substitute the string “_NUMBER_” for those rare words that represent a numeral. When tagging the test data, we could substitute “_NUMBER_” for all tokens that were unseen in the training data and represent a numeral.

Assignment

For this assignment, you will be analyzing and tagging the Brown corpus. The Brown corpus is a dataset of English sentences compiled in the 1960s. You can read more about it at http://en.wikipedia.org/wiki/Brown_Corpus

We have provided the following files:

Brown_train.txt	Untagged Brown training data
Brown_dev.txt	Untagged Brown development data
Brown_tagged_train.txt	Tagged Brown training data
Brown_tagged_dev.txt	Tagged Brown development data
Sample1.txt	Mystery sentences!
Sample2.txt	More mystery sentences!
perplexity.py	A script to analyze perplexity for part A
pos.py	A script to analyze POS tagging for part B
solutionsA.py	Skeleton code for part A
solutionsB.py	Skeleton code for part B

Sentences are separated by a newline character, and tags are included in this format: WORD/TAG.

The Report

You need to create a brief report about your work in a file "README.txt". At the top it should include your UNI and the amount of time your programs take to complete. Throughout the assignment, you will be asked to include specific output or comment on specific aspects of your work. You may include further commentary if you feel that it is relevant to the assignment; however, this is not required.

Submission

For submission, you will create a hidden directory that contains a secret directory as explained in class. Within this directory, create another directory called "Homework1". Put all files you will submit into this directory. For example, your submission directory will look similar to mine, which is:

~stk2122/hidden/0981439824589234/Homework1/

To begin the assignment, copy the entire contents of the directory "~coms4705/Homework1" into your submission directory. Fill in "solutionsA.py" and "solutionsB.py" with your own solutions. When you're finished, this directory should contain: "solutionsA.py", "solutionsB.py", "README.txt", both of the evaluation scripts we provided, and the six plain text files with sentences from the Brown corpus. Do not include the output files that our skeleton code produces for you.

You must use the skeleton code we provide you. Do not rename these files. In them, you will find our solutions with most of the code removed. In the missing code's place, you will find comments to guide you to a correct solution. Additionally, you will find a main method that sequentially performs each of the tasks outlined in the assignment; along the way, this main method creates output files. Do not modify the code that creates these files and pay close attention to the data structures specified in the comments.

Note: You must set the permissions in your submission directory please run the following command:

% chmod –R 755 ~<your uni>/hidden/<your key>/

Note: you must use $\log_2$ (base 2) probabilities. Use -1000 instead of $\log_2 0$.

Part A – Language Model

1) Calculate the uni-, bi-, and trigram probabilities of the sentences in "Brown_train.txt". You must use NLTK to help you. See the above tutorial for hints. Don't forget to add appropriate sentence start and end symbols. Our code will output your probabilities in a file "A1.txt". Here's what the first few lines of our file looks like:

   UNIGRAM detractor -19.4924341431
   UNIGRAM wolves -17.9074716424
   UNIGRAM midweek -19.4924341431
   UNIGRAM scoring -17.4924341431
   UNIGRAM 54,320 -19.4924341431

2) Use your n-gram model to score each sentence in the Brown training data with each n-gram model. Our code will output your scores in three files: "A2.uni.txt", "A2.bi.txt", "A2.tri.txt". The format of each of these files is the same. Here's what the first few lines of our "A2.uni.txt" looks like:

   -175.321844321
   -257.192657763
   -140.003392704
   -115.263993288
   -144.531931147

   Now, you need to run our perplexity script, "perplexity.py" on each of these files. To run the script, the command is:

   % python perplexity.py <file of scores> <file of sentences that were scored>

   Where <file> is one of the A2 output files. Include the output of our script in your README. Here's what our script printed when <file> was "A2.uni.txt".

   The perplexity is 999.677940956

3) As a final step in the development of your n-gram language model, implement linear interpolation among the three n-gram models you have created. Linear interpolation is a method that aims to derive a better tagger by using all three uni-, bi-, and trigram taggers at once. Each tagger is given a weight described by a parameter lambda. There are some excellent methods for approximating the best set of lambdas, but for now, set all three lambdas to be equal. You can

read more about linear interpolation in section 4.4.3 (page 15) of the readings posted on Courseworks. Our code outputs your scores into "A3.txt". The first few lines of our file look like:

-46.636035201
-85.8406465628
-58.6244067436
-47.5631994821
-52.7903849129

Run the perplexity script on the output file and include the script's output in your report. Here's our perplexity output:

The perplexity is 13.0930938155

4) Briefly comment on any differences in performance between the models with and without linear interpolation. Use the perplexity output to support your conclusions.

5) Both "Sample1.txt" and "Sample2.txt" contain sets of sentences; one of the files is an excerpt of the Brown dataset. Use your model to score the sentences in both files. Our code outputs the scores of each into "Sample1_scored.txt" and "Sample2_scored.txt". Run the perplexity script on both output files and include the output in your report. Use these results to make an argument for which sample belongs to the Brown dataset.

Part B – Part of Speech Tagging

1) First, you must separate the tags and words in "Brown_tagged_training.txt". You'll want to store the sentences without tags in one data structure, and then the tags of each sentence in another. Make sure to add sentence start and end symbols to both sets. There is no output for this question.
Hint: make sure you accommodate words that themselves contain backslashes – i.e. "1/2" is encoded as "1/2/NUM" in tagged form; make sure that the token you extract is "1/2" and not "1".

2) Now, calculate the trigram probabilities for the tags. Our code outputs your results to a file "B2.txt". Here are the first few lines of our file (we used '*' for a sentence start symbol):

TRIGRAM * X VERB -3.90689059561
TRIGRAM NOUN VERB ADV -2.84564255455
TRIGRAM PRT VERB DET -1.84386447329
TRIGRAM * ADP VERB -4.93221475197
TRIGRAM PRT CONJ . -7.92481250361

3) Now you will implement a smoothing method. To prepare for adding smoothing, replace every word that occurs five times or fewer with the string "_RARE_". It will be helpful to create a list of words that occur *more* than five times in the training data; when tagging, any word that does not appear in this list should be replaced with the token "_RARE_". Our code outputs your new

version of the training data to “A3.txt”. Here are the first two lines of our file:

At that time highway engineers traveled rough and dirty roads to accomplish their duties .

RARE _RARE_ vehicles was a personal _RARE_ for such employees , and the matter of providing state transportation was felt perfectly _RARE_ .

4) Next, calculate the emission probabilities on the modified dataset. Our code will output your results to “B4.txt”. Here are the first few lines of our file:

tubes NOUN -12.711977598
hundred NUM -6.32713325402
red ADJ -9.15426433692
fire VERB -13.8689189884
brick NOUN -14.0745476774

5) Now, implement the Viterbi algorithm for HMM taggers. The Viterbi algorithm is a dynamic programming algorithm that has many applications. For our purposes, the Viterbi algorithm is a comparatively efficient method for finding the highest scoring tag sequence for a given sentence. Please read about the specifics about this algorithm in sections 7.4 and 8.4 in the readings on Courseworks.
Note: your book uses the term “state observation likelihood” for “emission probability” and the term “transition probability” for “trigram probability.”
Using your emission and trigram probabilities, calculate the most likely tag sequence for each sentence in “Brown_dev.txt”. Don’t forget to replace rare words with “_RARE_”. Your tagged sentences will be output to “B5.txt”. Here are the first two tagged sentences in our file:

He/PRON had/VERB obtained/VERB and/CONJ _RARE_/VERB a/DET veteran/ADJ ship/NOUN called/VERB the/DET _RARE_/NOUN and/CONJ had/VERB _RARE_/VERB a/DET crew/NOUN of/ADP _RARE_/NOUN ,/. the/DET largest/ADJ he/PRON had/VERB ever/ADV commanded/VERB ./.

The/DET purpose/NOUN of/ADP this/DET fourth/ADJ _RARE_/NOUN was/VERB clear/ADJ ./.

Use the part of speech evaluation script, pos.py, to compare the output file with “Brown_tagged_dev.txt”. Include the output of our script in your report. To use the script, run the following command:
python pos.py <tagged file> <correct file>

When <tagged file> was “B5.txt” and <correct file> was <Brown_tagged_dev.txt> we got:

Percent correct tags: 74.8896749858

6) Finally, create an instance of NLTK’s trigram tagger set to back off to NLTK’s bigram tagger. Let the

bigram tagger itself back off to NLTK's default tagger using the tag "NN". Import the Brown data (do not include a specific category) as in the tutorial and use that to train all three taggers. Our code outputs your results to a file "B6.txt". Use pos.py to compare NLTK's tagger output to "Brown_tagged_dev.txt". Briefly comment on the performance of your HMM tagger in comparison with NLTK's trigram tagger with back offs. Use the output of pos.py to support your conclusions. Here are the first few lines of our "B6.txt":

He/NN had/HVD obtained/NN and/CC _RARE_/NN a/AT veteran/NN ship/NN called/VBN the/AT _RARE_/NN and/CC had/HVD _RARE_/NN a/AT crew/NN of/IN _RARE_/NN ,/, the/AT largest/JJT he/PPS had/HVD ever/RB commanded/VBN ./.

The/NN purpose/NN of/IN this/DT fourth/OD _RARE_/NN was/BEDZ clear/JJ ./.

And here is what pos.py prints after analyzing our "B6.txt"

Percent correct tags: 3.67307053374