COMS 4705 – Natural Language Processing – Fall 2015

Assignment 1

Language Modeling and Part of Speech Tagging

(version 9, September 24th, 2015)

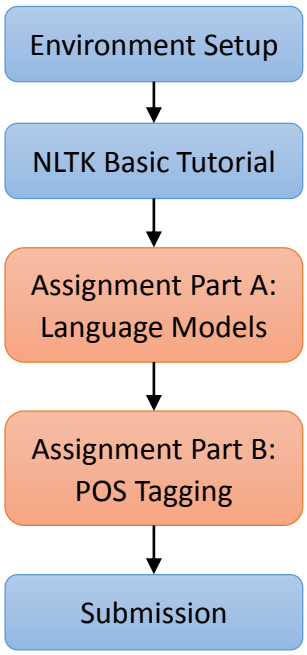
Due: Wednesday, October 7th, 11:59:59 PM

Introduction

In this assignment, we will:

1. Go through the basics of NLTK, the most popular NLP library for python;
2. Develop and evaluate several language models;
3. Develop and evaluate a full part-of-speech tagger using Viterbi algorithm.

This document is structured in the following sequence:



Setup everything to get you ready to start the assignment

Quick intro to NLTK, library that we will use throughout the semester

We will create, train and evaluate language models using the Brown corpus.

We will create, train and evaluate a part-of-speech tagger. This will require implementation of the Viterbi Algorithm.

Once you're done, follow the instructions to submit.

Environment Setup

We will assume at this point you already have a CLIC account and have already used SSH to connect to the CLIC servers. We will also assume you have already created a hidden directory with your homework PIN number (if not, see instructions sent through email before proceeding). The following instructions should be executed while logged to the CLIC server.

Step 1 – Add NLTK location to your default path

Open the file ~/.bashrc and add the following line to the end of the file:

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

After saving and closing the file, run the command below:

```
source ~/.bashrc
```

Step 2 – Create a link to the NLTK files, available at the course directory

Go to your home directory:

```
cd ~/
```

Create a symbolic link to NLTK data:

```
ln –s ~coms4705/nltk_data nltk_data
```

Step 3 – Copy the homework files to your hidden directory under Homework1 folder

Use the following command

```
cp -r ~coms4705/Homework1 ~/hidden/<YOUR_PIN>/
```

Basic NLTK Tutorial

The Natural Language Tool Kit (NLTK) is the most popular NLP library for python. NLTK has already been installed on the CLIC machines, so you do not have to install it yourself. If you followed the "Environment Setup" section, you should be good to start using NLTK now.

Now let's walk through some simple NLTK use cases that concern this assignment. For that, you will need first to open a python interactive shell. Just type `python` on the command line and you should see the interactive shell start.

Import the NLTK package

To use NLTK package, you must include the following line at the beginning of your code (or in this case just type in the interactive shell):

```
import nltk
```

Tokenization

To tokenize means to break a continuous string into tokens (usually words, but a token could also be a symbol, punctuation, or other meaningful unit). 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 Generation

An n-gram (in the context of this assignment) is a contiguous sequence of n tokens in a sentence. The following code returns a list of bigrams and a list of trigrams. Each n-gram is represented as a tuple in python (if you are not familiar with python tuples read the python tuple doc page)

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

We can calculate the count of each n-gram using the following code:

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

Or we can find all the distinct n-grams that contain the word “on”:

```
ngrams = [item for item in set(bigram_tuples) if "on" in item]
```

If you find it hard to understand the examples above, read about list/dict comprehensions here. List/dict comprehensions are a way of executing iterations in one line that may be very useful and convenient. Besides making coding easier, learning them will also help you understand code written by other python programmers.

Default POS Tagger (Non-statistical)

The most naïve way of tagging parts-of-speech is to 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 default tag in the constructor.

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

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')
```

Regular Expression POS Tagger (Non-statistical)

A regular expression tagger maintains a list of regular expressions paired with a tag (see the Wikipedia article for more information about regular expressions: 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. Example:

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

N-gram HMM Tagger (Statistical)

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

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, a sentence of the training data must be formatted as a list of tuples, where each tuple is a pair or word-tag (see example below).

```
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL')]
```

NLTK already provides corpora formatted this way. In particular, we are going to use the Brown corpus.

```
# 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 fall back 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. We will use this technique later in this assignment.

Provided Files and Report

Now let’s go back to the assignment folder you copied to your home directory (should be ~/Homework1). In this assignment we will be using the Brown Corpus, which is a dataset of English sentences compiled in the 1960s. We have provided this dataset to you so you don’t have to load it yourself from NLTK.

Besides data, we are also providing code for evaluating your language models and POS tagger, and a skeleton code for the assignment. In this assignment you should not create any new code files, but rather just fill the functions in the skeleton code.

We have provided the following files:

data/Brown_train.txt	Untagged Brown training data

data/Brown_tagged_train.txt	Tagged Brown training data
data/Brown_dev.txt	Untagged Brown development data
data/Brown_tagged_dev.txt	Tagged Brown development data
data/Sample1.txt	Additional sentences for part A
data/Sample2.txt	More additional sentences for part A
perplexity.py	A script to analyze perplexity for part A
pos.py	A script to analyze POS tagging accuracy for part B
solutionsA.py	Skeleton code for part A
solutionsB.py	Skeleton code for part B

The only files that you should modify throughout the whole assignment are solutionsA.py and solutionsB.py.

Data Files Format

The untagged data files have one sentence per line, and the tokens are separated by spaces. The tagged data files are in the same format, except that instead of tokens separated by spaces those files have TOKEN/TAG separated by spaces.

Report

Before starting the assignment, create a "README.txt" file on the homework folder. At the top, include a header that contains your UNI and name. Throughout the assignment, you will be asked to include specific output or comment on specific aspects of your work. We recommend filling the README file as you go through the assignment, as opposed to starting the report afterwards.

In this report it is not necessary to include introductions and/or explanations, other than the ones explicitly requested throughout the assignment.

Assignment Part A – Language Model

In this part of the assignment you will be filling the solutionsA.py file. Open the file and notice there are several functions with a #TODO comment; you will have to complete those functions. To understand the general workflow of the script read the **main()** function ***but do not modify it***.

1) Calculate the uni-, bi-, and trigram log-probabilities of the data in "Brown_train.txt". This corresponds to implementing the **calc_probabilities()** function. In this assignment we will always use **log base 2**.

   Don't forget to add the appropriate sentence start and end symbols; use "*" as start symbol and "STOP" as end symbol (These are defined as constants START_SYMBOL and STOP_SYMBOL in the skeleton ode). You may or may not use NLTK to help you identify the n-grams, but you should not use NLTK to tokenize text. Remember the text *is already tokenized* and the tokens are separated by spaces.

   The code will output the log probabilities in a file "output/A1.txt". Here's a few examples of log probabilities of uni-, bi-, and trigrams for you to check your results:

UNIGRAM captain -14.2809819899
UNIGRAM captain's -17.0883369119
UNIGRAM captaincy -19.4102650068

BIGRAM and religion -12.9316608989
BIGRAM and religious -11.3466983981
BIGRAM and religiously -13.9316608989

TRIGRAM and not a -4.02974734339
TRIGRAM and not by -4.61470984412
TRIGRAM and not come -5.61470984412

Make sure your result is exactly the same as the examples above. If it is, include in your README the log probabilities of the following n-grams (*note the n-grams are case-sensitive*):
UNIGRAM natural
BIGRAM natural that
TRIGRAM natural that he

2) Use your models to find the log-probability, or score, of each sentence in the Brown training data with each n-gram model. This corresponds to implementing the **score()** function.

Make sure to accommodate the possibility that you may encounter in the sentences an n-gram that doesn't exist in the training corpus. This will not happen now, because we are computing the log-probabilities of the training sentences, but will be necessary for question 5. The rule we are going to use is: if you find any n-gram that was not in the training sentences, set the whole sentence log-probability to -1000 (Use constant MINUS_INFINITY_SENTENCE_LOG_PROB).

The code will output scores in three files: "output/A2.uni.txt", "output/A2.bi.txt", "output/A2.tri.txt". These files simply list the log-probabilities of each sentence for each different model. Here's what the first few lines of each file looks like:

A2.uni.txt
-178.726835483
-259.85864432
-143.33042989

A2.bi.txt
-92.1039984276
-132.096626407
-90.185910842

A2.tri.txt
-26.1800453413
-59.8531008074

-42.839244895

Now, you need to run our perplexity script, "perplexity.py" on each of these files. This script will count the words of the corpus and use the log-probabilities computed by you to calculate the total perplexity of the corpus. To run the script, the command is:

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

Where <file of scores> is one of the A2 output files and <file of sentences that were scored> is "data/Brown_train.txt". **Include the perplexity of the corpus for the three different models in your README**. Here's what our script printed when <file> was "A2.uni.txt".

```
python perplexity.py output/A2.uni.txt data/Brown_train.txt
```

The perplexity is 1052.4865859

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. This corresponds to implementing the **linearscore()** function.

   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 of the book. In the case of linear interpolation, you will only set a sentence log-probability to -1000 if while you traverse the sentence you encounter an unigram, bigram and trigram that you have never seen before (in practice it is the same of encountering a new unigram).

   The code outputs scores to "output/A3.txt". The first few lines of this file look like:

   -46.5891638973
   -85.77421559
   -58.5442024163
   -47.5165051948
   -52.7387360815

   Run the perplexity script on the output file and include the perplexity in your README.

4) Briefly answer on your README the following question: When you compare the performance (perplexity) between the best model without interpolation and the models with linear interpolation, is the result you got expected? Explain why. (max 60 words, but 30 is fine too!)

5) Both "data/Sample1.txt" and "data/Sample2.txt" contain sets of sentences; one of the files is an excerpt of the Brown training 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 perplexity output of both samples in your

README. Use these results to make an argument for which sample belongs to the Brown dataset and which does not.

Assignment Part B – Part-of-Speech Tagging

In this part of the assignment you will be filling the solutionsB.py file. Open the file and notice there are several functions with a #TODO comment; you will have to complete those functions. To understand the general workflow of the script read the **main()** function ***but do not modify it***.

1) First, you must separate the tags and words in "Brown_tagged_train.txt". This corresponds to implementing the **split_wordtags()** function. You'll want to store the sentences without tags in one data structure, and the tags alone in another (see instructions in the code). Make sure to add sentence start and stop symbols to **both** lists (of words and tags), and use the constants START_SYMBOL and STOP_SYMBOL already provided. You don't need to write anything on README about 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. This corresponds to implementing the **calc_trigrams()** function. The code outputs your results to a file "output/B2.txt". Here are a few lines (not contiguous) of this file for you to check your work:
   TRIGRAM * * ADJ -5.20557515082
   TRIGRAM ADJ . X -9.99612036303
   TRIGRAM NOUN DET NOUN -1.26452710647
   TRIGRAM X . STOP -1.92922692559

   After you checked your algorithm is giving the correct output, add to your README the log probabilities of the following trigrams:
   TRIGRAM CONJ ADV ADP
   TRIGRAM DET NOUN NUM
   TRIGRAM NOUN PRT PRON

3) The next step is to implement a smoothing method. To prepare for adding smoothing, replace every word that occurs five times or fewer with the token "_RARE_" (use constant RARE_SYMBOL). This corresponds to implementing the **calc_known()** and **replace_rare()** functions.

   First you will 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_". You don't need to write anything on README about this question. The code outputs the new version of the training data to "output/B3.txt". Here are the first two lines of this 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, we will calculate the emission probabilities on the modified dataset. This corresponds to implementing the **calc_emission()** function. Here are a few lines (not contiguous) of this file for you to check your work:

America NOUN -10.99925955
Columbia NOUN -13.5599745045
New ADJ -8.18848005226
York NOUN -10.711977598

After you checked your algorithm is giving the correct output, add to your README the log probabilities of the following emissions **(note words are case-sensitive)**:
*** ***
Night NOUN
Place VERB
prime ADJ
STOP STOP
RARE VERB

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 8.4 and 9.4 in of the book.

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". This corresponds to implementing the **viterbi()** function. Your tagged sentences will be output to "B5.txt". Here is how the first two tagged sentences should be like:

He/PRON had/VERB obtained/VERB and/CONJ provisioned/VERB a/DET veteran/ADJ ship/NOUN called/VERB the/DET Discovery/NOUN and/CONJ had/VERB recruited/VERB a/DET crew/NOUN of/ADP twenty-one/NOUN ,/. the/DET largest/ADJ he/PRON had/VERB ever/ADV commanded/VERB ./.
The/DET purpose/NOUN of/ADP this/DET fourth/ADJ voyage/NOUN was/VERB clear/ADJ ./.

Note that the output doesn't have the "_RARE_" token, but you still have to count unknown words as a "_RARE_" symbol to compute probabilities inside the Viterbi Algorithm.

When exploring the space of possibilities for the tags of a given word, make sure to only consider tags with emission probability greater than zero for that given word. Also, when accessing the transition probabilities of tag trigrams, use -1000 (constant LOG_PROB_OF_ZERO in the code) to represent the log-probability of an unseen transition.

Once you run your implementation, use the part of speech evaluation script pos.py to compare the output file with "Brown_tagged_dev.txt". Include the accuracy of your tagger in the README file. To use the script, run the following command:

```
python pos.py output/B5.txt data/Brown_tagged_dev.txt
```

This is the result we got with our implementation of the Viterbi algorithm:
Percent correct tags: 93.3249946254

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 "NOUN". Implement this in the **nltk_tagger()** function. The code outputs your results to a file "B6.txt", and this is how the first two lines of this file should look like:
He/NOUN had/VERB obtained/VERB and/CONJ provisioned/NOUN a/DET veteran/NOUN ship/NOUN called/VERB the/DET Discovery/NOUN and/CONJ had/VERB recruited/NOUN a/DET crew/NOUN of/ADP twenty-one/NUM ,/. the/DET largest/ADJ he/PRON had/VERB ever/ADV commanded/VERB ./.
The/NOUN purpose/NOUN of/ADP this/DET fourth/ADJ voyage/NOUN was/VERB clear/ADJ ./.

Use pos.py to evaluate the NLTK's tagger accuracy and put the result in your README. This is the accuracy that we got with our implementation:

Percent correct tags: 87.9985146677

Submission

Before submission, run your code one more time and record in your README file the time it took to execute Part A and Part B, *separately*. Add the times of execution in the end of your README file.

You should submit your assignment through your hidden directory on the CLIC servers. Once you are done with the homework and has finalized the README.txt file, check once again that this is the path for your homework:

```
~/hidden/<YOUR_PIN>/Homework1/
```

As a final step, run a script that will setup the permissions to your homework files, so we can access and run your code to grade it:

```
~coms4705/hw1_set_permissions.sh <YOUR_PIN>
```

Make sure the command above runs without errors, and **do not make any changes or run the code again**. If you do run the code again or make any changes, you need to run the permissions script again. Submissions without the correct permissions may incur some grading penalty.