COMS 4705 – Natural Language Processing – Summer 2017
Assignment 1
Language Modeling and Part of Speech Tagging

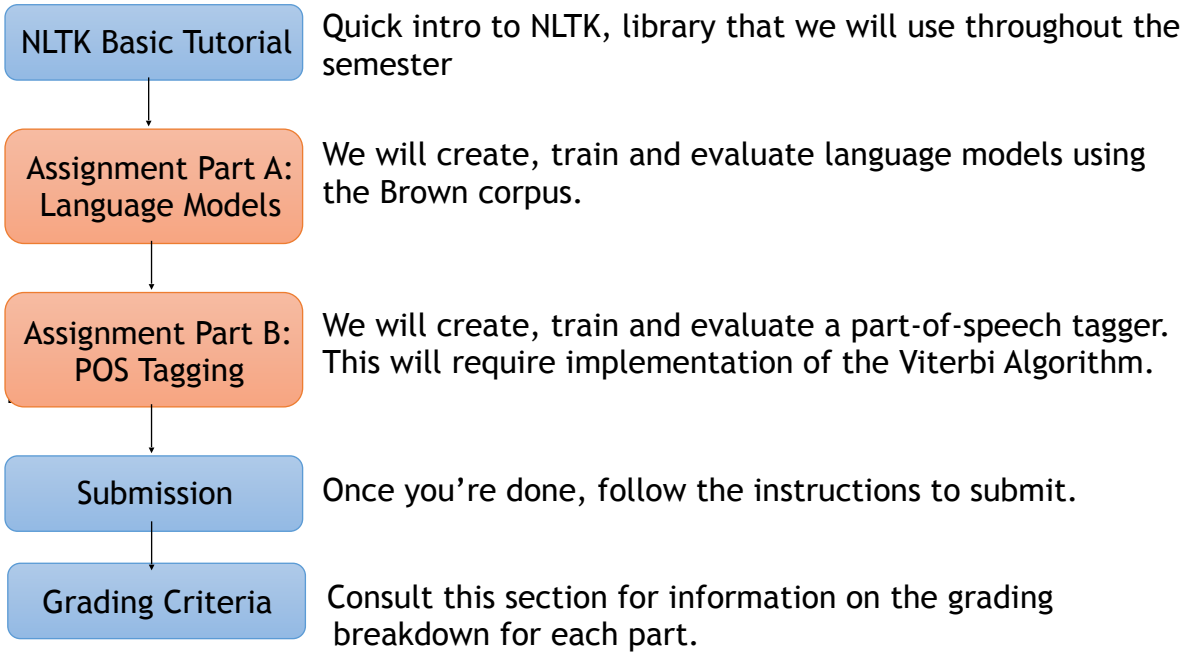
Due: Monday, June 26th, 11:59 PM

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:



Provided Files and Report

Now let's go back to the assignment folder you downloaded on Courseworks. 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 code). 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 near

BIGRAM near the

TRIGRAM near the ecliptic

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 and the **calc_perplexity()** 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 implement the calc_perplexity function in solutionsA.py and then run our perplexity script, "perplexity.py" on each of these files. This script simply calls the calc_perplexity function. The function uses count the words of the corpus and the log-probabilities computed by you to calculate the total perplexity of the corpus. Perplexity is basically a measurement of usefulness of a language model. See Ch. 4 "Language Modeling with N-Grams" of *Speech and Language Processing* for a discussion of perplexity.

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 NOUN

   TRIGRAM DET NUM NOUN

   TRIGRAM NOUN PRT CONJ

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)**:

   *** ***

   midnight NOUN

   Place VERB

   primary ADJ

   STOP STOP

   RARE VERB

   RARE X

5) Now, implement the forward algorithm for HMM taggers. This algorithm aims to compute the likelihood of a specific word sequence being generated by a HMM. You can read more about the algorithm in section 9.3 of *Speech and Language Processing*. Using the emission and trigram probabilities from the previous problems, calculate the prob-

ability of the word sequences in “Brown_dev.txt.” This corresponds to implementing the **forward()** function. Your probabilities will be output to “B5.txt”.

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.

Note: your book uses the term “state observation likelihood” for “emission probability” and the term “transition probability” for “trigram probability.”

Include the probability of the first word sequence in your README.

6) 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 section 9.4 in of the book.

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 “B6.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.

As above, 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/B6.txt data/Brown_tagged_dev.txt
```

This is the result we got with our implementation of the Viterbi algorithm:

Percent correct tags: 93.3249946254

7) 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. To use the script, run the following command:

   ```
   python pos.py output/B7.txt data/Brown_tagged_dev.txt
   ```

   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 courseworks.

Grading Criteria

Correctness: 90 points

Part A: 35 points

Question	1	2	3	4	5
Points	5	10	10	5	5

Part B: 55 points

Question	1	2	3	4	5	6	7
Points	0	10	0	10	10	15	10

Grading criteria for correctness

We will calculate the percent difference between your answers and the correct values. The percent correctness of a question maps to a percentage of points earned for that question as follows:

% Difference	Question Points
<= 1%	100
1%-3%	90
3%-10%	80
10%-30%	50
> 30%	25

Design and Documentation: 10 points

Running time	Part A	Part B
	5	5

Running time:

Part A: 100% if running time<=5 minutes, otherwise 0%. Part B: 100% if running time<=10 minutes, otherwise 0%.

Quality of README:

The instructions must be followed properly and the report must be clear, with the answers easily identifiable.

Late day policy

10% off for each day. After three days, the assignment will be given a score of zero.