CS447: Natural Language Processing	Fall 2018
Homework 2	
Handed Out: 9/28/2018	*Due: 11:59pm, 10/19/2018*

Please submit an archive of your solution (including code) on Compass by 11:59pm on the due date. Please document your code where necessary.

Getting started

All files that are necessary to do the assignment are contained in a tarball which you can get from:

`https://courses.engr.illinois.edu/cs447/secure/cs447_HW2.tar.gz`

You need to unpack this tarball (`tar -zxvf cs447_HW2.tar.gz`) to get a directory that contains the code and data you will need for this homework.

NLTK

For this assignment you will need the Natural Language Toolkit (`http://nltk.org/`). NLTK 3.0.0b is available as part of Canopy, which can either be loaded on the EWS machines or installed locally from `www.enthought.com/product/canopy`. To load canopy on EWS machines, use the command:

```
module load canopy
```

The provided scripts work only with NLTK 3.0.0 (installed independently or as a part of Canopy). If you do not want to use Canopy and already have a more recent version of NLTK installed[1], you can install pip via the instructions provided in Homework 0, and can use the following commands:

```
pip uninstall nltk
pip install -Iv --user nltk==3.0.0
```

This will tell pip to uninstall your current version of NLTK and then download and install NLTK 3.0.0.

Part 1: Hidden Markov Model Part of Speech Taggers (6 points)

1.1 Goal

The first part of the assignment asks you to implement a Hidden Markov Model (HMM) model for part of speech tagging. Specifically, you need to accomplish the following:

- implement the Viterbi algorithm for a bigram HMM tagger
- train this tagger on labeled training data (`train.txt`), and use it to tag unseen test data (`test.txt`).
- write another program that compares the output of your tagger with the gold standard for the test data (`gold.txt`) to compute the over overall token accuracy of your tagger. In addition, you need to compute the precision and recall for each tag and produce a confusion matrix.

1.2 Data

You will use three data files to train your tagger, produce Viterbi tags for unlabeled data, and evaluate your tagger's Viterbi output.

[1] NLTK 3.0.4, for example, will not successfully execute our scripts

1.2.1 Training

The file `train.txt` consists of POS-tagged text:

```
Kemper_NNP Financial_NNP Services_NNPS Inc._NNP ,_, charging_VBG that_DT program_NN trading_NN is_VBZ
ruining_VBG the_DT stock_NN market_NN ,_, cut_VBD off_RP four_CD big_JJ Wall_NNP Street_NNP firms_NNS
from_IN doing_VBG any_DT of_IN its_PRP$ stock-trading_NN business_NN ._.
The_DT move_NN is_VBZ the_DT biggest_JJS salvo_NN yet_RB in_IN the_DT renewed_VBN outcry_NN against_IN
program_NN trading_NN ,_, with_IN Kemper_NNP putting_VBG its_PRP$ money_NN --_: the_DT millions_NNS of_IN
dollars_NNS  in_IN commissions_NNS it_PRP generates_VBZ each_DT year_NN --_: where_WRB its_PRP$ mouth_NN is_VBZ ._.
```

Words are separated by spaces. Tags are attached to words, with a single underscore between them. In the actual file, each line is a single sentence. You will use this file to train a bigram HMM tagger (i.e. estimate its transition, emission and initial probabilities). For the purposes of this assignment, you need to use an UNK token to allow unseen words; use a threshold of 5. Additionally, you need to smooth the transition probabilities using Laplace smoothing (Lecture 4, slides 17-22). This allows your model to assign non-zero probability to unseen tag bigrams (which do exist in the test corpus).

1.2.2 Tagging

The file `test.txt` consists of raw text. Each line is a single sentence, and words are separated by spaces. Once you finish your implementation, run your tagger over `test.txt` and create a file `out.txt` that contains the Viterbi output.

1.2.3 Evaluation

For evaluation purposes, you need to compare the output of your tagger with `gold.txt`.

1.3 Provided Code/What you need to implement

Your solution will consist of two files, `hw2_hmm.py` and `hw2_eval_hmm.py`.

1.3.1 HMM tagger

You should implement your HMM tagger in the module `hw2_hmm.py`. You are free to implement your HMM tagger using whatever data structures you wish, but all of your code should be your own[2]. We provide a skeleton implementation for the `HMM` class with training and testing methods:

> `train(self, corpus)`: estimates the counts for the bigram HMM on a labeled training `corpus`, which is a nested list of `TaggedWord` objects[3] **(1 point)**
>
> `test(self, corpus)`: given unlabeled `corpus` (a nested list of word strings), print each sentence with its Viterbi tag sequence. This function is already implemented for you.

The tagging method requires you to implement the Viterbi algorithm for bigram HMMs:

> `viterbi(self, sen)`: given `sen` (a list of words), returns the Viterbi tag sequence for that sentence as a list of strings **(2 points)**

Implementation Hint: If you have a lexicon that tells you for each word which tags it can have, you don't have to go through all cells in each column. Similarly, if you first check whether a word has zero probability given a tag, you don't have to do all the computations to fill that cell. All your computations should again be in log space.

You may implement these functions however you wish, but please preserve the method signatures for autograding.

[2] Feel free to re-use your distribution code from HW1, if you like.

[3] A `TaggedWord` simply stores a word token and a POS tag (e.g. from `train.txt` or `gold.txt`) as two separate strings.

Note: The gold labels for the test data contain a transition ("`sharper_JJR $_$`") that is not present in the training data when using the default unknown word threshold of 5.[4] In order to prevent your tagger from assigning zero probability to a test sentence, you can use add-one smoothing on the transition distributions to ensure that a sequence with non-zero probability exists. Make sure that you still restrict the space of possible tags for each word token according to your tag lexicon.

1.3.2 Evaluation program

You should implement your evaluation program in `hw2_eval_hmm.py`. We ask that you implement the following methods for the `Eval` class:

`getTokenAccuracy(self)`: returns the percentage of tokens that were correctly labeled by the tagger **(0.5 points)**

`getSentenceAccuracy(self)`: returns the percentage of sentences where each word in a sentence was correctly labeled **(0.5 points)**

`getPrecision(self, tagTi)`: return the tagger's precision when predicting tag t_i **(0.5 points)**

Precision indicates how often t_i was the correct tag when the tagger output t_i

`getRecall(self, tagTj)`: return the tagger's recall for predicting gold tag t_j **(0.5 points)**

Recall indicates how often the tagger output t_j when t_j was the correct tag.

`writeConfusionMatrix(self, outFile)`: writes a confusion matrix to `outFile` **(1 point)**

We'd like you to create a confusion matrix that indicates how often words that are supposed to have tag t_i were assigned tag t_j (when $t_i = t_j$, a token is correctly labeled). The first line of your confusion matrix should be a comma- or tab-separated list of POS tags (`NN`, `NNP`, etc.); the j^{th} entry in this list indicates that the j^{th} column in the matrix stores the number of times that another tag was tagged as t_j. The first entry in the i^{th} following row should be tag t_i, followed by the counts for each t_j. Thus, the diagonal of the confusion matrix indicates the number of times that each tag was labeled correctly. The entries in this confusion matrix should be raw frequencies (i.e., you don't need to normalize the counts). You can use the counts in your confusion matrix to calculate precision and recall for each tag.

1.3.3 Sanity check

The Python script `hmm_sanity_check.py` will run your code and evaluate your HMM tagger on a sample sentence. It will check that your tagger correctly predicts the POS tags for that sentence, and that the probabilities of the model are reasonably close to the TA implementation. The script will also test your evaluation program to make sure that the methods you implement there behave correctly. Note that passing the script does not mean that your code is 100% correct (but it's still a good sign!).

1.4 What to submit for Part 1

For this portion, you should submit your two program files along with your confusion matrix:

1. `hw2_hmm.py`: your completed Python module for POS tagging using a bigram HMM(see section 1.3.1)
2. `hw2_eval_hmm.py`: your completed Python module for evaluating your HMM's output (see section 1.3.2)
3. `confusion_matrix.txt`: a text file containing the confusion matrix you produced from your HMM's output

We will provide a copy of the corpora/input files to test your program during grading.

[4] Both '`sharper`' and '`$`' appear at least 5 times, and `JJR` and `$` are the only tags they appear with, respectively; however, nowhere in the training data is there a bigram tagged as `JJR-$`.

1.5 What you will be graded on

You will be graded according to the correctness of your HMM tagger[5] (3 points), as well as the correctness of your token and sentence accuracy functions (1 point), precision and recall functions (1 point) and confusion matrix (1 point).

Part 2: Writing a Context-Free Grammar (4 points)

2.1 Goal

Your first task is to write a context-free grammar that can parse a set of short sentences. You will be using parsing code from the NLTK for this portion of the assignment, and your grammar does not need to be in Chomsky Normal Form.

2.2 Data

The file `sentences.txt` contains an evaluation set of 25 short sentences.

2.3 Provided code

We have provided a script (`hw2_nltkcfg.py`) that reads your grammar from `mygrammar.cfg` and tries to parse the evaluation set. The script expects the `sentences.txt` file to be in the same directory, and will write its output to a new file, `hw2_cfg_out.txt`. Call the script from the command line with:

```
python3 hw2_nltkcfg.py
```

For each sentence, the script will provide a brief message in the console (whether the parse was a `SUCCESS` or a `FAILURE`), along with the number of successful parse trees. More importantly, each successful parse tree will be written to the output file in a bracketed, tabbed format. When evaluating your grammar, you should look at this output to find ways to improve your grammar.

2.4 What you need to implement

You don't need to write (or submit) any actual Python code for this part of the assignment. Instead, your task is to define the rules for your CFG in `mygrammar.cfg`. This file already contains the lexical rules required for these sentences (i.e., words are mapped to their POS tags), along with a start symbol for the grammar that generates the nonterminal symbol `S`.

Please do not change or delete the existing rules.

Instead, you will need to add rules such as:

```
S -> NP VP
NP -> DT N
N -> NN | NNS | ...
etc.
```

You may define your own inventory of nonterminals, but please do not change the existing rules. Your grade will depend in part on your grammar's ability to successfully parse the evaluation sentences; however, we care more about your ability to define and explain a linguistically meaningful grammar (see below).

[5] Your tagger should not get 100% accuracy; correctness will be judged by comparing your Viterbi predictions against the predictions of the TA's tagger, allowing for noise due to ties and such.

Implementation hints

While it's possible to define a very simple grammar that parses every sentence, such as:

```
S -> LEX | LEX S
LEX -> NN | NNS | COMMA | VB | ...
```

that's not the point of this part of the assignment, and such a solution would receive zero credit. Similarly, a set of rules like:

```
S -> NN CC NN VBD FS
S -> PRP VBP NNS CC NNS FS
etc.
```

where each (extremely flat) rule generates the POS tag sequence for a particular sentence in the evaluation data is also not allowed. Instead, your grammar should use a set of nonterminals similar to those of the Penn Treebank parse trees you've seen in class. You can find a description of the POS tags used in this assignment at `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

2.5 What to submit

Along with your completed grammar file (`mygrammar.cfg`), you should submit your output file (`hw2_cfg_out.txt`). Additionally, you must answer the following discussion questions on Compass when you submit your assignment. These questions can be found by navigating to *Course Content* →*Homeworks* →*Homework 2* →*Homework 2 CFG Discussion Questions*

1. Describe how your grammar analyzes noun phrases. (0.5pts)
2. Describe how your grammar analyzes verb phrases. (0.5pts)
3. Which sentence has the most parses (please include the number of parses)? What about the sentences (besides their length) contributes to the large number of parses? (1.0pts)

2.6 What you will be graded on

You will be graded on your grammar's ability to successfully parse the evaluation sentences (1.0 pts) and on the quality[6] of your grammar (1.0 pts). Your answers to the discussion questions on Compass provide the rest of the grade (2.0 pts).

Submission guidelines

You should submit your solution as a compressed tarball on Compass; to do this, save your files in a directory called *abc123*_hw2 (where *abc123* is your NetID) and create the archive from its parent directory (`tar -zcvf` *abc123*`_hw2.tar.gz` *abc123*`_hw2`). Please include the following files:

1. `hw2_hmm.py`: your completed Python module for POS tagging using a bigram HMM (see section 1.3.1)
2. `hw2_eval_hmm.py`: your completed Python module for evaluating your HMM's output (see section 1.3.2)
3. `confusion_matrix.txt`: a text file containing the confusion matrix you produced from your HMM's output
4. `mygrammar.cfg`: your final CFG for parsing the sentences from Part 1
5. `hw2_cfg_out.txt`: the list of trees produced by your grammar on the evaluation sentences in Part 1 (the output of `hw2_nltkcfg.py`)

[6] See the **Implementation hints** in section 2.4

6. `README.txt`: a text file containing a summary of your solution

Additionally, you must answer the following discussion questions on Compass when you submit your assignment. These questions can be found by navigating to *Course Content* →*Homeworks* →*Homework 2* →*Homework 2 CFG Discussion Questions*

Appendix: Using NLTK on EWS Machines

For a friction-less environment setup on your EWS machine, we recommend loading NLTK from canopy which is already installed for your convenience, and using the default python3 interpreter as follows:

```
[zpahuja2@linux-a2 ~]$ module load canopy
[zpahuja2@linux-a2 ~]$ pip install -Iv --user nltk==3.0.0
[zpahuja2@linux-a2 ~]$ python3
Python 3.4.9 (default, Aug 14 2018, 21:28:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import nltk
>>> nltk.__version__
'3.0.0'
```

Note that it comes with NLTK 3.0.0 and Python 3.4.9, which is the environment configuration of the auto-grader we will use to grade your submission. We have made this choice for it is the simplest way for you to get started with NLTK. If you are using Python 3.5.2 on your local machine, it should be acceptable as well in most, if not all, cases but we highly encourage you to test your code on EWS before submission.