CS447: Natural Language Processing — Fall 2017

Homework 2

Handed Out: 10/06/2017 — *Due: 10pm, 10/27/2017*

Please submit an archive of your solution (including code) on Compass by 10:00pm on the due date. Please document your code where necessary.

Getting started

Note: *we will not be using NLTK for this assignment, and your code will be evaluated in an environment that does not have NLTK installed.*

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

`http://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.

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 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.

1.2.1 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 with a frequency 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[1]. 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[2] **(0.5 points)**
>
> `test(self, corpus)`: given unlabeled `corpus` (a nested list of word strings), print each sentence with its Viterbi tag sequence **(0.5 points)**

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.

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.[3] 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)**

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

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

[3] 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-$`.

`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 (Lecture 6, slide 27) 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.

1.5 What you will be graded on

You will be graded according to the correctness of your HMM tagger[4] (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).

[4]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.

Part 2: Pointwise Mutual Information (4 points)

2.1 Goal

The second part of the assignment asks you to use pointwise mutual information (PMI, Lecture **11**) to identify correlated pairs of words within the corpora.

2.1.1 Definition of $\mathbf{P(w_i)}$ and $\mathbf{P(w_i, w_j)}$

There are many ways to define the probability of an event or pair of events. For this assignment, we are interested in the probability of observing each word (or pair of words) within a single sentence. We consider a word type to be observed in a sentence if it occurs at least once; multiple occurrences of the same word within a single sentence still only count as one observed event. Thus:

> $\mathbf{P(w_i)}$ is the probability that a sentence S drawn uniformly at random from the corpus contains at least one occurence of word w_i
>
> $\mathbf{P(w_i, w_j)}$ is the probability that a sentence S drawn uniformly at random from the corpus contains at least one occurence of word w_i *and* at least one occurence of word w_j

2.2 Data

The data for this part of the assignment is stored in `movies.txt`, the training corpus from Homework 1.

2.3 Provided Code

We have provided the module `hw2_pmi.py`; This file contains code for reading in the corpus and an empty definition for your PMI class. It is up to you how you will implement the required functionality, but at the very least your code should provide the methods described in the next section.

We have provided two helper methods in the PMI class:

> `pair(self, w1, w2)`: returns a pair (2-tuple) of words in sorted order; for any two words `w1` and `w2`, `pair(w1, w2) == pair(w2, w1)`

You can use the `pair` method to prevent duplicate word pair entries.

> `writePairsToFile(self, numPairs, wordPairs, filename)`: given `wordPairs` (a list of PMI value/word pair triples, produced by the `getPairsWithMaximumPMI` method described in 2.4.3), this method calculates each pair's PMI and writes the first `numPairs` entries to the file specified by `filename`

Using these methods and the others you will implement in the next section, you should be able to examine the effect of word frequency on word pair PMI.

2.4 What you need to implement

2.4.1 Calculating PMI (1 point)

You need to be able to calculate the PMI of a pair of words in the vocabulary.

> `getPMI(self, w1, w2)`: returns float `p`, where `p` is the pointwise mutual information for a pair of words.

Implementation hints Use `math.log(x, 2)` or `numpy.log2(x)` get the base-2 log of x.

You might be better off caching all of the relevant observed counts from the training data than calculating those quantities on the fly. Think about how you can use the `pair` method and the data structures you've seen so far to store co-occurrence counts for pairs of words.

2.4.2 Defining the vocabulary (0.5 points)

You need to be able to store the vocabulary of the training corpus, and obtain a list of words that occur in at least a certain number of sentences. The frequency cutoff `k` will have a large effect on the types of word pairs that have the highest PMI.

> `getVocabulary(self, k)`: returns a list of words L_w, where every word in L_w was observed in at least `k` different sentences in the training data.

Note: For Part 2, "frequency cutoff" refers to the number of *sentences* a word appears in, i.e. multiple occurences of a word in a single sentence only contribute a single count towards the frequency cutoff.

2.4.3 Finding pairs of words with high PMI (1 point)

Given a vocabulary of possible words, you need to be able to find the n unique pairs of words in that vocabulary that have the highest PMI.

> `getPairsWithMaximumPMI(self, words, n)`: given a vocabulary list (`words`), returns another list of the pairs of words that have the highest PMI (without repeated or duplicate pairs). Each entry in the returned list should be a triple `(pmiValue, w1, w2)`, where `pmiValue` is the PMI of the pair of words `(w1, w2)`.

Implementation hints You may want to use a heap (see module `heapq`) to efficiently maintain the list of top n candidates as you search through the set of possible word pairs. You definitely want to restrict your search to only those pairs that were observed in the training corpus, rather than the space of all possible word pairs.

2.5 Discussion (0.5 points per question)

You will submit your answers for this section on Compass.

Once you have finished implementing your model, you should be able to generate different sets of words using `getVocabulary`, and find the pairs of words within each vocabulary that have high PMI. What does the list of pairs look like if we consider fewer and fewer rare words?

Generate vocabularies using frequency cutoffs of e.g. 2, 5, 10, 50, 100, and 200, and compare the 100 word pairs that have the highest PMI for each vocabulary. Then consider the following questions, providing your answers on Compass through the **Homework 2 Pointwise Mutual Information Discussion Questions** test.

1. Which cutoff is most useful for identifying the first and last names of popular actors and actresses?
2. Which cutoff is most useful for identifying common English phrases?
3. Which cutoff is least useful for identifying common English phrases or names?

2.5.1 Sanity check

The Python script `pmi_sanity_check.py` will check your PMI methods.

2.6 What to submit

See the end of this document for full submission guidelines, but the files you must submit for this portion are:

> `hw2_pmi.py`: your completed Python module for calculating pointwise mutual information (see section 2.4)

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

You can include your word pair lists, but we may reproduce some of them on our own to verify the accuracy and speed of your code.

Submission guidelines

You should submit your solution as a compressed tarball on Compass; to do this, save your files in a directory called *abc123*_cs447_hw2 (where *abc123* is your NetID) and create the archive from its parent directory (`tar -zcvf abc123_cs447_hw2.tar.gz abc123_cs447_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. `hw2_pmi.py`: your completed Python module for calculating pointwise mutual information (see section 2.4)
5. `README.txt`: a text file containing a summary of your solution