CS447: Natural Language Processing — Fall 2018

Homework 3

Handed Out: 10/19/2018 — *Due: 11:59pm, 11/09/2018*

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/HW/cs447_hw3.tar.gz`

You need to unpack this tarball (`tar -zxvf cs447_hw3.tar.gz`) to get a directory that contains the code and data you will need for this homework.

Part 1: Pointwise Mutual Information (4 points)

1.1 Goal

The first part of the assignment asks you to use pointwise mutual information (PMI) to identify correlated pairs of words within the corpora.

1.1.1 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

1.2 Data

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

1.3 Provided Code

We have provided the module `hw3_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 1.4.3), 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.

1.4 What you need to implement

1.4.1 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)` to 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.

1.4.2 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 1, "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.

1.4.3 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.

1.5 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 3 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?

1.5.1 Sanity check

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

1.6 What to submit

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

`hw3_pmi.py`: your completed Python module for calculating pointwise mutual information (see section 1.4)

`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.

Part 2: Parsing with Probabilistic Context-Free Grammars (6 points)

2.1 Goal

Your second task is to implement the CKY algorithm for parsing using a probabilistic context-free grammar (PCFG). Given a PCFG, the algorithm uses dynamic programming to return the most-likely parse for an input sentence.

2.2 Data

We have given you a text file (`toygrammar.pcfg`) containing a toy grammar that generates sentences similar to the running examples seen in class:

```
the woman eats the sushi with the tuna
the woman eats some sushi with the chopsticks
the woman eats some sushi with a man
etc.
```

Each binary rule in the grammar is stored on a separate line, in the following format:

```
prob P -> LC RC
```

where *prob* is the rule's probability, P is the left-hand side of the rule (a parent nonterminal), and LC and RC are the left and right children, respectively.

For unary rules, we only have a single child C and the line has format:

```
prob P -> C
```

We provide code for reading this file format to produce the equivalent `PCFG` object in Python.

2.3 Provided code

We provide the module `hw3_pcfg.py`, which contains several classes that may be useful for chart parsing. You should look over the source code yourself, but a brief summary of these classes includes:

`Rule`: a grammatical `Rule` has a probability and a parent category, and is extended by `UnaryRule` and `BinaryRule`.

`UnaryRule`: a `UnaryRule` is a `Rule` with only a single child (word or nonterminal).

`BinaryRule`: a `BinaryRule` is a `Rule` with two children.

`Item`: an `Item` is a chart entry that stores a label and a Viterbi probability. It is extended by `LeafItem` and `InternalItem`.

`LeafItem`: a `LeafItem` is a chart entry that corresponds to a leaf in the parse tree. For a leaf, the label is a word and the Viterbi probability is 1.0.

`InternalItem`: an `InternalItem` is a chart entry that corresponds to a nonterminal with a particular span in the parse tree. Its label is a nonterminal symbol, and its Viterbi probability is the probability of the best subtree rooted by that symbol (for this span). Its number of parses counts the number of possible trees rooted at the label for this span. Additionally, an `InternalItem` maintains a tuple of pointers to its children (if the children were generated by a `BinaryRule`) or single child (if generated by a `UnaryRule`). This tuple is a backpointer for the Viterbi parse.

`PCFG`: a `PCFG` maintains a collection of `Rules`; for your CKY algorithm, the rules are sorted by their right-hand-side in the `ckyRules` dictionary.

2.4 What you need to implement

Your task is to finish implementing the CKY method in `PCFG`:

`CKY(self, sentence)`: given `sentence` (a list of word strings), return the root of the Viterbi parse tree (i.e. an InternalItem with label `TOP`[1] whose probability is the Viterbi probability). By recursing on the children of this item, we should be able to get the complete Viterbi tree. If no such tree exists, return `None` (a parse failure).

In order to finish the CKY method implementation, you must also modify the constructor for `InternalItem` to correctly set the number of parses. We also provide the stubs for the `Chart` and `Cell` classes to help guide you in your implementation of the CKY chart.

Implementation hints

In Python, to obtain a tuple for a singleton value `v`, use `(v,)` instead of `(v)`. You will need this for the `InternalItem`'s child pointer when applying unary rules.

While our grammar is small, we still want you to implement your CKY algorithm using log probabilities (using `math.log(prob)` with the default base). It is good practice for implementing an actual NLP system.

You will probably want to define submethods for your CKY algorithm, as well as data structures to represent the parse chart or individual cells (each covering a particular span). You are welcome to add any functionality you need, as long as your `CKY` method returns an `InternalItem` object as we have defined it.

When filling the cells of your chart, you should add all possible items that can generate the span using a binary rule before adding items that can generate the span using a unary rule. Keep in mind that you only check for unary rules once per cell; our grammar does not allow chains of unary rules within a cell.

[1] The start symbol for the grammar

2.4.1 Test script

We have provided a hardcoded test script to check your implementation. After you have implemented your `CKY` algorithm in `hw3_pcfg.py`, run

```
python hw3_pcfg_test.py
```

to evaluate your parser.

2.5 What to submit

The only file you need to submit for this part is your completed `hw3_pcfg.py` program. We will evaluate your code using a test harness similar to the one provided (all files will be in the same format and we will initialize your `PCFG` in the same way, but we may use a different grammar and test sentences).

2.6 What you will be graded on

The grade for your implementation of the CKY algorithm will be based on returning the correct Viterbi trees (2.0 pts) and probabilities (2.0 pts) for our test harness, as well as the correct number of parse trees for each sentence (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*_hw3 (where *abc123* is your NetID) and create the archive from its parent directory (`tar -czvf` *abc123*`_hw3.tar.gz` *abc123*`_hw3`). Please include the following files:

1. `hw3_pmi.py`: your completed Python module for calculating pointwise mutual information (see section 1.4)
2. `README.txt`: a text file containing a summary of your solution
3. `hw3_pcfg.py`: your completed Python module for parsing using CKY and PCFGs in Part 2

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 3* →*Homework 3 Pointwise Mutual Information Discussion Questions*