CS447: Natural Language Processing — Fall 2015

Homework 3

Handed Out: 10/22/2015 — *Due: 10pm, 11/12/2015*

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

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.

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 via the instructions provided in Homework 0 (`https://courses.engr.illinois.edu/cs447/HW/cs447_HW0.zip`).

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:

```
sudo pip uninstall nltk
sudo pip install -Iv 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: Writing a Context-Free Grammar (4 points)

1.1 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.

1.2 Data

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

1.3 Provided code

We have provided a script (`hw3_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, `hw3_cfg_out.txt`. Call the script from the command line with:

```
python hw3_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.

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

Debugging

Additionally, you can provide `--gui` as an optional argument for debugging:

```
python hw3_nltkcfg.py --gui
```

With `--gui` enabled, the script will display a pop-up window with each successful parse as a tree. In order to view the next parse tree, you must close the window. We provide minimal debugging support for looking at partial charts in the event of a parse failure. If you feel that you need to see this information to debug your grammar, you are free to add additional debugging statements to `hw3_nltkcfg.py`.

1.4 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).

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`.

1.5 What to submit

Along with your completed grammar file (`mygrammar.cfg`), you should submit your output file (`hw3_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 3* →*Homework 3 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)

1.6 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[2] of your grammar (1.0 pts). Your answers to the discussion questions on Compass provide the rest of the grade (2.0 pts).

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.

[2] See the **Implementation hints** in section 1.4

`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`[3] 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.

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).

[3] The start symbol for the grammar

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. `mygrammar.cfg`: your final CFG for parsing the sentences from Part 1
2. `hw3_cfg_out.txt`: the list of trees produced by your grammar on the evaluation sentences in Part 1 (the output of `hw3_nltkcfg.py`)
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 CFG Discussion Questions*