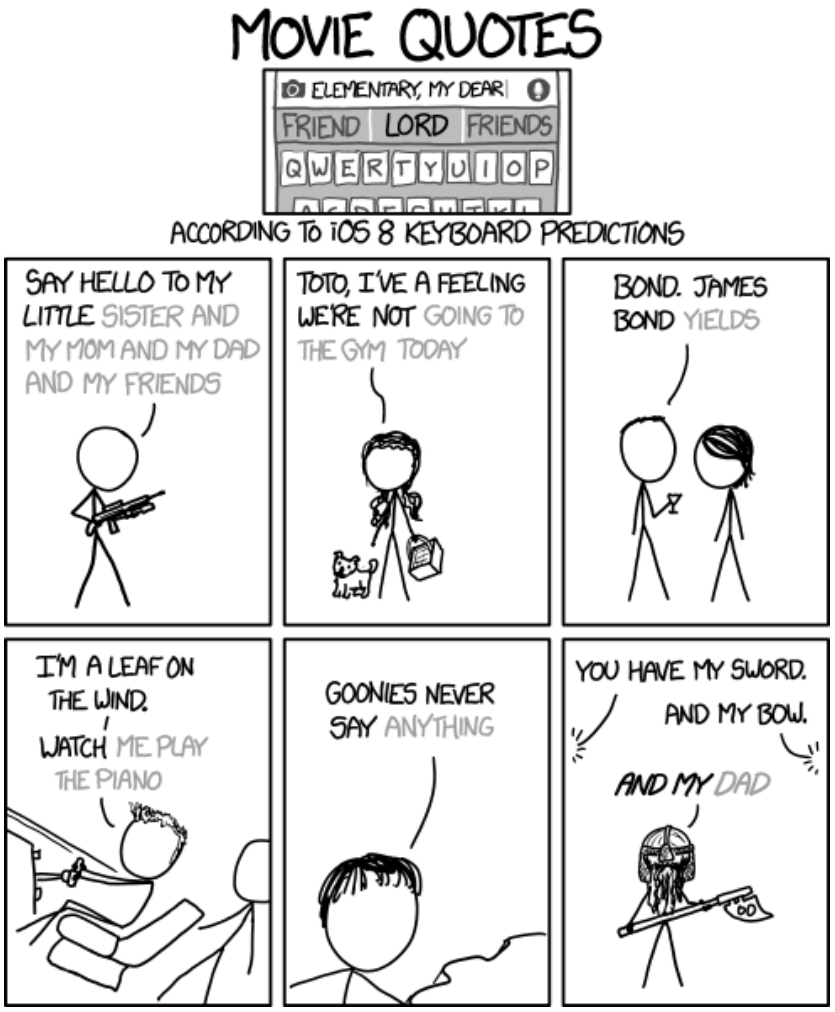
CS159 - Assignment 4

Due: Wednesday, October 7, 11:59pm (any time zone)



http://www.xkcd.com/1427/

In the last assignment we extracted PCFG rules using maximum likelihood estimation from an existing treebank and then binarized the grammar. In this assignment, we will now take a binarized grammar and use it to parse a new sentence using the CKY parsing algorithm.

For this assignment, I am again giving you a fair amount of flexibility for how you implement it. You may use whatever programming language you like (within reason – check with me if you plan to use something that is out of the ordinary). I have, however, provided you with a `GrammarRule`

class in Java that is similar to the one used for the last assignment (except it uses weights instead of probabilities).

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, e-mail me asap and I will try and pair you up.

As always, read through *the whole handout* before starting coding. In particular, I've given you a number of hints that should help direct you.

1 CKY parsing

Create a CKY parser. Your parser should be initialized based on a binary grammar. Once it is initialized, it should be able to parse multiple sentences without having to reread the grammar. Your parser must be efficient, which means you need to be efficient about many of the lookup operations in the algorithm.

To get you some very basic experience playing with command-line scripts (and because I'm allowing you to use any language you want), along with your code, you will also be submitting a script called `parse.sh` that allows you to run your parser from the command-line. Your script should take two command-line parameters as input. The first, should be the name of the grammar file and the second, the name of a file consisting of sentences to be parsed, with one sentence per line. Your script should then output the most-likely parse for each sentence along with the score for that parse separated by a tab, one per line. The parse tree output should be the parenthesized format we saw before for assignment 2. If the sentence does not have a parse, just output `NULL`.

For example, the following very basic script would work for a java version:

```
#! /bin/bash

java -Xmx1G -cp <where_code_base_is> nlp.parser.CKYParser $1 $2
```

assuming that the Java parser took two parameters from the command-line. `-cp <where_code_base_is>` specifies the classpath. If this is confusing or you have troubles with this, please come talk to me sooner than later (don't wait until the last minute to ask me about this!).

As an example, if we tried to parse the following sentences using `example.pcfg`:

```
Mary likes John .
John codes with John .
Mary likes to code .
write giant programs .
giant programs write John .
```

We would see the output:

```
(S (X3 (NP (NNP Mary)) (VP (VB likes) (NP (NNP John)))) (. .))  -1.9
(S (X3 (NP (NNP John)) (VP (VB codes) (PP (IN with) (NP (NNP John))))) (. .))   -2.1
NULL
(S (VP (VB write) (NP (JJ giant) (NNS programs))) (. .))        -2.3
(S (X3 (NP (JJ giant) (NNS programs)) (VP (VB write) (NP (NNP John)))) (. .))   -2.35
```

Please make sure that you follow the output specification exactly!

When you think you have it all working, parse the file `test.sentences` with the large grammar file `full.pcfg`. My parser takes less than a minute to parse these 10 sentences (though one doesn't have a parse). If you're not careful about efficiency, yours can take much, much longer.

Grammar rules

For the last assignment, to keep debugging simpler, we learned PCFG rules. However, if you try and parse with PCFG rules you can have underflow problems as you start to multiply together small probabilities. Instead, we're going to use rules that have a score/weight associated with them. In our case, the weights will be the log of the probabilities, but in general, we could use any arbitrary weights. Unlike probabilities, with weights the score of a parse is now the **sum** of the weights of the associated rules, and we would still like to pick the parse tree with the largest sum. For log probs, these sums will be negative.

2 Provided Code/Data

I have provided you with some resources to get you started:

`http://www.cs.pomona.edu/classes/cs159/assignments/assign4-starter.zip`

- **Code:** I have provided you with the same `GrammarRule` class as the last assignment, however, I've changed the comments to reflect the fact that we're now using weights (instead of probabilities).
- **Data:** I have provided you with a number of grammars and some sample output to get you started. `example.output` contains the parser output from parsing `example.input` using the grammar file `example.pcfg`. `full.pcfg` contains a very large grammar that should be able to parse most sentences (though we're not dealing with out of vocabulary, so if the sentence has a new word, it won't parse).

How to Proceed

The following is how I would suggest proceeding, though you're welcome to do it however makes the most sense for you:

1. Make sure you understand the CKY algorithm. We walked through the algorithm in class and the book provides pseudo-code. What information are you going to need to store in your table? All of the table access operations need to be fast ($O(1)$). How are you going to store things in the table to make this work?

2. In addition to the parse for 4a, consider working through a few other examples by hand to generate test cases. For example, you could do another sentence using the `example.pcfg` grammar or you could convert one of the grammars discussed in class to a weighted grammar (by taking the log of the probabilities) and redo one of the class examples. These will be important for debugging your program.

3. Write the constructor for your parsing class which will read in the grammar rules and store them in an efficient manner. I suggest storing them in three separate data structures, one for lexical rules, one for non-lexical unary rules and one for binary rules. Think about how you're going to be accessing each of these in the algorithm (look at the hints below for more on this).

4. Write a helper class representing one entry in your CKY table. What do you need to store? What types of questions will you need to support? How can you answer these questions efficiently? It may be simpler not to worry about storing the actual back references for reconstructing the parse and just worrying about the weights to start with.

5. Start writing your parsing method. Create a new CKY table (two dimensions) and fill in the diagonals based on the the lexical components. I would suggest writing a method that adds a constituent to your table (either in your entry class or as a standalone method). Print out the added constituents (either as they're added or by printing out the table) and make sure everything is added appropriately. You can compare against your hand-written example(s).

6. Add the check for unary rule applications in your add method. *Every* time you add a new constituent (whether it be from a binary **or** a unary rule) you need to check to see if any unary rules apply. Make sure to avoid infinite loops by checking that by adding the constituent, you're getting a better parse. In particular, if you're adding a constituent to an entry in the table that already has that constituent only add it if it has a better score. Also, make sure you've stored your unary rules in such a way that checking if any unary rule applies is *fast*. Check again against your hand-crafted test examples.

7. Write the main loops to fill in the rest of the table entries. If you're good about debugging your add method beforehand, you should just be able to focus on the loops and not on adding things to the table. You should be able to get a full parse now. Check your result against your hand example. Make sure the weights are right.

8. Work on actually reconstructing the parse. For each added constituent, you'll need to keep track of what subconstituents it came from. This may require creating a new data structure that's going to feel similar to a simplified version of `ParseTree`. You'll also probably need to modify your table entry class. Once you have the backpointers setup, try and print out the parse. Recursion will be your friend here. A parse tree is a recursive structure.

3 Hints

- If you need more memory (which you may) you can give the Java virtual machine more memory by adding `-Xmx1G` as a flag (which will set your heap at 1G). From the command line, just add it after `java`. From `Eclipse` you'll need to add it as a VM argument under `Run->Run Configurations...` and then under the `Arguments` menu.
- To get things to run efficiently, you'll need to use hashtables (e.g., HashMaps) in a number of places.
- You will also need to be careful about the algorithm decisions that you make if you want to be able to parse with the full grammar. One place where you can run into trouble is when applying the binary rules. When you're filling in the table with the binary rules, you're trying to create new constituents by combining constituents in some entry to the left, call it $entry_1$, and some entry below, call it $entry_2$. There are two ways you can see if you have a binary rule that matches:

  a. consider all pairwise combinations in $entry_1$ and $entry_2$ and see if any of those pairs match the right-hand side of any of the binary rules
  b. iterate through all of the binary rules and see if any of the rules match, that is RHS1 is in $entry_1$ and RHS2 is in $entry_2$

  One of these is approaches *much* faster than the other!
- Debug your code incrementally and make sure to print things out as you go. If you wait until the end to try and debug it, you're going to have a bad time.

- We're only looking for the best parse, which means for any entry in the table, we only need to store the best version (based on weight) of each constituent.
- If you get stuck on printing out the parse, take a look at the `ParseTree` class and its `toString` method from the last assignment. The idea should be fairly similar.
- Unary rules are tricky. Make sure that any time you add a new constituent, you add a unary rule, but be careful about not getting stuck in infinite loops.
- Spend some time making sure you understand the algorithm and the design of your code. It's not a ton of code to write, but it does take some thought figuring out how everything fits together.

4 Extra credit

There are two possibilities for extra credit on this assignment. Each is worth 3 points.

- **Real trees**: Right now, we're outputting binarized versions of the trees. We'd really like to get the original grammar trees back out. Add a flag `-original` to your `parse.sh` script that outputs original grammar trees. Make sure that the default behavior is still the binarized version. Include in your `output` folder a parsed version of `test.sentences` called `test.sentences.parsed.original` containing these new parses.
- **Beam search**: We can speed up the implementation of the parser at the sacrifice of optimality by doing beam search: at each cell, only keep the K most likely hypotheses. Decreasing K will result in faster parses, while increasing K will result in slower parses, but more likely to be correct. Add a flag `-beam <num>` to your `parse.sh` script that uses $K = num$ as the beam threshold.

5 Subtractive credit

Even if you get the test examples right, your code may still not be working properly. Some of the intricacies of unary rules, etc. are not thoroughly tested on this smaller example. If you thought through your algorithm and code design, you should be in good shape. However, I will "sell" you the weights of the correct, best parses for the first five test sentences for 3 points. That is, I will give you the weights (not the full parses), but you will receive a 3 point deduction. If you take this option, please do not share the answer with others in the class.

6 When you're done

Create a directory that will house all of your code and results for this project. To make things easier for me, include one sub-folder with your code called `code` and another sub-folder with your output called `output`, nested within the main directory. Your `parse.sh` script should be in the main directory.

If you worked with a partner, make sure both of your names are in the comments and only submit one copy, though, make sure to tag your partner in the submission system.

What to submit

In your `code` subdirectory, you should submit your full working code. If the naming of the files isn't obvious about where various things are being done, please include a `README` file explaining your organization.

In your `output` directory you should have a file call `test.sentences.parsed` representing your parse of `test.sentences` using `full.pcfg`.

In the main directory, include your `parse.sh` script for running your parser.

Please make sure to follow the folder outline and file naming conventions above.

As before, `zip` this folder and submit online through the course submission mechanism.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file you modify.
- Each class and method should have appropriate JavaDoc comments (doc strings if you do it in Python).
- If anything is complicated, put a short note in there to help me out if there are any issues.

This is a non-trivial assignment and it can get complicated, which makes code style and comments very important so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
Parser	50
Efficiency	10
style/commenting	5
extra credit	6
total	65 + 6 extra