Computational Linguistics I *CMSC723, Fall 2010*

P2: Parsing

Hand in at: `http://www.cs.utah.edu/~hal/handin.pl?course=cmsc723`.

Introduction

In this project, we'll explore probabilistic context free grammars and parsing as a way of modeling natural language syntax (in this case, English). You will use PCFGs for two tasks: generation and recognition (aka parsing). You will be evaluated based on (a) what percentage of your generated sentences are grammarical and (b) what percentage of your fellow student's grammatical sentences you can parse. (Plus how well you can parse some simple sentences we give you.)[1] As a final part of the project, you will built a broad-coverage parser for English by extracting rules from a treebank.

To make this project more self-contained, we will work with a limit vocabulary for 222 words, from the initial scenes of Monty Python and the Holy Grail. These are in the `allowed_words` file, and you *may not* generate sentences with words not in the file. Note that this data *is* case-sensitive. In `Vocab.gr` you will find a hand-built set of pre-terminal rules, together with weights (all 1). Some words here are ambiguous and you should feel free to use or not use whatever parts of this you want.

1 Warm-up for Generation *(10%)*

(Yeah that's right: you don't get much credit for this part: this is just to help you get things running properly!)

Consider the following PCFG:

```
0.8    S        S1 .
0.2    S        VP .

1.0    S1       NP VP

0.9    NP       Det Nbar
0.1    NP       Proper

0.9    VP       VerbT NP
0.1    VP       VP PP

0.9    Nbar     Noun
0.1    Nbar     Nbar PP

1.0    PP       Prep NP
```

This grammar (see `basic.gr`) is binary/unary, so should be relatively easy to work with. The pre-terminals are defined in `Vocab.gr`. (Note the trick we had to play with S/S1 in order to make sure that the grammar was binary – it would have been nicer to write "`S -> NP VP .` ", but that's not allowed!) Note that the

[1] Some of this assignment is modeled after the competitive grammar writing idea by Jason Eisner and Noah Smith: `http://www.clsp.jhu.edu/grammar-writing/`.

lexical rules in `Vocab.gr` are *not normalized*, so you'll have to normalize them to a probability distribution as you read them in.

Write a program that reads in one or more grammar files and generates sentences according to them. You program should just assume that any right hand side (RHS) term it hits that doesn't appear anywhere as a LHS is a terminal. For each random sentence you generate, print out its *log probability* (you MUST use natural log). Here are ten random sentences I generated:

```
% cat basic.gr Vocab.gr | ./generate.pl 10
-11.955       Guinevere rides this coconut .
-22.349       Sir_Bedevere has each corner across another defeater .
-13.025       another sovereign is the defeater .
-22.349       that pound on a defeater covers Patsy .
-13.025       a fruit drinks each king .
-11.954       this master carries Dingo .
-13.025       that weight carries every coconut .
-8.9590       carries any swallow .
-13.025       each story has each quest .
-13.025       that land has another chalice .
```

(Note that your program doesn't have to work like mine – that's just how I chose to write it.)

For test purposes, you can leave off the `Vocab.gr` and then it will generate strings of pre-terminals:

```
-0.74995      Det Noun VerbT Det Noun .
-0.74995      Det Noun VerbT Det Noun .
-2.84181      Proper VerbT Det Noun .
-0.74995      Det Noun VerbT Det Noun .
-0.74995      Det Noun VerbT Det Noun .
-3.26325      Det Noun VerbT Det Noun Prep Det Noun .
-3.26325      Det Noun VerbT Det Noun Prep Det Noun .
-0.74995      Det Noun VerbT Det Noun .
-2.84181      Proper VerbT Det Noun .
-7.44698      Det Noun Prep Proper VerbT Proper .
```

If you run yours and get markedly different outputs, or get the same "sentences" but with different log probabilities, then you have something funky going on!

(WU1) Give twenty randomly generated sentences (with words as terminals). Mark which ones you feel are more-or-less grammatical. What percentage is it? (This accounts for all the points for this section.)

2 Chart Parsing *(50%)*

In this section, we will implement a chart parser based on the CKY dynamic programming algorithm.

Your algorithm should read in one or more grammars, and a test file, and produce parses for each sentence (one per line) in the test file. For example, if `tst` contains:

```
Det Noun VerbT Det Noun .
Proper VerbT Proper .
Det Noun VerbT Det Noun .
VerbT Det Noun .
VerbT Det Noun .
Det Noun VerbT Proper .
Proper VerbT Det Noun .
Proper VerbT Det Noun .
Det Noun VerbT Det Noun .
VerbT Det Noun .
Det Det Det .
```

Then you should be able to run your parser to produce things like:

```
-0.7499   (S (S1 (NP Det (Nbar Noun)) (VP VerbT (NP Det (Nbar Noun)))) .)
-4.9336   (S (S1 (NP Proper) (VP VerbT (NP Proper))) .)
-0.7499   (S (S1 (NP Det (Nbar Noun)) (VP VerbT (NP Det (Nbar Noun)))) .)
-1.9255   (S (VP VerbT (NP Det (Nbar Noun))) .)
-1.9255   (S (VP VerbT (NP Det (Nbar Noun))) .)
-2.8418   (S (S1 (NP Det (Nbar Noun)) (VP VerbT (NP Proper))) .)
-2.8418   (S (S1 (NP Proper) (VP VerbT (NP Det (Nbar Noun)))) .)
-2.8418   (S (S1 (NP Proper) (VP VerbT (NP Det (Nbar Noun)))) .)
-0.7499   (S (S1 (NP Det (Nbar Noun)) (VP VerbT (NP Det (Nbar Noun)))) .)
-1.9255   (S (VP VerbT (NP Det (Nbar Noun))) .)
no parse!
```

In this example, all the sentences were generated by the generation from the previous section, except for the last one which I added by hand, and which (obviously!) has no parse under this grammar.

Helpful debugging hint: If you run your generate program, and then parse the output, you should get exactly the same (log) probabilities!!!

If you need some help debugging, it is often useful to print out the chart after your CKY algorithm has run. For example, on the first sentence in the test file above, my chart looks like the following at the end of parsing:

```
0  1  Det      0.0    X             X
0  2  NP      -0.211  Det Nbar      1
0  5  S1      -0.527  NP VP         2
0  6  S       -0.750  S1 .          5
1  2  Nbar    -0.105  Noun          X
1  2  Noun     0.0    X             X
2  3  VerbT    0.0    X             X
2  5  VP      -0.316  VerbT NP      3
2  6  S       -1.926  VP .          5
3  4  Det      0.0    X             X
3  5  NP      -0.211  Det Nbar      4
4  5  Nbar    -0.105  Noun          X
4  5  Noun     0.0    X             X
5  6  .        0.0    X             X
```

The columns here are: start position, end position, label, log probability, rule, and split point. For instance, the fourth line means that we found a “S” spanning 0 to 6, derived from “S1 .” and the split point was 5. (This means that the “S1” ran from 0 to 5 and the “.” from 5 to 6.) In some cases (eg., unary rules or terminals), the last two fields are missing, in which case you just see “X”.

You will be evaluated on the sentences in `basic.sentences`. You should parse these, and store the parse trees (in the same format as the example above) in `basic.sentences.parsed`. Use the grammar in the union of `basic.gr` and `Vocab.gr`. The scoring will be:

- Five percent for producing well-formed output (i.e., logprob then whitespace then properly formatted trees; or “no parse!” for sentences that cannot be parsed).
- Five percent for correctly printing “no parse!” for the sentences that are unparsable under the grammar. (Fractional points will be awarded, for instance if you print no parse for 80% of them, you’ll get 0.8.)
- Five percent for correctly printing parse trees for the sentences that are parsable under the grammar. (Fractional points as above.)
- Twenty percent for correct parse trees. (Fractional points are possible.)
- Fifteen percent for correct log probabilities (up to an error of ±0.01). (Fractional points are possible.)

If you’re worried about formatting, run the `validate.pl` script to make sure your output is conformant with our scoring:

```
% cat basic.sentences.parsed | ./validate.pl
```

If there is nothing wrong, it will just print out all the trees it reads in. However, if it encouters a line it cannot recognize, it will try to provide a not-completely-useless error message.

Implementation Note: I highly highly highly recommend not being dumb about how you implement CKY. In particular, my first implementation involved iterating over all start, end and mid positions, and then iterating over the entire grammar, checking to see if stuff matches. This is the usual way CKY is presented. It took about 11 seconds to parse a 13 word sentence. I then reimplemented it so that you could look up grammar rules by their right hand sides, and then just iterated over the elements in the chart, checking to see how to put them together in the grammar. It now runs in 1.4 seconds. It doesn’t matter for this part of the assignment, but you will be very unhappy in the last section of the assignment if you have something 10 times slower than it needs to be.

3 Grammar Writing *(15%)*

In this section, you will attempt to write the best possible grammar you can for our Monty Python vocabulary. Your first task should be to make sure that your grammar can handle all the sentences in `examples.txt`. The basic grammar I provided should only be able to parse the first two sentences. You don’t need to produce perfect parse trees (whatever that means), but they should be “reasonable.” (I.e., I can always just add `X -> X X` and `X -> word` for all words, to my grammar, to get perfect coverage.) Producing these trees is worth 3% of your grade.

For the rest, you need to just make the best grammar you can. There are two parts to this, which emphasize precision and recall of the grammar.

- Precision: Your grammar should *generate* as few ungrammatical sentences as possible. You will be partially scored based on the fraction of grammatical sentences that you generate. You could attain a perfect score by having a grammar that always generates a single, unique grammatical sentence.
- Recall: Your grammar should *recognize* as many grammatical sentences as possible, and assign them as high a probability as possible. This competes, of course, with a grammar that only generates a single sentence, since it will assign zero probability to any other sentence.

Your immediate question will be: where do these grammatical sentences come from? The answer is: your fellow classmate’s generated sentences!

Like in project 1, there will be a hacky online script for evaluating your grammars. You will be able to periodically (more frequently than in P1, but not constantly) able to upload a new grammar. When you upload a new grammar, two things will happen:

- We will run it on our current "test set" of sentences and allow you to download its output. You will be able to see from this (a) what sentences it cannot parse and (b) what sentences it assigns very low probabilities to. *Note: during the online process, not all sentences in the "test set" are guaranteed to be grammatical, so you'll have to look at them manually for a sanity check!*
- We will use it to generate new sentences to be added to the next "test set." Yes, this will include ungrammatical sentences in the test set, but that's life.

At the end of the project, we will use Amazon's Mechanical Turk to rate the grammaticality of a random sample of sentences generated by each user's grammar. Your *precision* will be your score according to this metric. We will score about 20-50 sentences per team, depending on how many teams there are and how expensive this turns out to be `:)`.

To compute recall, we will do the following. Your parser will be run on all sentences in the test set (including the ungrammatical ones). We will then take the 100 sentences to which your grammar assigns the highest probability (ties broken randomly, including "no parse!" cases). Your recall is the fraction of those that are grammatical (as judged by turkers).

You should spend enough time on this part to get something *decent*, but don't waste too much time. It seems that the basic grammar generates about 10% grammatical sentences. Scoring will be as follows:

- One point for a precision over 10%, another for over 20%, another for over 30%, up to five points for a precision over 50%.
- Same for recall: one to five points possible at the same cutoffs.

This means that you can get half credit for this part section by just having a grammar that generates a single grammatical sentence.

Extra credit: The team with the highest F-measure[2] will get eight extra percentage points. The second highest, five; and the third highest, three.

Note on scoring: If I'm really bad at anticipating the difficulty of this, and no team does better than, say, 30% on one of the measures, I'll rescale the measures in your favor. (I won't rescale the measures out of your favor.)

Hint: If you find it easier to write non-binary grammars, then I've provided a script `binarize.pl` that will take a non-binary grammar and output a binarized version. For example:

```
% cat nonbinary.gr
0.8 S    NP VP PP .
0.2 S    VP .

% cat nonbinary.gr | ./binarize.pl
0.8 S NP VP__PP__.
1.0 VP__PP__. VP PP__.
1.0 PP__. PP .
0.2 S    VP .
```

[2] F-measure $= 2PR/(P+R)$ is the harmonic mean of precision and recall.

(WU2) Briefly describe any big ideas that went into your grammar (if you had any), or if it's just a series of hacks... what worked out and what didn't work out? Write about a paragraph about your experience. (Worth 3% of your grade, though we reserve the right to give out bonus points for particularly clever grammars, even if they don't perform very well.)

Important Warning: It's possible to write a grammar that makes my machine hang (eg., add `S -> S` with probability 1). That makes me angry. As a result, my generator will stop recursing at a depth of 40, and will also not let your grammar run for more than a minite to generate 100 sentences. So you should try your grammar on your *own* first, make sure it works, *then* upload it. Your grammar should also tend to generate short-ish sentences. We will run it by generating 200 sentences, and then take the first 100 that are of length 20 or less. If it produces less than 100 of length 20 or less, it will be rejected.

The online system is at: `http://www.cs.utah.edu/~hal/tmp/grammar.pl`.

4 Treebank Parsing *(*25%*)*

I've provided a subset of the Wall Street Journal portion of the Penn Treebank in `wsj.gz`. I've also provided a script (`countRules.pl`) that will read in the treebank and spit out all the context free rules (non-binarized) that are used, together with their counts.

For example:

```
% zcat wsj.gz | head -n1
0       (S (NP (NNP "Mr.") (NNP "Vinken")) (VP (VBZ "is") (NP (NP (NN "chairman")) (PP
(IN "of") (NP (NP (NNP "Elsevier") (NNP "N.V.")) (, ",") (NP (DT "the") (NNP "Dutch")
(VBG "publishing") (NN "group")))))) (. "."))
```

This is the first tree. (I've put quotes around the terminals so that they can be distinguished easily from non-terminals, for instance in the case of the non-terminal ".".) We can get the rules from just this single tree by:

```
% zcat wsj.gz | head -n1 | ./countRules.pl
1       ,       ","
1       .       "."
1       DT      "the"
1       IN      "of"
1       NN      "chairman"
1       NN      "group"
1       NNP     "Elsevier"
1       NNP     "Dutch"
1       NNP     "N.V."
1       NNP     "Vinken"
1       NNP     "Mr."
2       NP      NNP NNP
1       NP      NP PP
1       NP      DT NNP VBG NN
1       NP      NP , NP
1       NP      NN
1       PP      IN NP
1       S       NP VP .
1       VBG     "publishing"
1       VBZ     "is"
1       VP      VBZ NP
```

Here, there was only one rule that was used twice: the one that expands an NP into two NNPs (NNP = proper noun).

If we run this on all of the WSJ text, we see some rules like the following (this will take 30 seconds or so to run):

```
% zcat wsj.gz | ./countRules.pl
...
28022   ,       ","
...
19583   S       NP VP
16175   S       VP
10785   S       NP VP .
1399    S       PP , NP VP .
...
20642   NP      NP PP
18715   NP      DT NN
13329   NP      PRP
...
```

All of these seem pretty reasonable.

(Note: you don't need to feel obligated to use my `countRules.pl` script: it's just there if you want it.)

If your parser from the previous section works generically, you should be able to say something like:

```
% zcat wsj.gz | ./countRules.pl | ./binarize.pl > wsj.gr

% cat test-file
"Mr." "Vinken" "is" "chairman" "of" "Elsevier" "N.V." "," "the" "Dutch" "publishing"
"group" "."

% cat wsj.gr | ./parse.pl test-file
-92.0275445523356       (S (NP (NNP "Mr.") (NNP "Vinken")) (VP__. (VP (VBZ "is") (NP__PP
(NP (NN "chairman")) (PP (IN "of") (NP (NP (NNP "Elsevier") (NNP "N.V.")) (,__NP (, ",")
(NP (DT "the") (JJ__NN__NN (JJ "Dutch") (NN__NN (NN "publishing") (NN "group")))))))))
(. ".")))

% cat wsj.gr | ./parse.pl test-file | ./debinarize.pl
-92.0275445523356       (S (NP (NNP "Mr.") (NNP "Vinken")) (VP (VBZ "is") (NP (NN
"chairman")) (PP (IN "of") (NP (NP (NNP "Elsevier") (NNP "N.V.")) (, ",") (NP (DT "the")
(JJ "Dutch") (NN "publishing") (NN "group"))))) (. "."))
```

The final step takes about 1.8 seconds on my machine, which is pretty reasonable for a 13 word sentence. In this case, $N^3 = 2197$ and $|G| = 55774$. If you implemented your parser stupidly (by iterating over the whole grammar) like I did the first time, it will take about two minutes.

One problem that comes up is trying to parse a sentence that isn't in the training data. If you run your parser on `dev.txt` (this takes me about 2 minutes), you should find that 28 of the 100 sentences do not have a parse under this grammar. We can see how well our parser is doing by evaluating our predicted parses against gold standard parses (before evaluating, don't forget to *debinarize!*):

```
% cat wsj.gr | ./parse.pl dev.txt | ./debinarize.pl > dev.pred
% ./evaluate.pl dev.truth dev.pred
```

```
# of true spans              = 2794
# of hypothesized spans      = 1960
# of correct spans           = 1945
# of correctly labeled spans = 1855
labeled precision            = 0.946428571428571
labeled recall               = 0.663922691481747
labeled f-measure            = 0.78039545645772
unlabeled precision          = 0.99234693877551
unlabeled recall             = 0.69613457408733
unlabeled f-measure          = 0.818258308792596
```

Probably the number we most care about is labeled f-measure, which is 74.4% in this case. Which isn't terrible, considering that we produced *nothing* for over a quarter of the trees!

Your real task is to try to make the parser faster by implementing pruning. In particular, I should be able to specify a paramter K to your parser, so that no cell in the chart ever contains more than K entries. A simple way to implement this (though probably not the fastest) is to fill up all chart cells for a given length, and then go back and prune them (I prune cells *before* applying unary rules, otherwise you might lose some backpointers and bad things will happen). For each cell, you'll find the top-K highest probability items, and remove everything else. (Of course, if there are fewer than K items in a chart cell, you don't need to do anything.)

If I set my pruning size $K = 2$, then my number of unparsable sentences goes up to 83, and my labeled f-measure goes down to 23%. However, it runs a heck of a lot faster (the first is the no-pruning version; the second prunes with $K = 2$):

```
% time cat wsj.gr | ./parse.pl dev.txt > /dev/null
real    1m24.025s
user    1m22.214s
sys     0m0.050s

% time cat wsj.gr | ./parse.pl -k 2 dev.txt > /dev/null
real    0m21.251s
user    0m21.180s
sys     0m0.017s
```

So it's basically 3-4 times faster. If yours *doesn't* get faster, you probably implemented CKY stupidly—I told you not to do that!

Note: The above results are *if you also prune the root cell in the chart.* This is actually kind of a stupid thing to do, as pointed out by some students in the class. It takes extra time and makes the results worse. If you prune everything *except* the root cell, your labeled F should be about 59% and you should have only about 38 unparsable sentences.

(WU3) Generate a plot with time-to-parse on the x-axis and labeled f-measure on the y-axis, for pruning sizes of $K \in \{1, 2, 3, 4, 5, \dots, 10\}$. Does the plot level off? Are there any values of K for which you find *fewer* parses on the development data?

You will be evaluated based on whether the trees you're producing are reasonable (all within a few percentage points F-measure from mine), and whether your graph looks reasonable.

5 Extra Credit: Smoothing and Improved Parsing

This entire section is extra credit. Feel free to do it if you want, or totally ignore it if you don't want. Basically you are tasked with making a better parser, and will be judged based on your performance on `test.txt`. Note that this file is pretty big, so you'll have to be a bit clever. One important aspect is that your parser must produce *something* for every sentence: saying "no parse" is no longer an option. To get this to work, you will need to smooth. You can do anything else you wish (aside from running other people's parsers, of course!) that doesn't use additional data. Below are some ideas; there are more in the book: add either full lexicalization or partial lexicalization (see a paper by Dan Klein and Chris Manning on "Unlexicalized Parsing" from a few years ago for ideas on how to do partial lexicalization); do better smoothing; do better binarization; use grand-parent information; use sibling information; do better POS tagging, eg., using suffix rules; handle OOV terms better; etc.

My parser, with no pruning, and nothing else special (basically the same thing you saw from before) gets the following results on the test data (this took me about 20 minutes):

```
% cat wsj.gr | ./parse.pl test.txt | ./debinarize.pl > test.pred
% ./evaluate.pl test.truth test.pred
# of true spans              = 29719
# of hypothesized spans      = 20852
# of correct spans           = 20500
# of correctly labeled spans = 19166
labeled precision            = 0.919144446575868
labeled recall               = 0.644907298361318
labeled f-measure            = 0.757983824721678
unlabeled precision          = 0.983119125263764
unlabeled recall             = 0.689794407618022
unlabeled f-measure          = 0.810741333966107
```

(Of course, you don't actually get to see `test.truth`!)

The scoring will be as follows. For each *two percent* you beat me by (my score is 75.798), you'll get 1% extra credit. Futhermore, the top team will get 8%, the next will get 5% and the third will get 3%. (Note: I only care about labeled F-measure for this.)

(WU4) Describe what you did and what your results are.

Like in P1, you can see how you're doing online: `http://www.cs.utah.edu/~hal/tmp/parser.pl`.

6 What To Hand In

Please hand in the following:

- `writeup.pdf` – A document answering all the WU questions.
- `partners.txt` – A file listing names and IDs of all members of your team.
- `basic.sentences.parsed` – Your parser's output on `basic.sentences`.
- `montypython.gr` – Your final grammar file for the grammar writing section of the project.
- `dev.pred` – Your parser's output (no pruning!) on the development data.
- `dev.{1,5,10}.pred` – Your parser's output with the indicated amount of pruning.
- `test.pred` – The best you can possibly do on the test data (this is extra credit only)