ULL 2015
Assignment 1
part 2

Assignment

In part 2 of assignment 1 you are gonna implement, train and evaluate a version of the DMV model by extracting a dependency grammar from a training corpus, transforming it into a PCFG using Headden's transform and using bitpar to train it with EM. In this document we will provide some pointers (the commands, as before, can also be found in the document a1-commands.txt). You can follow the following steps:

- Firstly, create a corpus to run your learning algorithm on. Most dependency algorithms are trained on sequences of POS-tags (why?), you will have to extract them from your corpus of choice. We have provided you with a gold standard (for unlabeled directed dependency parses) for all sentences from wsj01-21 without tags, traces and punctuation up to length 10, so it makes sense to pick (part of) this corpus.
- Generate an initial version of the grammar from the POS-tag sequences, for instance by first extracting all possible bependencies and then transform them into PCFG rules using Headden's appreach to model DMV.
- Use Bitpar to train the weights of your initial grammar (as in part 1 of the assignment).
- Parse the training sentences with your trained grammar (what should your corpus look like?).
- Transform the resulting parses back to dependency notation.
- Evaluate the quality of your parses against a gold standard.

Extracting POS-tag sequences up to length 10 from a treebank

You can extract POS-tag sequences from a treebank using sed and grep by removing phrasal nodes by matching everything between opening brackets (like '(NP (') and terminals by matching things that end with a closing bracket:

```
cat $input_treebank | sed 's/(/( /g;s/([^(]\+ (//g;s/[^ (]\+)//g;
s/[(]//g;s/ \+/ /g;s/^ \+//' | grep '[A-Za-z]' > $POS-tag-sequences
```

To filter out all sequences that have more than 10 tags, use:

```
cat $all_sequences | awk '{if (NF<11) print}' > $sequences_up_to_length_10
```

Generate all possible dependencies from a sequence of POS tags

You can generate all possible dependencies within a POS-tag sequence using awk:

```
cat $pos_tag_sequences |  awk '{for (i=1;i<=NF;i++)
{print ``ROOT'',$i, ``right''; for (j=i+1;j<=NF;j++)
{print $i,$j, ``right''; print $j,$i, ``left''}}}' > $all_dependencies
```

(You should check on a toy-corpus if this does what you want).

To avoid duplicates and get a very crude initialisation of your parameters, sort and count the dependencies you found:

```
less $all_dependencies | sort | uniq -c | sort -g -r -k 1 > $all_dependencies_sorted
```

Evaluating your dependency parses

To evaluate your results you have to take care of several things, some of which are not very trivial. For instance, transforming your PCFG representations back to dependency parses is tricky when a word occurs in a sentence more than once. However, we are only interested in the dependency 'skeleton' (directed but unlabeled structures), which makes matters easier. We will encode these dependency skeletons as in the gold standard provided in the file wsj10.txt.gold, in which the numbers tell for every position in the sentence what the position of its head is (ROOT has position 0). This gold standard was made by enriching a treebank with head annotations using the program treep (http://www.isi.edu/ chiang/software/treep/treep.html) and using these to extract the dependency structure.

Head-annotating your own parses

You can transform your CFG-parses into dependency skeletons by head-annotating them, and then using this to infer the dependency skeletons. In the Johnson's transform, the heads are easy to recognise and the parses can be annotated with a regular expression:

```
cat $Headden_parses                          # read in file
| sed 's/\((L[^ ]\+\) /\1-H /g'              # mark L_head as head
| sed 's/\([^ _]_L\) /\1-H /g'               # mark head_L as head
| sed 's/\((R[^_]_[^ ]\+\) /\1-H /g'         # mark R1_head and Rp_head as head
| sed 's/^(S (\([^ ]\+\) /(S (\1-H /g'       # mark Y_head in '(S (Y_head' as
| sed 's/\(_[lr]\)/\1-H/g'                   # mark terminals as head
> parses_head                                # write to parses_head
```

NB: This regular expression assumes that the non-terminals that are introduced for non-terminal X in Headden's encoding of DMV are 'Y_X', 'L_X', 'R_X', 'L1_X', 'R1_X', 'LP_X' and 'RP_X', and the terminals are X_l and X_r. If you used different (non)terminal symbols (for instance L-head instead of L_head), you will have to adapt the regular expression to get the desired result. Ask for help if you cannot figure out how.

After annotating your parses, you should check if every production of your parse is annotated with a head (check both left and right dependencies). An easy way of doing this is using the treeviewer that can be downloaded from Federico Sangati's website (`http://homepages.inf.ed.ac.uk/fsangati/Viewers/ConstTreeViewer_13_05_10.jar`). Use the option 'show heads' to see the head-annotation.

Transforming your head-annotated parses into dependency skeletons

You can use the python script transform_parses.py to transform your head-annotated parses into dependency skeletons:

```
python transform_parses.py $parses_head_annotation > dep_parses
```

If running this script results in an error, this is most probably caused by the fact that your parse is not fully annotated with heads. You should go back to the previous step and ensure yourself that the different type of productions all expand to a sequence of symbols of which exactly one is marked as head.

Evaluation

We evaluate our parses by checking how many words in each sentence have been assigned the right head (how many arrows are correctly placed). You can use the python script evaluateD.py to do this, you can run it as follows:

```
python evaluateD.py $input_file $gold_file
```

This will output the average accuracy per sentence.