Maman 14: Treebank Grammars for Hebrew Parsing

Reut Tsarfaty
www.tsarfaty.com

Submission due in May 27, 2018 23:55 (in individuals or teams of two)

Abstract

In this assignment we implement a PCFG-based statistical parser for Hebrew. While PCFGs work well for grammars in CNF form, the Hebrew treebank (like many other treebanks) does not yield a strictly CNF grammar. How can we use CYK to parse Hebrew sentences nonetheless? How well can we parse given the limited amount of Hebrew training data? In this work we aim to answer these questions.

Task Overview. In parsing we would like to receive a set of sentences as input and return the corresponding set of parse-trees as output. In Hebrew, we continue to assume that sentences are represented as sequences of morphological segments, so the yield of the parse-tree is in fact a sequence of morphemes.

To perform the Hebrew parsing task, you recieve a Java package that implements a very simple dummy parser.[1] The dummy parser is given a trainset and a goldset as input, and yields a set of flat parse-trees as output. The dummy parser works as follows: it reads-in the treebanks, it reads-off a treebank CFG based on the trees in the trainset, it takes every yield of a tree in the goldset as an unparsed sentence, and it assigns a flat dummy tree as a parse hypothesis to this sentence. It finally prints all outputs (trees, grammar, lexicon) into files. Along with the dummy parser you receive the code of the standard parse-evaluation software *evalb*.

Your task is to extend this into a working CYK parser which uses a treebank PCFG:

Q1. Install and evaluate the dummy parser (10pts)
Q2. Binarize the parse-trees and estimate rule probabilities (25pts)
Q3. Implement a version of CYK which allows for unary productions (25pts)
Q4. Parse, evaluate and analyze your Hebrew parser (20pts)
Q5. Based on error analysis, improve your parser (20pts)

Your grade for this assignment will be based on both the quality of your parser (correctness and completeness) and on the empirical quantitative results it obtains. In order to test your basic CYK algorithm, you may use any of the parsing examples we have executed in class and in the homework. For improving your parser, you may use any of the ideas we discussed in class or other ideas discussed in the literature.

[1] It is possible to complete the assignment in a different programming language, as long as you accept that you'd have to implement the given "dummy parser" yourself. The parser should compile/run on linux.

1 Baseline Architecture (10 points)

The pa2.zip archive from the moodle website contains the following directories:

- **data** // the train and gold portions of the Hebrew treebank
- **evalb** // the standard *evalb* evaluation program
- **exps** // sample output files of the experiments
- **src** // the archive *parser.jar* with source code of the dummy parser

The parser requires the following files, both of which are located under **data**:.

- *heb-ctrees.train*
- *heb-ctrees.gold*

Each tree in these files is written in a bracketed format (see lecture slides). For example:

```
(TOP (S (NP (CDT EFRWT) (NN ANFIM)) (VP (VB MGIEIM)) (PP (IN M) (NN TAILND)))
```

The **src** directory contains a jar file with the following java packages:[2]

- **bracketimport** (importer for bracketed trees)
- **decode** (a parsing algorithm)
- **grammar** (a set of rules)
- **parse** (the main method)
- **train** (a treebank-grammar extractor)
- **tree** (a constituency tree)
- **treebank** (a set of trees)
- **utils** (file management)

In the following sections you may use any of the provided classes, but you are not allowed to change the following packages: **bracketimport, utils**.
The **evalb** directory contains the standard distribution for parser evaluation (to be compiled on your machine).
The **exps** contains sample output files for the execution of the dummy parser.

Installation In order to install and run the parser, execute the following commands:

```
> unzip pa2.zip
> cd pa2/src
> unzip parser.jar
> cd ../
> java -cp src/ parse.Parse ./data/heb-ctrees.gold ./data/heb-ctrees.train exps/test
```

The entry point is the parse.Parser main() method, which does the following:

- *Read Input* // Reads in the trees input files into two Treebank objects, myGoldTreebank and myTrainTreebank respectively
- *Train* // Reads off the CFG rules from the myTrainTreebank into a Grammar object, and update the rule counts in the grammar
- *Parse* // Reads off the terminal sequence from the myGoldTreebank and attempt to parse it using the decode function. So far, the decode function implements a dummy baseline, which delivers a flat tree in which all words are tagged by NN.
- *Write Output* // Writes the parse trees, grammar files and lexicon into a file

[2] You are allowed to use a different programming language as long as you replicate this parser.

The parse command will generate three files under **exps**:

- *test.parsed* // the flat parse-trees
- *test.gram* // the grammar rules, with -lopProb(r) for each rule r
- *test.lex* // the lexicon file, with with -lopProb(r) for each emission rule r, group all emissions per word in one line.

These test files should have the same contents as those under exps/dummy-parser/ .
You can view the output trees using the viewtree script in the utils, by running the following from a unix command line:[3]

```
> wish ./src/utils/viewtree exps/test.parsed
```

In order to score the trees, you would need a standard evaluation software, which is found under the **evalb** directory.
To install this software, run the following commands:

```
> cd  evalb
> make
> cd ../
> ./evalb/evalb -p evalb/new.prm data/heb-ctrees.gold  exps/test.parsed > exps/test.eval
```

Go over the format of test.eval and make sure you understand it.[4]

Questions Evaluate the output of the dummy-parser and answer the questions:
(1) What is the parsing accuracy (precision, recall, f-score) when evaluating the dummy-parser output against the gold file for sentences of length up to 40 segments?
(2) What is the tagging accuracy when evaluating the dummy-parser output against the gold file for sentences of length up to 40 segments?

2 Training PCFGs (25pts)

(1) The *Train.train(Treebank)* procedure reads off a grammar from the treebank and keeps track of rule-counts. However, it does not provide rule probabilities. Provide a formula for estimating the rule probabilities using the rule counts provided in the Grammar object. Implement the probability estimation by extending the *train* code. Make sure your procedure updates the MinusLogProb value of each rule in the rule set with $-log(estimatedprob(A \rightarrow \alpha))$.
(2) As you have noticed, the extracted CFG is not in CNF, because it contains rules of arbitrary length. To overcome this, we will implement a transformation that binarizes trees prior to training. Suggest an algorithm that adds $n-2$ internal nodes for a production of length n.

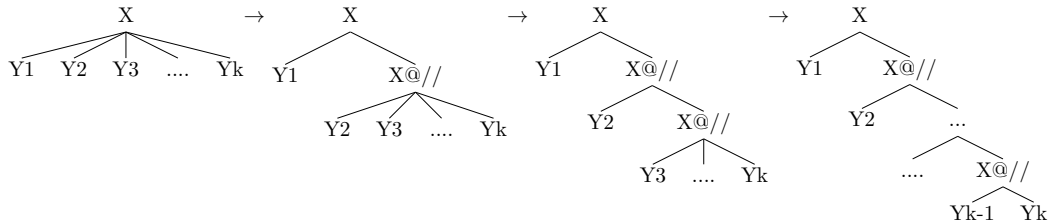
Describe and Motivate the algorithm, and provide the pseudocode. Implement this algorithm by extending train package.

[3] If you do not have wish installed on your system, there are additional other viewers available online (e.g., the ConstTreeViewer in `http://homepages.inf.ed.ac.uk/fsangati/`)

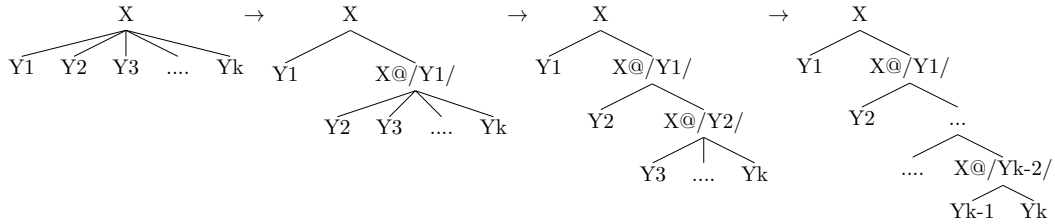
[4] You may alternatively use a java-based EvalC from the website of Federico Sangati `http://homepages.inf.ed.ac.uk/fsangati/`, if you find it more convenient.

(3) Using this way of binarizing trees means that some nodes are harder to estimate than others. Why is this so? We can fix this by employing a Markov assumption, specifying the number of sisters to remember in any added node.

- A 0-order model will look as follows:



- A 1-order model will look as follows:



- And so on

Klein and manning (2003) call this *horizontal markovization*. Add a parameter $h = int$ to the parser, where *int* can be -1,0,1,2,... an integer number. -1 refers to the default settings you have already implemented. $0, 1, 2, ...$ refer to the number of sisters you encode in the memory /Y1..Yn/ of every added node. What is the meaning of this transformation? What kind of generalization would it potentially add to your parser?

3 Decoding with PCFGs (25pts)

(1) Implement an extension of the CKY decoding algorithm for finding the best parse tree given a grammar, allowing for both binary and unary productions ($A \rightarrow B$ or $A \rightarrow BC$ where $A, B, C \in \mathcal{N}$). Your objective is to minimize the sum of minus-log-probability of the cell-items in the chart (this is in order to avoid number-underflow). Assign NN to any word segment which has been unseen in training. If your algorithm fails on a tree, provide the dummy-parser output instead. Implement this by extending the dummy-parse in the *decode* code. Provide the objective function in terms of -LogProb, provide the pseudocode, and describe the complexity (space as well as runtime) of the algorithm you provided.

4 Results and Analysis (20pts)

(1) Run your parser using the revised trainer and decoder and report the precision, recall, f-score, and tagging accuracy for sentences up to length 40, for $h = -1, 0, 1, 2$. **Make sure your code performs "undo" of any transformation, that is, remove any artificial node added for binarization, before comparing parse trees with the gold**.

(2) Take the first 10 sentences of your best experiment, and evaluate them manually against their gold counter-parts. That is, manually inspect the errors and suggest a classification of them to error types.
(3) In light of the error types you defined, discuss: what kind of knowledge is missing for your grammar in order to get more accurate parse trees?

5 Pushing Through (20pts)

(1) Make a modification of the training procedure so that your treebank grammar obtains better results. You may do it in any of the way(s) you discussed in class, or ones you learned about in the literature, or ones you would like to experiment with. Motivate your modifications.
(2) Run your parser with your improved grammar and report the precision, recall, f-score, and tagging accuracy for sentences up to length 40. **Make sure you undo any transformation before comparing with the gold trees**.
(3) Take the first 10 sentences in the best parsed file and evaluate them manually against their gold counter-parts. That is, inspect the errors and cluster them by types.
(4) Based on your error types, how would you suggest to keep improving your parser in next cycles?

6 Bonus (15pts)

The best parsing result in class will receive a 15 pts bonus. Your parser will be tested on a disjoint test-set, different than the gold set you used for development and evaluation.

Submission

What The submission archive should contain the following files and directories:.

- *data/* All data files, as provided
- *evalb/* All evalb installation, as provided
- *src/* Your extended code, documented, packed in a jar file.
- *exps/* All input, output and parameter files coming out of your experiments.
- *doc/* A typed report (addressing the above questions) in *.pdf format.

Your report should be brief and concise, and should not exceed four a4 pages.

How

- Submissions should be made electronically via moodle
- Submissions should be made on or before May 27th 2018, 23:55.
- Submissions should be made in pairs or individuals.

Good Luck!