ממן 13 | מבוא לעיבוד שפה טבעית 22933

PCFG Parsing for segmented Hebrew

In this assignment, you will be building one or more PCFG (Probabilistic Context Free Grammar) parsers! **Your input for this task** is a subset of a given constituency treebank, which you are given for train and evaluation.

Implementing these algorithms can be tedious, but, if you make sure to fully understand and plan each of their stages, you should arrive at a correct and performant implementation without special complications. The assignment additionally entails the analysis and clustering of errors that you observe in the predictions of your implemented parsers — be sufficiently sharp in your analysis!

Your final submission will be evaluated on a separate, held-out, portion of the same treebank, rather than via cross-validation; make sure not to violate the rules about the datasets, which are specified in the dataset spec section. We begin by presenting the treebank.

Understanding the Treebank

First, observe the treebank at hand. The trees included in this treebank are given in so-called “bracketed notation”, which is a sequential format for representing a tree:

```
(TOP (S (ADVP (RB RCUB)) (VP (VB MWXZR)) (PP (IN ALI) (NP (PRP ATM))) (NP (NP (H H) (NN
XWMR)) (SBAR (REL F) (S (VP (VB NFLX)) (PP (IN AL) (NP (PRP ANI)))))) (yyDOT yyDOT)))
```

In this format, nodes are delimited by spaces, and parentheses denote nesting levels. The nesting expresses the structure of the tree: in each parenthesized span, the first element denotes a tree node, and the rest of the elements are this node’s children. In this treebank, the root of the entire sentence is always written as `TOP`, and naturally, it is always the first element of the sequence.

It would be sometimes convenient, to unpack this sequence into an indented form, just to observe its structure more conveniently, as exemplified on the next page below. (You are not required to write code for this transformation, unless it somehow really helps you).

```
(TOP
  (S
    (ADVP
      (RB RCUB)
    )
    (VP
      (VB MWXZR)
    )
    (PP
      (IN ALI)
      (NP
        (PRP ATM)
      )
    )
    (NP
      (NP
        (H H)
        (NN XWMR)
      )
      (SBAR
        (REL F)
        (S
          (VP
            (VB NFLX)
          )
          (PP
            (IN AL)
            (NP
              (PRP ANI)
            )
          )
        )
      )
    )
    (yyDOT yyDOT)
  )
)
```

The bracketed-expression from the previous page, indented. Flipping this diagram on its side, the tree structure becomes visually more evident. You may also more vividly observe the same original expression as a tree, in various online tools such as this one here.

The tag ontology used in this treebank, i.e. a mapping between the tag names and what they mean, is the Penn TreeBank Tag Set, which is summarized here, and some further elucidation is given here and elsewhere.

Being a constituency parse tree, the leaves are all terminals (literals of the language). In our case they are more specifically *morphologically segmented* tokens of the Hebrew language, though not written in Hebrew letters. Rather, in this treebank, the literals are written in a **phonetic transliteration standard** which uses Roman letters rather than the Hebrew alphabet. **Transliteration** is the act of writing a word of one Language in the alphabet of another; in our case this side-steps the bidirectional display issues, that otherwise plague html rendering engines, editors and file viewers alike.

Task 1.1: Implementing and Evaluating CYK Parsers

Your task is to develop a CYK parser, training from a **Probabilistic Context-Free-Grammar (PCFG)** that you derive from the train set portion of the given treebank. The goal of your parser is to output a parse tree for any input sentence provided to it, such that your parser is correct as much as possible. In the following we walk you through the various elements entailed, and provide the specifications for this task.

Training

Being a generative model, for training you need to derive a Probabilistic Context-Free-Grammar from the given treebank, as discussed in class. Your parser should be trained from only the train-set. This derivation, which is considered the training part of the algorithm, entails:

Extracting and counting *parent* □ *child* relationships

Naturally, in order to be able to do so, one typically reads the sequential tree representations from file into a memory data structure; You may use your own tree node python classes, or use any of the python tree libraries that work with python 3.6.5 (we'd recommend your own compact class system as the default choice).

Pre-processing the rules into CYK

Recall that the CYK Algorithm can only use grammar rules given in *Chomsky Normal Form* (CNF), whereas treebanks do not typically come in Chomsky Normal Form! To this effect, in class we have presented two transformation techniques:

- *Binarization*, a technique for transforming *n-ary* production rules into binary ones.
- *Percolation,* a technique for transforming *unary* production rules (rules of the form $A \rightarrow B$, where both A and B are non-terminals) into binary ones.
- In both cases we end-up with a rule set comprised of binary rules only, with the exception of rules of the form $A \rightarrow a$ (where a is a terminal), which are allowed in Chomsky Normal Form and in CYK.

Estimating the rule probabilities

- As discussed in class*
- Can smoothing still be relevant?

* observe the probability constraints defined in class; you may also use them to verify your probability space

Decoding

Of course, given the training phase has been accomplished, decoding for a given input sentence would be accomplished via the CYK chart algorithm. The following should be noted —

Applying transforms as a pre-processing stage will enable safely getting through the CYK chart algorithm, but they do come with a price — the sentence parses that we will ultimately get at the end of the pipeline will be *different* that those in the treebank. That's because the treebank was created obliviously of these transforms — for a given sentence, the treebank will not contain non-terminal symbols we've introduced in our binarization transform. And likewise your percolation transforms were not engineered into the treebank. It is therefore necessary to unwind (de-transform) any transforms applied to the grammar, as a post-processing step after building the trees, in order to get outputs safely comparable to those in the test set.

Evaluation

Your implementation should output the most probable parse for a given input sentence.
Observe that for being compatible with the treebank itself, and fully usable for evaluation, the output parse should follow these obvious properties, by unwinding all levels of transformation your code has relied on —

- **Bracketed Notation**
  Obviously, you should encode every resulting tree back to the same bracketed notation used by the treebank

- **Your output parses should be fully compatible with the original grammar of the treebank.**
  As already mentioned, you should de-transform any transforms applied as preprocessing — the test set does not contain any "virtual nodes" you may have added in the transformations. Such nodes persisting in the output are typically not so useful for any consumer of your parser, and obfuscate linguistic properties when kept in the final output.

Once you have successfully implemented your parser, evaluate its performance on the provided test set, using the metrics *Precision*, *Recall*, and *F1*, as defined for ParseEval; refresh yourself in the class slides as for their mathematical definition in the context of parsing.

Unlike some other parsing methods, your parser does not obtain a Part-Of-Speech tagging for every input sentence as a separate step, but rather the Part-Of-Speech tagging emerges during the chart pass. Evaluate the Part-Of-Speech tagging that your parser obtains, using the standard token accuracy metric for POS tagging, which you have previously used in previous exercises.

Alleviating note: to save on round-trip time in your iterations, and following the current academic standard in this task, you can exclude any sentence whose length is greater than 40 segments, from your testing.

Scoring methodology

You are free to *either* use the supplied ParseEval evaluation utility* provided in directory `evalb` (with instructions for how to build it supplied there), or write your own python code for evaluation scoring of your trees against the test set trees. In case you opt for your own code, you are free and even encouraged to compare your evaluation outputs to those obtained with the given utility. Make sure to observe our definition of the evaluation metric here.

Task 1.2

It turns out that avoiding the percolation by adapting the CYK algorithm to *directly handle* unary productions, has its benefits. Submit a separate implementation that modifies the core algorithm to directly deal with unary productions, instead of percolating (but retain binarization). Here we do not provide the algorithm modification recipe to you. Similarly to previous bonus questions, you have to conceive it yourself, making sure it is efficient, and well-implemented. All else in this task is the same as in Task 1.1.

* you may also use the equivalent Java Class from the OpenNLP Project,
or Java class from Stanford CoreNLP from here instead, as long as you understand their outputs.

Task 2: Error Clustering Analysis

Analyze a subset (at least 25) of the sentences that have been not parsed correctly in your evaluation step —

Either manually, or via a clustering algorithm such as K-Nearest Neighbors, provide a sharp analysis of the types of errors. In doing so, make sure to cover the following desiderata —

- If manually — pick the sentences to analyze wisely (avoid too much selection bias)
- If automatically — consider how to come up with error types which are both explainable (interpretable by a human, and forming an easy to define group, each) and sufficiently prevalent relative to the amount of errors.

Provide a well-articulated account of the error types which you discern in the predicted outputs. Use tables, charts, or a simpler yet adequate quantitative account (whichever best meets the objective of being fully transparent about your error analysis, and any special points being made) regarding the types of error that you come up with, in your analysis.

You are free to submit this task relating to either Task 1.1 or to Task 1.2.

Task 3: Improving the Parser More!

The algorithm as you have implemented above, is known to possess various shortcomings leading to suboptimal performance (even after percolation has been avoided). You can do better! Improve the parser by carefully implementing **one (or more) of the improvement paths presented in class**.

Submit your improved parser, and a thorough account of how much error types have changed, duly analyzing against the analysis of the previous question.

Spec of Transliteration

The treebank uses this transliteration table in its terminals:

א	ב	ג	ד	ה	ו	ז	ח	ט	י	כ	ל	מ	נ	ס	ע	פ	צ	ק	ר	ש	ת
A	B	G	D	H	W	Z	X	J	I	K	L	M	N	S	E	P	C	Q	R	F	T

As you can see, Hebrew final letter forms (אותיות סופיות) are not separately included in the transliteration. Likewise, special punctuation and other symbols which aren't letters, have also been transliterated in this treebank —

%	“	,	:	(	“	.	-	)	!	?	;	...
O	U	yyCM	yyCLN	yyLRB	yyQUOT	yyDOT	yyDASH	yyRRB	yyEXCL	yyQM	yySCLN	yyELPS `(this character is actually called ellipsis)`

As these are somewhat gory details, we provide utility functions for the conversion back to normal Hebrew orthography (excluding the use of Hebrew final letter forms), in the accompanying code bundle.

The Dataset

In all tasks, you must adhere to the following role of each subset of the treebank:

Dataset Split	File	Allowed usage
Train	`data/heb-ctrees.train`	For training only
Eval	`data/heb-ctrees.gold`	For your evaluation
Held-out	`You never see it. And you are not allowed to obtain it over the Internet or otherwise`	We keep it. Your models will be evaluated against it.

What To Submit

task	zip file subdirectory	contents
1.1	1.1	Your runnable project for this task
1.2	1.2	Your runnable project for this task
2	2	PDF Document with/without external well-formatted charts/tables
3	3	• Your runnable project for this task* • Your Analysis (PDF once again, as above)

* only the your best accomplished parser from this task
** The code submission for each question should conform to the accompanying code Spec

Scoring Table for this Assignment

Definition of the ParseEval F1 Score for the sake of all tasks of this exercise:

We define the ParseEval F1 Score as the standard ParseEval F1 score where the tree root (the `Top` element), as well as the parts of speech, are not included at all in the Precision, Recall and F1 calculation.

The scoring table:

Task	Max Score Points		Score Criteria
1.1	20 pts	22	**ParseEval F1 Score**
		3	**POS Token Accuracy**
1.2	25 pts	22	**ParseEval F1 Score**
		3	**POS Token Accuracy**
2	25 pts		**Correctness and quality of analysis**
3	30 pts	22	**ParseEval F1 Score**
		3	**POS Token Accuracy**
		7	**Improvement paths chosen**
		8	**Correctness and quality of analysis**

Bear in mind that few points are added for terrifically self-explanatory, organized, and/or effectively-documented code which also works well, whereas more points are deducted for egregiously dirty and mis-organized code; Keep code submissions reasonably clean please.