Programming homework 2: Parsing with PCFGs (CKY algorithm)

New Attempt

Due Thursday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip
Available until Mar 6 at 11:59pm

Programming Homework 2: Parsing with Context Free Grammar (100 pts)

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Please make sure you are developing and running your code using Python 3.

Introduction

In this assignment you will implement the **CKY algorithm** for CFG and PCFG parsing. You will also practice retrieving parse trees from a parse chart and working with such tree data structures.

You can download the files for this project here: **hw2.zip (https://courseworks2.columbia.edu/courses/174039/files/16790661?wrap=1) (https://courseworks2.columbia.edu/courses/174039/files/16790661/download?download_frd=1)** (or in the Files section on Courseworks).

Python files:

To get you started, there are three Python files for this project.

1. cky.py will contain your parser and currently contains only scaffolding code.
2. grammar.py contains the class Pcfg which represents a PCFG grammar (explained below) read in from a grammar file.
3. evaluate_parser.py contains a script that evaluates your parser against a test set.

Data files:

You will work with an existing PCFG grammar and a small test corpus.
The main data for this project has been extracted from the ATIS (Air Travel Information Services) subsection of the Penn Treebank. ATIS is originally a spoken language corpus containing user queries about air travel. These queries have been transcribed

Course Chat

Treebank phrase structure syntax.
The data set contains sentences such as "*what is the price of*

There were 576 sentences in total, out of which 518 were usec
probabilites) and 58 for test. The data set is obviously tiny com
typically that would not be enough training data. However, bec
extracted grammar is actually able to generalize reasonably w

There are 2 data files:
atis3.pcfg - contains the PCFG grammar (980 rules)
atis3_test.ptb - contains the test corpus (58 sentence).

Take a look at these files and make sure you understand the fc
are a little different from what we have seen in class. Consider

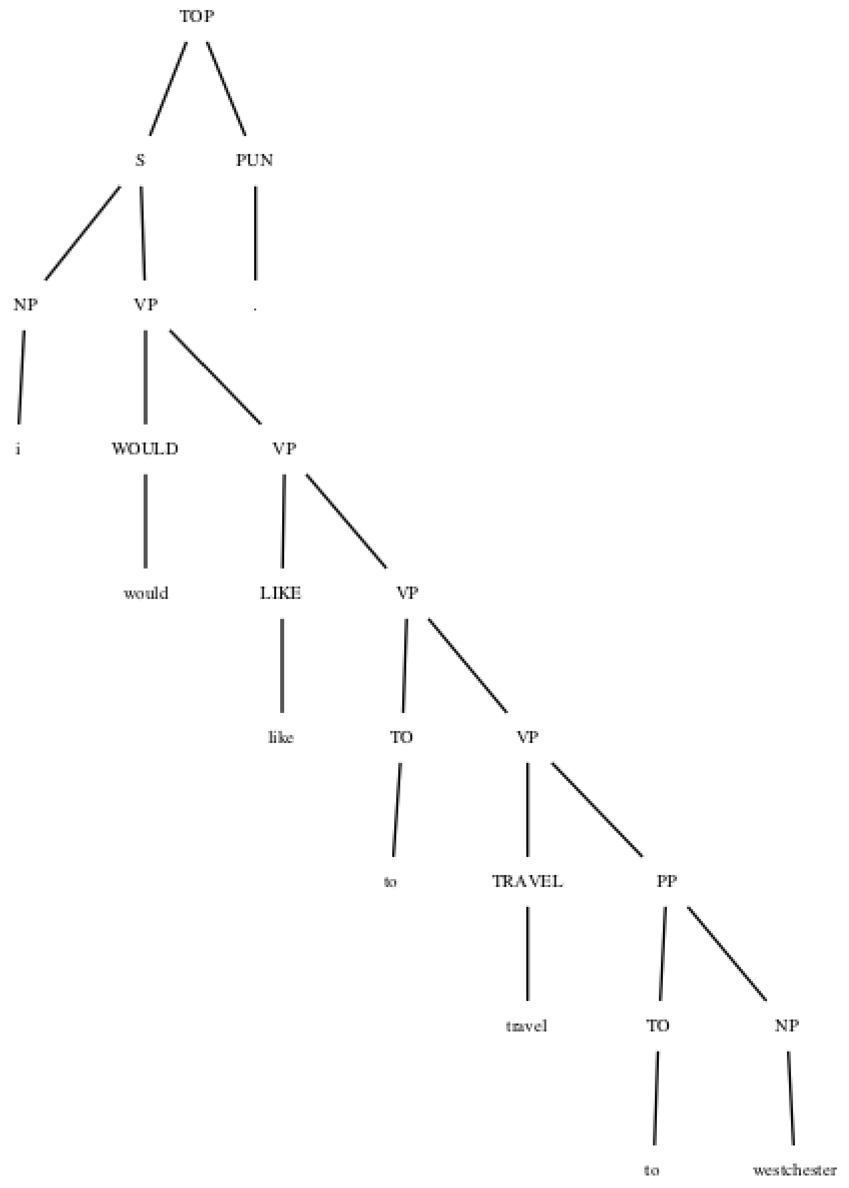
```
(TOP (S (NP i) (VP (WOULD would) (VP (LIKE like) (VP (TO to)
tchester))))))) (PUN .))
```

Today

RW Ruoxuan Wang 3/2/23
6:07 PM

1

Send



Note that there are no part of speech tags. In some cases phrases like NP directly project to the terminal symbol (NP -> westchester). In other cases, nonterminals for a specific word were added (TRAVEL -> travel).
The start-symbol for the grammar (and therefore the root for all trees) is "TOP". This is the result of an automatic conversion to make the tree structure compatible with the grammar in Chomsky Normal Form.

While you are working on your parser, you might find it helpful to additionally create a small toy grammar (for example, the one in the lecture slides) that you can try on some hand written test cases, so that you can verify by hand that the output is correct.

Part 1 - reading the grammar and getting started (20 pts)

Take a look at grammar.py. The class *Pcfg* represents a PCFG grammar in chomsky normal form. To instantiate a *Pcfg* object, you need to pass a file object to the constructor, which contains the data. For example:

```
with open('atis3.pcfg','r') as grammar_file:
    grammar = Pcfg(grammar_file)
```

You can then access the instance variables of the *Pcfg* instance to get information about the rules:

```
>>> grammar.startsymbol
'TOP'
```

The dictionary *lhs_to_rules* maps left-hand-side (lhs) symbols to lists of rules. For example, we will want to look up all rules of the form PP -> ??

```
>>> grammar.lhs_to_rules['PP']
[('PP', ('ABOUT', 'NP'), 0.00133511348465), ('PP', ('ADVP', 'PPBAR'), 0.00133511348465), ('PP', ('AF
TER', 'NP'), 0.0253671562083), ('PP', ('AROUND', 'NP'), 0.00667556742323), ('PP', ('AS', 'ADJP'), 0.
00133511348465), ... ]
```

Each rule in the list is represented as (lhs, rhs, probability) triple. So the first rule in the list would be

PP -> ABOUT NP with PCFG probability 0.00133511348465.

The *rhs_to_rules* dictionary contains the same rules as values, but indexed by right-hand-side. For example:

```
>>> grammar.rhs_to_rules[('ABOUT','NP')]
[('PP', ('ABOUT', 'NP'), 0.00133511348465)]

>>> grammar.rhs_to_rules[('NP','VP')]
[('NP', ('NP', 'VP'), 0.00602409638554), ('S', ('NP', 'VP'), 0.694915254237), ('SBAR', ('NP', 'VP'),
0.166666666667), ('SQBAR', ('NP', 'VP'), 0.289156626506)]
```

TODO:

- Write the method *verify_grammar*, that checks that the grammar is a valid PCFG in Chomsky Normal form. Specifically you need to verify that each rule corresponds to one of the formats permitted in CNF. To do this, you can assume that the lhs symbols of the rules make up the inventory of nonterminals. Any other symbol should be interpreted as a terminal.
  You also need to ensure all probabilities for the same lhs symbol sum to 1.0 (approximately). Because of numeric inaccuracies the sum may not be exactly 1.0. You may want to take a look at the *isclose* method in **Python's *math* package.** **(https://docs.python.org/3/library/math.html)**
- Then change the main section of grammar.py to read in the grammar, print out a confirmation if the grammar is a valid PCFG in CNF or print an error message if it is not. You should now be able to run grammar.py on grammars and verify that they are well formed for the CKY parser.

Part 2 - Membership checking with CKY (30 pts)

The file cky.py already contains a class *CkyParser*. When a *CkyParser* instance is created a *grammar* instance is passed to the constructor. The instance variable grammar can then be used to access this *Pcfg* object.

TODO: Write the method *is_in_language(self, tokens)* by implementing the CKY algorithm. Your method should read in a list of tokens and return *True* if the grammar can parse this sentence and *False* otherwise. For example:

```
>>> parser = CkyParser(grammar)
>>> toks =['flights', 'from', 'miami', 'to', 'cleveland', '.']  // Or: toks= 'flights from miami to
cleveland .'.split()
>>> parser.is_in_language(toks)
True
>>> toks =['miami', 'flights','cleveland', 'from', 'to','.']
>>> parser.is_in_language(toks)
False
```

While parsing, you will need to access the dictionary *self.grammar.rhs_to_rules*. You can use any data structure you want to represent the parse table (or read ahead to Part 3 of this assignment, where a specific data structure is prescribed).

The ATIS grammar actually overgenerates a lot, so many unintuitive sentences can be parsed. You might want to create a small test grammar and test cases. Also make sure that this method works for grammar with different start symbols.

Part 3 - Parsing with backpointers (30 pts)

The parsing method in part 2 can identify if a string is in the language of the grammar, but it does not produce a parse tree. It also does not take probabilities into account. You will now extend the parser so that it retrieves the most probable parse for the input sentence, given the PCFG probabilities in the grammar. The lecture slides on parsing with PCFG will be helpful for this step.

TODO: Write the method *parse_with_backpointers(self, tokens).* You should modify your CKY implementation from part 2, but use (and return) specific data structures. The method should take a list of tokens as input and returns a) the parse table b) a probability table. Both objects should be constructed during parsing. They replace whatever table data structure you used in part 2.

The two data structures are somewhat complex. They will make sense once you understand their purpose.

The first object is **parse table containing backpointers**, represented as a dictionary (this is more convenient in Python than a 2D array). The keys of the dictionary are spans, for example *table[(0,3)]* retrieves the entry for span 0 to 3 from the chart. The values of the dictionary should be dictionaries that map nonterminal symbols to backpointers. For example: *table[(0,3)]['NP']* returns the backpointers to the table entries that were used to create the NP phrase over the span 0 and 3. For example, the value of *table[(0,3)]['NP']* could be *(("NP",0,2),("FLIGHTS",2,3))*. This means that the parser has recognized an NP covering the span 0 to 3, consisting of another NP from 0 to 2 and FLIGHTS from 2 to 3. The split recorded in the table at table[(0,3)]['NP'] is the one that results in the ***most probable parse for the span [0,3] that is rooted in NP.***

Terminal symbols in the table could just be represented as strings. For example the table entry for table[(2,3)]["FLIGHTS"] should be "flights".

The **second object is similar, but records log probabilities** instead of backpointers. For example the value of *probs[(0,3)]['NP']* might be -12.1324. This value represents the log probability of the *best parse tree (according to the grammar)* for the span 0,3 that results in an NP.

Your *parse_with_backpointers(self, tokens)* method should be called like this:

```
>>> parser = CkyParser(grammar)
>>> toks =['flights', 'from', 'miami', 'to', 'cleveland', '.']
>>> table, probs = parser.parse_with_backpointers(toks)
```

During parsing, when you fill an entry on the backpointer parse table and iterate throught the possible splits for a (span/nonterminal) combination, that entry on the table will contain the back-pointers for the the current-best split you have found so far. For each new possible split, you need to check if that split would produce a higher log probability. If so, you update the entry in the backpointer table, as well as the entry in the probability table.

After parsing has finished, the table entry *table[0,len(toks)][grammar.startsymbol]* will contain the best backpointers for the left and right subtree under the root node. *probs[0,len(toks)][grammar.startsymbol]* will contain the total log-probability for the best parse.

cky.py contains two test functions *check_table_format(table*) and *check_prob_format(probs)* that you can use to make sure the two table data structures are formatted correctly. Both functions should return True. Note that passing this test does not guarantee that the content of the tables is correct, just that the data structures are probably formatted correctly.

```
>>> check_table_format(table)
True
>>> check_probs_format(probs)
True
```

Part 4 - Retrieving a parse tree (20 pts)

You now have a working parser, but in order to evaluate its performance we still need to reconstruct a parse tree from the backpointer table returned by parse_with_backpointers.

Write the function *get_tree(chart, i,j, nt)* which should return the parse-tree rooted in non-terminal nt and covering span i,j.

For example

```
>>> parser = CkyParser(grammar)
>>> toks =['flights', 'from', 'miami', 'to', 'cleveland', '.']
>>> table, probs = parser.parse_with_backpointers(toks)
>>> get_tree(table, 0, len(toks), grammar.startsymbol)
('TOP', ('NP', ('NP', 'flights'), ('NPBAR', ('PP', ('FROM', 'from'), ('NP', 'miami')), ('PP', ('TO', 
'to'), ('NP', 'cleveland')))), ('PUN', '.'))
```

Note that the intended format is the same as the data in the treebank. Each tree is represented as tuple where the first element is the parent node and the remaining elements are children. Each child is either a tree or a terminal string.

Hint: Recursively traverse the parse chart to assemble this tree.

Part 5 - Evaluating the Parser (0 pts, but necessary to know that your parser is working well)

The program evaluate_parser.py evaluates your parser by comparing the output trees to the trees in the test data set.

You can run the program like this:

```
python evaluate_parser.py atis3.pcfg atis3_test.ptb
```

Even though this program has been written for you, it might be useful to take a look at how it works.
The program imports your CKY parser class. It then reads in the test file line-by-line. For each test tree, it extracts the yield of the tree (i.e. the list of leaf nodes). It then feeds this list as input to your parser, obtains a parse chart, and then retrieves the predicted tree by calling your get_tree method.

It then compares the predicted tree against the target tree in the following way (this is a standard approach to evaluating constiuency parsers, called PARSEVAL):
First it obtains a set of the spans in each tree (including the nonterminal label). It then computes precision, recall, and F-score (which is the harmonic mean between precision and recall) between these two sets. The script finally reports the coverage (percentage of test sentences that had *any* parse), the average F-score for parsed sentences, and the average F-score for all sentences (including 0 f-scores for unparsed sentences).

My parser implementation produces the following result on the atis3 test corpus:

Coverage: 67%, Average F-score (parsed sentences): 0.95, Average F-score (all sentences): 0.64

What you need to submit

cky.py
evaluate_parser.py
grammar.py

Do not submit the data files.
Pack these files together in a zip or tgz file as described on top of this page. **Please follow the submission instructions precisely!**