CS187 Lab 3-4: Probabilistic parsing and parse disambiguation

November 3, 2024

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2024/lab3-4.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

Continuing our work on PCFG, you'll implement a probabilistic version of the CKY algorithm and apply it to a parse disambiguation task.

Preparations

[ ]:
```
import math
import nltk
import pandas as pd

from collections import Counter
from collections import defaultdict
from pprint import pprint
```

1 Prepositional phrase attachment ambiguity

In this lab, we'll set aside the rather limited world of arithmetic expressions, focusing on a common example of structural ambiguity in natural language called *prepositional phrase (PP) attachment*. A PP can modify both noun phrases and verb phrases, often creating ambiguity as to what constituent a PP should be attached to.

Here's a small grammar that includes PPs as noun phrase modifiers, as verb phrase modifiers, and as verb arguments.

[ ]:
```
probabilistic_grammar = nltk.PCFG.fromstring(
    """
    S -> NP VP [1.0]

    NP -> DT N [0.6]

    N -> N PP [0.2]

    VP -> TV NP [0.9] | DTV NP PP [0.1]

    PP -> P NP [1.0]

    DT ->  'a' [0.5] | 'the' [0.5]

    N -> 'books' [0.2] | 'gifts' [0.2]
    N -> 'table' [0.2] | 'book' [0.2]

    NP -> 'Taylor' [0.2] | 'Jordan' [0.2]

    DTV -> 'bought' [0.5] | 'put' [0.5]
    TV ->  'bought' [0.5] | 'saw' [0.5]

    P ->   'on' [0.3] | 'of' [0.4] | 'by' [0.1] | 'for' [0.2]
    """
```

```
)
```

> In this grammar, we've incorporated some verb subcategorization information. Here, `TV` stands for "transitive verb", a verb that takes a single NP argument, as in "Taylor bought it", and `DTV` stands for "ditransitive verb", which takes two arguments, an NP and a PP, as in "Taylor bought it for him".

Notice that the probabilities of all the rules with the same left-hand side sum to 1.

Consider the sentence

[ ]:
```
example1 = "Taylor bought a book for Jordan".split()
```

Take out a piece of paper and work with your lab partner to draw parse trees for this sentence according to this grammar. How many can you find?

To verify, we'll parse the sentence using a parser provided by NLTK. The `InsideChartParser` returns *all* parses of a sentence according to a probabilistic grammar, along with their probabilities. (It uses a more general algorithm than CKY, so doesn't require the grammar be in CNF.)

Parsing this sentence with the above PCFG results in two possible parses, displaying PP attachment ambiguity:

[ ]:
```
parser = nltk.parse.InsideChartParser(probabilistic_grammar)
possible_parses = list(parser.parse(example1))

for i, tree in enumerate(possible_parses):
  print(f'Possible parse #{i+1} with probability {tree.prob():.3g}:\n')
  tree.pretty_print()
```

Question: What is the more natural parsing, the one that leads to the preferred *reading* of the sentence, the reading that you understand the sentence as expressing? Is it the most probable parse tree?

Type your answer here, replacing this text.

Change some of the rule probabilities (try to change as few as possible) such that the other tree has higher probability.

[ ]:
```
# TODO - define `probabilistic_grammar_reweighted`.
probabilistic_grammar_reweighted = ...
```

[ ]:
```
parser2 = nltk.parse.InsideChartParser(probabilistic_grammar_reweighted)
possible_parses2 = list(parser2.parse(example1))

for i, tree in enumerate(possible_parses2):
  print(f'Possible parse #{i+1} with probability {tree.prob():.3g}:\n')
  tree.pretty_print()
```

Now we use the PCFG you defined to parse an only slightly different sentence.

[ ]:
```
example2 = "Taylor bought a book by Jordan".split()

possible_parses = list(parser2.parse(example2))

for i, tree in enumerate(possible_parses):
  print('Possible parse #{} with probability {:.3g}:\n'.format(i+1,tree.prob()))
  tree.pretty_print()
```

Question: Now what is the more natural parse for the sentence? Is it the most probable one? Can the PCFG be modified such that both sentences are parsed according to the natural readings for these sentences? Explain the problem.

Type your answer here, replacing this text.

2 Conversion to Chomsky Normal Form

The grammar (`probabilistic_grammar`) is almost in Chomsky Normal Form (which would make it suitable for the CKY algorithm). How many of its rules are not in CNF?

[ ]:
```
# TODO
non_cnf_rule_count = ...
```

[ ]:
```
grader.check("non_cnf_rule_count")
```

Convert the grammar `probabilistic_grammar` by hand to a CNF grammar. Try to make as few changes to the grammar as possible.

> **Important:** You should start with `probabilistic_grammar`, not `probabilistic_grammar_reweighted`.
>
> **Hint:** As part of your solution, you should make use of a new nonterminal called `NP_PP` that covers a noun phrase followed by a prepositional phrase, as can be found, for instance, in ditransitive verb phrases. (Later tests will depend on this.)

[ ]:
```
# TODO - convert `probabilistic_grammar` to CNF. You should make
#        use of a new nonterminal `NP_PP`.
probabilistic_grammar_cnf = nltk.PCFG.fromstring(
    ...
)
```

[ ]:
```
grader.check("cnf_conversion")
```

3 Probabilistic CKY

You've been availing yourself of a probabilistic parsing algorithm in the NLTK package. It's time to see how such algorithms work. In lab 3-2 you worked with the CKY algorithm as a recognizer and its extension to a parser using backpointers. In the following section you will familiarize yourself

with the probabilistic extension of the CKY parser, as presented by Jurafsky & Martin (Appendix C), which returns the most probable parse (MPP) of a string according to a PCFG grammar.

Now that we have a CNF grammar `probabilistic_grammar_cnf`, we can use the CKY algorithm to parse an example sentence. For reference, here is a pseudo-code version of the probabilistic CKY algorithm:

```
 1.  define cky-mpp(string = w1, ..., wN, grammar):
 2.      for j in [1..N]:                    # each end string position

             # handle rules of the form A -> w
 3.          for all A where A -> wj in grammar:
 4.              T[j-1, j, A] := Pr(A -> wj)

             # handle rules of the form A -> B C
 5.          for length in [2..j]:           # each subconstituent length
 6.              i := j - length             # start string position
 7.              for split in [i+1..j-1]     # each split point
 8.                  for all A where
 9.                          A -> B C in grammar
10.                          and T[i, split, B] > 0
11.                          and T[split, j, C] > 0:
12.                      new_prob := Pr(A -> B C)
13.                                  x table[i, split, B]
14.                                  x table[split, j, C]
15.                      if T[i, j, A] < new_prob
16.                         then T[i, j, A] := new_prob
17,                              back[i, j, A] := (split, B, C)
18.      return (build_tree(back[0, N, S]), T[0, N, S]
```

This PCKY algorithm is almost identical to the CKY variant you used in lab 3-2, with only a few differences, namely:

1. Table dimensions for a sentence of N words:

- CKY: $(N+1) \times (N+1)$
- PCKY: $(N+1) \times (N+1) \times |N|$

2. Table values:

- CKY: list of constituents
- PCKY: probabilities, where `table[i, j, A]` is the maximum probability of nonterminal `A` covering words between string positions `i` and `j`

3. Backpointers:

- CKY: mapping from nonterminals to set of all possible split positions and rules
- PCKY: mapping from nonterminals to the single most probable split position and rule

Notice that the probabilities of all the rules with the same left-hand side sum to 1.

We will implement the required $(N+1)\times(N+1)\times|N|$ three-dimensional tables as $(N+1)\times(N+1)$ two-dimensional tables, in which each cell will hold a dictionary mapping nonterminals to the required entry values. We will implement separate recognition and backpointer tables:

- For the recognition table `table`, the entry values are the probabilities.
- For the backpointers table `back`, the entry values are the appropriate backpointer `(split, B, C)`.

As in lab 3-2, all the cells that need not be filled contain '—'. All other cells are initialized with an appropriate default dictionary. Run the following code to initialize the tables. (You don't need to go over it, we will look at a specific cell to better understand the content.)

[ ]:
```
def make_tables(words):
    words = [""] + words
    N = len(words)

    # initialize data in tables
    table_data = [["---" for i in range(N)] for j in range(N)]
    back_data = [["---" for i in range(N)] for j in range(N)]

    # add in upper triangular elements
    for i in range(N):
        for j in range(N):
            if i < j:
                table_data[i][j] = defaultdict(float)
                back_data[i][j] = defaultdict(lambda x: None)

    # generate corresponding data frames
    table = pd.DataFrame(table_data, columns=words, index=range(N))
    table.columns = pd.MultiIndex.from_arrays([table.columns] + [range(N)])
    back = pd.DataFrame(back_data, columns=words, index=list(range(N)))
    back.columns = pd.MultiIndex.from_arrays([back.columns] + [range(N)])
    return (table, back)

table, back = make_tables(example1)
```

Let's print out one of the tables:

[ ]:
```
table
```

We'll "play parser" and fill both tables for the first four values of `j` following the pseudo-code above. Below, we'll finish filling the tables for the remaining two values of `j` yourself. Make sure you understand what's going on in the cell below before continuing.

[ ]:
```
                              # j = 1 (Taylor)
table.iloc[0,1]['NP'] = 0.2


                              # j = 2 (bought)
table.iloc[1,2]['DTV'] = 0.5
```

```
table.iloc[1,2]['TV'] = 0.5
                                # i = 0, split = 1: no changes

                              # j = 3 (a)
table.iloc[2,3]['DT'] = 0.5
                                # i = 1, split = 2: no changes
                                # i = 0, split = 1: no changes
                                # i = 0, split = 2: no changes

                              # j = 4 (book)
table.iloc[3,4]['N'] = 0.2
                                # i = 2, split = 3
table.iloc[2,4]['NP'] = 0.06
back.iloc[2,4]['NP'] = (3, 'DT', 'N')
                                # i = 1, split = 2
table.iloc[1,4]['VP'] = 0.027
back.iloc[1,4]['VP'] = (2, 'TV', 'NP')
                                # i = 1, split = 3: no changes
                                # i = 0, split = 1
table.iloc[0,4]['S'] = 0.0054
back.iloc[0,4]['S'] = (1, 'NP', 'VP')
                                # i = 0, split = 2: no changes
                                # i = 0, split = 3: no changes
```

We can see what progress has been made so far by examining the parser table.

[ ]:
```
table
```

Mercifully, we'll fill out the rest of the table for you.

[ ]:
```
                              # j = 5 (for)
table.iloc[4,5]['P'] = 0.2
                                # i = 3, split = 4: no changes
                                # i = 2, split = 3: no changes
                                # i = 2, split = 4: no changes
                                # i = 1, split = 2: no changes
                                # i = 1, split = 3: no changes
                                # i = 1, split = 4: no changes
                                # i = 0, split = 1: no changes
                                # i = 0, split = 2: no changes
                                # i = 0, split = 3: no changes
                                # i = 0, split = 4: no changes

                              # j = 6 (Jordan)
table.iloc[5,6]['NP'] = 0.2
                                # i = 4, split = 5
table.iloc[4,6]['PP'] = 0.04
back.iloc[4,6]['PP'] = (5, 'P', 'NP')
```

```
                                # i = 3, split = 4
table.iloc[3,6]['N'] = 0.0016
back.iloc[3,6]['N'] = (4, 'N', 'PP')
                                # i = 3, split = 5: no changes
                                # i = 2, split = 3
table.iloc[2,6]['NP'] = 0.00048
back.iloc[2,6]['NP'] = (3, 'DT', 'N')
                                # i = 2, split = 4
table.iloc[2,6]['NP_PP'] = 0.0024
back.iloc[2,6]['NP_PP'] = (4, 'NP', 'PP')
                                # i = 2, split = 5: no changes
                                # i = 1, split = 2
table.iloc[1,6]['VP'] = 0.000216
back.iloc[1,6]['VP'] = (2, 'TV', 'NP')
                                # i = 1, split = 3: no changes
                                # i = 1, split = 4: no changes
                                # i = 1, split = 5: no changes
                                # i = 0, split = 1
table.iloc[0,6]['S'] = 4.32e-05
back.iloc[0,6]['S'] = (1, 'NP', 'VP')
                                # i = 0, split = 2: no changes
                                # i = 0, split = 3: no changes
                                # i = 0, split = 4: no changes
                                # i = 0, split = 5: no changes
```

The table is now complete.

[ ]:
```
table
```

In addition to the table of probabilities, the algorithm also maintains the backpointer table, from which the most probable parse can be extracted. Let's look at it.

[ ]:
```
back
```

Reconstruct the parse tree that the algorithm finds by chasing backpointers in this table, and enter it using parenthesized string notation in the next cell.

> **Hint:** You'll definitely need pencil and paper for this. Start with the entry at 0,6 and work backwards from there.

[ ]:
```
# TODO
best_parse = nltk.Tree.fromstring(
    ...
)
```

[ ]:
```
grader.check("best_parse")
```

Let us look at the probability of this tree, as cached in the [0,6] entry in the probability table:

[ ]:
```
table.iloc[0,6]
```

Question: Compare the probability of this tree you just computed to the probabilities of the parses that the NLTK algorithm generated at the start of the lab, and explain the result.

Type your answer here, replacing this text.

Of course, it's nicer to have a function extract the parse from the backpointer table. This will also come in handy in Project 3. Write a function `extract_parse` that takes a list of words, a backpointers table, and a root nonterminal, and returns a tree covering the whole string (that is, between string positions 0 and the length of the string) rooted in the provided root nonterminal.

For instance, you should get the following behavior

```
>>> extract_parse(example1, back, 'S').pretty_print()
               S
   ____________|___
  |                VP
  |        ________|____
  |       |             NP
  |       |       ______|___
  |       |      |          N
  |       |      |    ______|___
  |       |      |   |          PP
  |       |      |   |       ___|____
  NP      TV     DT  N       P       NP
  |       |      |   |       |       |
Taylor bought   a  book     for     Jordan
```

> **Hint**: Extracting the parse using the backpointer table is bset visualized as a recursive process of extracting a parse tree rooted in a certain nonterminal `root` and covering the span between string positions `i` and `j`. Your code will be simplified by having it structured in that way as well.
>
> To do so, start by extracting the pertinent back table entry for `root`, `i`, and `j`. It should provide enough information to recursively extract two trees, one for nonterminal `B` between `i` and `split` and one for nonterminal `C` between `split` and `j`. These can be put together to construct the tree rooted in `root` between `i` and `j`. (You'll have to figure out what to do for the base case of this recursion, where the unary rules are used.)

[ ]:
```
def extract_parse(words, back, root):
    """Returns the parse for the sequence of `words` as specified by
    the `back`-pointers table for the portion of the input between
    string positions `i` and `j` parsed as nonterminal `root`
    """
    ## BEGIN SOLUTION NO PROMPT
    # compute sentence length
    N = len(words)
    # add a dummy element to words to simplify 1-indexing
```

```
    words = [''] + words

    def extract(root, i, j):
        """Returns a parse tree for the substring of `words` between
        string positions `i` and `j` rooted at the nonterminal `root`
        """
        entry = back.iloc[i, j]
        if not entry:
            # should only happen for unary rules
            assert i == j - 1
            return nltk.Tree(root, [words[j]])
        else:
            # binary rule; find split point and left and right nonterminals
            split, B, C = entry[root]
            left  = extract(B, i, split)
            right = extract(C, split, j)
            return nltk.Tree(root, [left, right])
    ## END SOLUTION NO PROMPT
    ...
```

[ ]:
```
grader.check("extract_parse")
```

4 Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there general additions or changes you think would make the lab better?

Type your answer here, replacing this text.

Question: What *specific single* change to the lab would have made your learning more efficient? This might be an addition of a concept that was not explained, or an example that would clarify a concept, or a problem that would have captured a concept in a better way, or anything else you can think of that would have made this a better lab.

Type your answer here, replacing this text.

End of Lab 3-4

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```