CS187 Lab 3-3: Probabilistic context-free grammars

October 5, 2023

[ ]:
```python
# 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/cs187-2021/lab3-3.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

In previous labs, you have practiced constituency parsing using context-free grammars with the CKY parsing algorithm. In this lab you will extend this framework to a probabilistic one, probabilistic context-free grammars (PCFG).

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `math.prod`
- `nltk.tree.Tree.productions`

Preparations

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

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

1 Syntactic ambiguity

Let's start with the following simplified grammar for arithmetic word expressions from the last lab:

[ ]:
```
arithmetic_grammar = nltk.CFG.fromstring("""
    S -> NUM | S OP S
    OP -> ADD | MULT

    NUM -> 'zero' | 'one' | 'two' | 'three' | 'four' | 'five'
    NUM -> 'six' | 'seven' | 'eight' | 'nine' | 'ten'

    ADD -> 'plus'
    MULT -> 'times'
""")
```

As a running example throughout this lab, we'll use the example phrase "two times three plus four".

[ ]:
```
example = "two plus three times four"
```

We can use the given CFG to parse this example phrase and print the possible parse trees.

[ ]:
```
parser = nltk.parse.BottomUpChartParser(arithmetic_grammar)
parses = list(parser.parse(example.split()))

for i, tree in enumerate(parses):
  print(f"Parse {i+1}:\n")
  tree.pretty_print()
```

Each parse tree represents a structured arithmetic expression (the *abstract syntax* of the concrete expression, for those of you with CS51 backgrounds). Manually calculate the value of the resulting equation for each of the parse trees.

```
[ ]: #TODO
     result_tree1 = ...
     result_tree2 = ...
```

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

We got two different parse trees for this simple expression. The occurrence of different structural interpretations of the same text is called *structural ambiguity* or *syntactic ambiguity*. Since natural language is oftentimes ambiguous, this is a very real concern.

In this particular case, the two syntactic structures corresponded to two different semantic values. As an exercise, try to construct an ambiguous expression (name it `pseudo_ambiguous`) such that all of its parse trees correspond to the same value, thereby demonstrating that not all structural ambiguity leads to semantic ambiguity.

```
[ ]: # TODO - construct an ambiguous expression such that all of its parse
     # trees correspond to the same value. `pseudo_ambiguous` should be
     # a string.
     pseudo_ambiguous = ...
```

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

One approach to dealing with the issue of syntactic ambiguity is by defining a scoring system to score the possible parses and choosing the highest scoring tree. We will see how this can be done by taking a probabilistic approach to CFG.

2 Probabilistic context-free grammars

To assign probabilities to strings, we will use a probabilistic context-free grammar (PCFG), a CFG in which each rule is augmented with a probability. A PCFG rule will be notated

$$A \rightarrow \beta\ [p]$$

where A is a nonterminal, β is a sequence of terminals and nonterminals, and p is a probability associated with the rule.

We'll write $\Pr(\beta \mid A)$ for the probability associated with the rule $A \rightarrow \beta$.

To constitute a valid probability distribution we require that for every nonterminal A

$$\sum_{A \rightarrow \beta \in \mathcal{P}} \Pr(\beta \mid A) = 1$$

where $\mathcal{P}$ is the set of CFG productions of the grammar. That is, the probabilities associated with all rules with the same left-hand side must sum to one.

Define `probabilistic_arithmetic_grammar` to be a probabilistic version of `arithmetic grammar` above, where the nonterminal probability distributions are **as uniform across the productions as possible**.

> You'll use the NLTK `nltk.PCFG.fromstring` function, which allows you to add the probabilities in brackets after each right-hand side, just as we've been doing above. For example, to notate `NUM -> 'zero'` as of probability 0.5, use `NUM -> 'zero' [0.5]`.

[ ]:
```
# TODO - define `probabilistic_arithmetic_grammar`. Round to
#        *3* significant figures if not divisible.
```

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

We can use the nltk.CFG.productions() method to get a list of the PCFG's productions:

[ ]:
```
probabilistic_arithmetic_grammar.productions()
```

Each of the productions in the list is an instance of the ProbabilisticProduction class. Each such instance is defined by three parameters: its left hand side (`lhs`), right-hand side (`rhs`), and rule probability (`prob`). These attributes can be accessed separately:

[ ]:
```
## Extract the second rule
pprod_example = probabilistic_arithmetic_grammar.productions()[1]

## Display its various components
print(f'For the production "{pprod_example}":\n'
      f'left hand side of the rule is {pprod_example.lhs()}\n'
      f'right hand side of the rule is {pprod_example.rhs()}\n'
      f'probability of the rule is {pprod_example.prob()}')
```

For non-probabilistic grammars, the class of productions is Production, which doesn't have a probability attribute and is only defined by its lhs and rhs attributes:

[ ]:
```
print(f'PCFG production: {probabilistic_arithmetic_grammar.productions()[1]} \n'
      f'        vs.\n'
      f'CFG production:  {arithmetic_grammar.productions()[1]}')
```

3 Parse tree probabilities

To use a PCFG to select among parse trees, we need to be able to calculate the probability of a parse tree as specified by the PCFG. We take the probability of a parse tree to be simply the product of the probabilities of each constituent in the tree, the probability of the rule associated with the constituent.

You'll use the PCFG `probabilistic_arithmetic_grammar` to calculate the probability of each of the parse trees in `parses`, the list of trees that were parsed from the `example` sentence.

To do that, you'll need to get all the productions used in a parse tree (using the productions method), find their probabilities, and multiply them together.

First, we will create a dictionary from the PCFG, so that we can easily access the rule probabilities. Write a function which accepts a PCFG and returns a dictionary whose keys are the CFG (not PCFG) productions and values are the associated probabilities.

> To construct a CFG production from a PCFG production, you can use `nltk.grammar.Production(production.lhs(), production.rhs())`.

[ ]:
```
#TODO - returns a dictionary whose keys are `nltk.grammar.Production` objects
#       and whose values are the associated probabilities
def pcfg_to_dict(pcfg):
  ...
```

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

We can use the function you wrote to convert `probabilistic_arithmetic_grammar` to a dictionary and inspect it to make sure it's working.

[ ]:
```
pprint(pcfg_to_dict(probabilistic_arithmetic_grammar))
```

Now for the payoff: Write a function that takes a parse tree and a PCFG and returns the probability of the parse tree according to the PCFG. The `pcfg_to_dict` function you just wrote is likely to come in handy.

> Note that we are asking for the probability (not the log probability). We **don't work in log space** in this lab for simplicity, but for parse trees of longer sentences (which you'll see in the project) you might have to work in the log space to avoid underflows.

[ ]:
```
# TODO: returns the probability of the parse tree.
# `tree.productions() might be useful for getting the
#  productions of a parse tree
def parse_probability(tree, pcfg):
    ...
```

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

We'll use it to calculate and print out the probability of each parse tree.

[ ]:
```
for i, tree in enumerate(parses):
    print(f'Probability of parse tree {i+1} is '
          f'{parse_probability(tree, probabilistic_arithmetic_grammar):1.2e}')
    tree.pretty_print()
```

Question: Which of the trees is the most probable parse? Explain why. If the two have the same probability, explain why that is the case instead, and describe how you might adjust the rule probabilities if possible so that they have different probabilities.

Type your answer here, replacing this text.

4 Lexicalizing the grammar

In order to allow parse probabilities to be more sensitive to contexts, it turns out to be useful to *lexicalize* the grammar – splitting (some of the) nonterminals based on what particular words they dominate. There are many techniques for performing this lexicalization. For this grammar, we'll split the `S` nonterminal based on the main operator that it dominates (if any). We'll thus have nonterminals `S_ADD`, `S_MULT`, and `S_NUM`. Thus, instead of a rule `S -> S OP S`, we'll have rules like:

```
S_ADD -> S_NUM ADD S_NUM
S_ADD -> S_NUM ADD S_ADD
S_ADD -> S_NUM ADD S_MULT
S_ADD -> S_ADD ADD S_NUM
```

and so forth. By splitting the nonterminals (and hence the productions) in this way, we can assign different probabilities to cases where, for instance, the primary operator on the left is a number, or addition, or multiplication.

Here is the lexicalized grammar:

[ ]:
```
lexicalized_arithmetic_grammar = nltk.CFG.fromstring(
    """
    S -> S_NUM | S_ADD | S_MULT

    S_NUM -> NUM

    S_ADD -> S_NUM ADD S_NUM
    S_ADD -> S_NUM ADD S_ADD
    S_ADD -> S_NUM ADD S_MULT
    S_ADD -> S_ADD ADD S_NUM
    S_ADD -> S_ADD ADD S_ADD
    S_ADD -> S_ADD ADD S_MULT
    S_ADD -> S_MULT ADD S_NUM
    S_ADD -> S_MULT ADD S_ADD
    S_ADD -> S_MULT ADD S_MULT

    S_MULT -> S_NUM MULT S_NUM
    S_MULT -> S_NUM MULT S_ADD
    S_MULT -> S_NUM MULT S_MULT
    S_MULT -> S_ADD MULT S_NUM
    S_MULT -> S_ADD MULT S_ADD
    S_MULT -> S_ADD MULT S_MULT
    S_MULT -> S_MULT MULT S_NUM
    S_MULT -> S_MULT MULT S_ADD
    S_MULT -> S_MULT MULT S_MULT

    NUM -> 'zero'   | 'one'    | 'two'
    NUM -> 'three'  | 'four'   | 'five'
    NUM -> 'six'    | 'seven'  | 'eight'
    NUM -> 'nine'   | 'ten'
```

```
    ADD -> 'plus'
    MULT -> 'times'
    """
)
```

Use this grammar to parse the example phrase ("two plus three times four") defined as `phrase` above.

```
[ ]: # TODO - parse `example` using the lexicalized grammar. `lexicalized_parses`
     #        should be a list of parses.
     lexicalized_parses = ...
```

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

Examine the trees, and make sure that you understand why they look the way they do. Notice that because of the lexicalization, the highest `S_` node corresponds to the highest operator in the parse – `S_MULT` when `MULT` is the highest operator and `S_ADD` when `ADD` is the highest operator.

```
[ ]: for i, tree in enumerate(lexicalized_parses):
       print(f"Possible parse {i+1}:\n")
       tree.pretty_print()
```

We can augment this grammar with probabilities as well.

Again, do so **making the probabilities for rules with the same left-hand side as uniform as possible**.

```
[ ]: # TODO - define `probabilistic_lexicalized_arithmetic_grammar`.
     #        Round to *3* significant figures if not divisible.
     probabilistic_lexicalized_arithmetic_grammar = nltk.PCFG.fromstring(
         ...
     )
```

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

Using this PCFG, we can calculate the probabilities associated with the two parses of the example phrase.

```
[ ]: for i, tree in enumerate(lexicalized_parses):
         print(f'Probability of parsed tree {i+1} is '
               f'{parse_probability(tree,␣
      ↪probabilistic_lexicalized_arithmetic_grammar):1.2e}')
         tree.pretty_print()
```

Make sure that you understand why the parse probabilities are the way they are. Why do they differ from the probabilities for the corresponding trees of the previous grammar? Why do the two trees still have the same probability? Call over a staff member for a quick check of your understanding.

5 Estimating rule probabilities from a corpus

In the previous section, you received a CFG augmented with rule probabilities that were arbitrarily stipulated. But where should rule probabilities come from? One way to generate rule probabilites is to learn them from a training corpus.

In this section you will use a toy corpus of sentences parsed according to the lexicalized grammar to generate maximum likelihood estimates of rule probabilities by counting the number of occurrences of a rule used in the corpus.

[ ]:
```
## The raw corpus, before splitting into separate phrases
corpus_raw = """
    # seven
    (S (S_NUM (NUM seven)))
    # one plus two
    (S (S_ADD (S_NUM (NUM one)) (ADD plus) (S_NUM (NUM two))))
    # two times three
    (S (S_MULT (S_NUM (NUM two)) (MULT times) (S_NUM (NUM three))))
    # two plus six times one
    (S (S_ADD (S_NUM (NUM two)) (ADD plus) (S_MULT (S_NUM (NUM six)) (MULT␣
 ↪times) (S_NUM (NUM one)))))
    # eight plus three plus seven
    (S (S_ADD (S_ADD (S_NUM (NUM eight)) (ADD plus) (S_NUM (NUM three))) (ADD␣
 ↪plus) (S_NUM (NUM seven))))
    # two plus three times four
    (S (S_ADD (S_NUM (NUM two)) (ADD plus) (S_MULT (S_NUM (NUM three)) (MULT␣
 ↪times) (S_NUM (NUM four)))))
    # eight times four times two
    (S (S_MULT (S_MULT (S_NUM (NUM eight)) (MULT times) (S_NUM (NUM four)))␣
 ↪(MULT times) (S_NUM (NUM two))))
    # five times two plus one
    (S (S_ADD (S_MULT (S_NUM (NUM five)) (MULT times) (S_NUM (NUM two))) (ADD␣
 ↪plus) (S_NUM (NUM one))))
    # five plus one times four
    (S (S_ADD (S_NUM (NUM five)) (ADD plus) (S_MULT (S_NUM (NUM one)) (MULT␣
 ↪times) (S_NUM (NUM four)))))
    # two times three plus four
    (S (S_ADD (S_MULT (S_NUM (NUM two)) (MULT times) (S_NUM (NUM three))) (ADD␣
 ↪plus) (S_NUM (NUM four))))
    # ten plus two times three
    (S (S_ADD (S_NUM (NUM ten)) (ADD plus) (S_MULT (S_NUM (NUM two)) (MULT␣
 ↪times) (S_NUM (NUM three)))))
    # four times three plus two times one
    (S (S_ADD (S_MULT (S_NUM (NUM four)) (MULT times) (S_NUM (NUM three))) (ADD␣
 ↪plus) (S_MULT (S_NUM (NUM two)) (MULT times) (S_NUM (NUM one)))))
    # four plus three times two plus one
```

```
    (S (S_ADD (S_ADD (S_NUM (NUM four)) (ADD plus) (S_MULT (S_NUM (NUM three))␣
 ↪(MULT times) (S_NUM (NUM two)))) (ADD plus) (S_NUM (NUM one))))
"""

def corpus_from_string(raw):
  """Return a corpus as a list of sentences.

  The `raw` corpus is split at newlines, trimmed of whitespace,
  and comment lines and blank lines are eliminated.
  """
  return list(filter(lambda x: x != '' and x[0] != '#',
                     map(lambda sent: sent.strip(),
                         raw.split('\n'))))

## The processed corpus we'll use
corpus = corpus_from_string(corpus_raw)
```

Recall that for the rule probabilities to define a valid probability distibution, the following needs to hold

$$\sum_{A \to \beta \in G} \Pr(\beta \mid A) = 1$$

where G is the set of productions.

In order to get an estimate for each production probability, we can count the number of occurrences of the production, normalizing by the number of occurrences of all productions with the same left-hand side.

$$\begin{aligned}
\Pr(\beta \mid A) &= \frac{\sharp(A \to \beta)}{\sum_{\beta'} \sharp(A \to \beta')} && (1)\\
&= \frac{\sharp(A \to \beta)}{\sharp(A)} && (2)
\end{aligned}$$

We will define three functions:

1. `rule_counter` - Accepts a list of sentences and returns a dictionary of rule counts (where the key is the NLTK CFG production (defined by the lhs and rhs) and the value is the number of rule occurrences).
2. `lhs_counter` - Accepts a list of sentences and returns a dictionary of lhs counts (where the key is the lhs nonterminal and the value is the count of that nonterminal's occurences as a lhs).
3. `train_pcfg` - Accepts a CFG and a list of sentences and returns a PCFG with probabilities based on the training corpus; assumes that the parses in the corpus are consistent with the CFG argument.

Implement these functions as specified above.

Hint: The following NLTK functions may be useful:

- nltk.Tree.fromstring
- nltk.grammar.PCFG
- nltk.grammar.productions
- nltk.grammar.ProbabilisticProduction

[ ]:
```
#TODO
def rule_counter(sentence_list):
  ...

#TODO
def lhs_counter(sentence_list):
  ...

#TODO
def train_pcfg(cfg, sentence_list):
  ...
```

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

Now we can use the `train_pcfg` function that you wrote to build a PCFG version of the `lexicalized_arithmetic_grammar`, with rule probabilities derived from the corpus.

[ ]:
```
trained_probabilistic_lexicalized_arithmetic_grammar =␣
 ↪train_pcfg(lexicalized_arithmetic_grammar, corpus)
print(trained_probabilistic_lexicalized_arithmetic_grammar)
```

Observe that the probabilities of the two rules `S_ADD -> S_NUM ADD S_MULT` and `S_MULT -> S_ADD MULT S_NUM` are now different from each other. (They were both the same in the previous grammar, since you made the probabilities as uniform as possible.)

We'll use NLTK's implementation of the probabilistic CKY algorithm (`nltk.ViterbiParser`) to generate the best parse for some strings according to this induced PCFG. (You'll implement this yourself in lab 3-4.)

[ ]:
```
trained_parser = nltk.
 ↪ViterbiParser(trained_probabilistic_lexicalized_arithmetic_grammar)
```

Use this parser to parse the `example` phrase "two plus three times four" from above. Which parse does it return? Do you understand why?

> Be careful. The parser returns a Python generator of the parses, not a list. You can't use the generator twice, so you should save the `induced_grammar_parses` as a list constructed from the generator object to pass all of the tests.

[ ]:
```
# TODO - parse `example` using `induced_parser`
trained_grammar_parses = ...
```

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

```
[ ]: for i, tree in enumerate(trained_grammar_parses):
         print(f'Probability of parse tree {i+1} is '
               f'{parse_probability(tree,␣
      ↪trained_probabilistic_lexicalized_arithmetic_grammar):1.2e}')
         tree.pretty_print()
```

Now consider a new example:

```
[ ]: example2 = "three plus nine plus two"
```

How many parses there are for this new expression “three plus nine plus two” according to the induced PCFG? Set the variable in the next cell accordingly.

```
[ ]: # TODO
     example2_parse_count = ...
```

Let’s examine the most probable parse for this example.

```
[ ]: for i, tree in enumerate(trained_parser.parse(example2.split())):
         print(f'Probability of parsed tree {i+1} is '
               f'{tree.prob():1.2e}')
         tree.pretty_print()
```

In trying to parse this example, you undoubtedly obtained a single parse with zero probability.

That doesn’t seem right. A particular construction not occurring in the training set doesn’t warrant assigning zero probability to any and all examples that use that construction. No matter how large a training set, there can always be unattested constructions. In the case at hand, one of the needed productions (`NUM -> nine`) has zero probability, which in turn followed from the fact that “nine” occured nowhere in the training corpus.

Question: With a *single word*, what technique that you’ve learned would be appropriate to solve this problem.

Type your answer here, replacing this text.

Question: The example that we provided of an ambiguity in arithmetic expressions is admittedly quite artificial. Can you think of other (more natural) examples, in natural language or elsewhere, where this phenomenon might occur?

Type your answer here, replacing this text.

6 Lab debrief – for consensus submission only

Question: We’re interested in any thoughts your group has 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:

- 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 additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of Lab 3-3

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

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