Lab 4-2: Semantic interpretation with SQL

June 26, 2023

[121]:
```
# 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/cs236299-2023-spring/lab4-2.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[122]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

1 Course 236299

1.1 Lab 4-2 - Semantic interpretation with SQL

In the previous lab, you built a syntactic-semantic grammar for semantic interpretation of natural language into First-Order Logic (FOL) expressions.

In this lab, you'll use a similar approach for a standard NLP task, natural-language database query. You'll use the compositional semantics methods of the previous lab to convert natural-language queries into SQL queries, which can be actually executed against a SQL database. Familiarity with this task will be useful in the fourth project segment, where you'll be building systems for natural-language queries against the ATIS flight information database.

2 Preparation

[123]:
```
import os
import pprint
import sys
import wget

import nltk
import sqlite3
```

[124]:
```
# Download code for augmented grammars
remote_script_dir = "https://raw.githubusercontent.com/nlp-236299/data/master/
  ↪scripts/"
local_script_dir = "./scripts/"

# Create and search the local script directory
os.makedirs(local_script_dir, exist_ok=True)
sys.path.insert(1, local_script_dir)

# Download files to script directory
wget.download(remote_script_dir + "trees/transform.py", out=local_script_dir)

# Import functions for transforming augmented grammars
import transform as xform
```

Consider the phrase "flights from Boston to New York". The representation of the property denoted by this phrase might be (as per last lab),

$$\lambda x.Flight(x) \wedge Origin(x, Boston) \wedge Destination(x, NewYork)$$

If instead we had a SQL database with a `flight` relation with fields `flight_id`, `origin`, and `destination`, we might translate this phrase into the following query

```
SELECT flight_id from flight WHERE origin == "Boston" and destination == "NewYork"
```

which returns the flight IDs of all the “flights from Boston to New York”.

Then, we will be able to run the generated SQL query on a database of flights, to actually answer the query. This process can be described as:

NL query   SQL query   response

We will focus on the first transformation (NL question   SQL query) using a syntactic-semantic grammar. The second transformation (SQL query   response) will be executed automatically by the database.

3 Establishing the SQL database

First, we will initialize the SQL dataset. We will populate the dataset similarly to the flight world of the previous lab.

We initialize the same constants as in the previous lab:

[125]:
```
# Constants
Boston = "Boston"
NewYork = "New York"
TelAviv = "Tel Aviv"
DL10 = "DL10"
DL11 = "DL11"
DL13 = "DL13"
LY01 = "LY01"
LY12 = "LY12"
Morning = "Morning"
Evening = "Evening"
```

We populate the SQL dataset using a single table called **Flights**:

Flights:

flightid	origin	destination	departureTime	arrivalTime
DL10	Boston	NewYork	Morning	Evening
DL11	Boston	TelAviv	Evening	Morning
DL13	NewYork	Boston	Evening	Evening
LY01	TelAviv	NewYork	Evening	Morning
LY12	NewYork	TelAviv	Morning	Evening

To reset the database when working on the lab, you can re-run the cell below.

[126]:
```
def establish_database():
    conn = sqlite3.connect(":memory:")
    c = conn.cursor()

    c.execute(
        "CREATE TABLE Flights (flightid TEXT, origin TEXT, destination TEXT,␣
 ↪departureTime TEXT, arrivalTime TEXT)"
    )
    c.executemany(
        "INSERT INTO Flights VALUES (?, ?, ?, ?, ?)",
        [
            (DL10, Boston, NewYork, Morning, Evening),
            (DL11, Boston, TelAviv, Evening, Morning),
            (DL13, NewYork, Boston, Evening, Evening),
            (LY01, TelAviv, NewYork, Evening, Morning),
        ],
    )
    return c

c = establish_database()
```

Let's query the table, to verify that it contains the proper rows:

[127]:
```
print('Flights:')
res = c.execute('SELECT * FROM Flights')
for row in res:
  print(row)
```

```
Flights:
('DL10', 'Boston', 'New York', 'Morning', 'Evening')
('DL11', 'Boston', 'Tel Aviv', 'Evening', 'Morning')
('DL13', 'New York', 'Boston', 'Evening', 'Evening')
('LY01', 'Tel Aviv', 'New York', 'Evening', 'Morning')
```

Complete the initialization of the database (using `c.execute('INSERT INTO...')`), for the last flight:

flightid	origin	destination	departureTime	arrivalTime
LY12	NewYork	TelAviv	Morning	Evening

Hint: Function `establish_database` provides an example of how to insert into the database.

[128]:
```
#TODO - Insert the final flight into the table
#             Note that you should either use the string "New York" (with␣
 ↪space),
#             or the Python object `NewYork` (without space), but don't use␣
 ↪"NewYork".
c.executemany('INSERT INTO Flights VALUES (?,?,?,?,?)', [(LY12,␣
 ↪NewYork,        TelAviv,        Morning,        Evening)])
```

[128]:
```
<sqlite3.Cursor at 0x7ffb53405340>
```

[129]:
```
grader.check("add_flight")
```

[129]:
```
All tests passed!
```

We can test that the row was properly added:

[130]:
```
res = c.execute('SELECT * FROM Flights')
for row in res:
  print(row)
```

```
('DL10', 'Boston', 'New York', 'Morning', 'Evening')
('DL11', 'Boston', 'Tel Aviv', 'Evening', 'Morning')
('DL13', 'New York', 'Boston', 'Evening', 'Evening')
('LY01', 'Tel Aviv', 'New York', 'Evening', 'Morning')
('LY12', 'New York', 'Tel Aviv', 'Morning', 'Evening')
```

In the previous lab, you created a syntactic-semantic grammar that used the lambda calculus to build FOL expressions (as Python objects) to represent the meanings of queries. In this lab, you'll use the same syntactic productions, but instead, map constituents to functions that build SQL queries.

Complete the following grammar. The first rule is provided as an example.

Hints:

1. Recall that the semantic composition functions are functions from right-hand side meanings (there might be zero or more) to the meaning of the whole.
2. The general structure of SQL queries for this grammar will be:

   `SELECT DISTINCT flightid from Flights WHERE ...`

   For consistency, the queries will always have a `WHERE` clause. If there are no conditions required in the body of the `WHERE` clause, you can just use `1` as the `WHERE` body. (The `1` value is SQL's proxy for the Boolean value `TRUE`.) Again for consistency, you might want always to have a `1` as the final condition, e.g.,

   `SELECT DISTINCT flightid from Flights WHERE origin == "New York" AND arrivalTime == "Evening" AND 1`

3. Use semantic types consistently. For instance, you might use the following typings for meanings:

Syntactic Type	Semantic Type	Example
Q	str (a full query)	`'SELECT DISTINCT flightid FROM Flights WHERE 1'`
NP, PP, PP_PLACE, PP_TIME	str (a `WHERE` body) -> str (a `WHERE` body)	`lambda P: f'origin == NewYork AND {P}'`
LOC	str (a place)	`'NewYork'`
TIME	str (a time)	`'Morning'`

By making NP and PP meanings functions from `WHERE` bodies to `WHERE` bodies, it's easy to have compound conditions like `origin == "New York" AND arrivalTime == "Evening"`.

[131]:
```
#TODO - Add augmentations to the grammar to generate SQL queries.
# We gave you a few to get started.

grammar_spec = """
    Q -> NP                          : lambda NP: NP("1")
    NP -> 'flights'                  : lambda: lambda P: f"SELECT DISTINCT␣
 ↪flightid from Flights WHERE {P}"
    NP -> NP PP                      : lambda NP, PP: lambda P: f"{NP(PP)} AND␣
 ↪{P}"

    PP -> PP_PLACE                   : lambda x: x
    PP -> PP_TIME                    : lambda x: x

    PP_PLACE -> 'from' LOC           : lambda loc: f'origin == "{loc}"'
              | 'leaving' LOC        : lambda loc: f'origin == "{loc}"'
              | 'to' LOC             : lambda loc: f'destination == "{loc}"'
              | 'arriving' 'at' LOC  : lambda loc: f'destination == "{loc}"'

    PP_TIME -> 'arriving' TIME       : lambda time: f'arrivalTime == "{time}"'
             | 'departing' TIME      : lambda time: f'departureTime == "{time}"'
             | 'leaving' TIME        : lambda time: f'departureTime == "{time}"'

    LOC -> 'Boston'                  : lambda: Boston
    LOC -> 'New' 'York'              : lambda: NewYork
    LOC -> 'Tel' 'Aviv'              : lambda: TelAviv

    TIME -> 'in' 'the' 'morning'     : lambda: Morning
    TIME -> 'in' 'the' 'evening'     : lambda: Evening
"""
```

[132]:
```
grammar, augmentations = xform.parse_augmented_grammar(grammar_spec,␣
 ↪globals=globals())
```

To test the grammar, we can parse a sample query:

[133]:
```
parser = nltk.parse.BottomUpChartParser(grammar)

for parse in parser.parse('flights from Boston leaving in the morning'.split()):
  parse.pretty_print()
```

```
                                   Q
                                   |
                                   NP
          _____________________|___________
        NP                                 |
    _____|______                            |
   |            PP                          PP
   |            |                           |
   |         PP_PLACE                    PP_TIME
   |      ______|_______         ___________|_____
   NP    |             LOC      |               TIME
   |     |              |       |      ___________|______
flights from          Boston leaving  in         the  morning
```

3.1 Semantically interpreting syntactic trees

With parse tree in hand, and the dictionary of semantic augmentations for each syntactic rule, we can recursively traverse the tree and compute its meaning.

Write a function `interpret`, which takes a parse tree and a dictionary of semantic augmentations indexed by syntactic production (as returned by `xform.parse_augmented_grammar`) and returns the meaning for the tree.

The function will be naturally recursive, since to compute the meaning of the tree you'll need to apply the appropriate semantic composition function to the meanings of the subtrees (that's the recursive bit).

Hints: 1. To iterate over a tree's child subtrees, you can simply iterate as if it were a list: `[child for child in tree]`. Note that `child` can be either a string (for terminals), or an `nltk.Tree` object storing the subtree (for nonterminals).

2. To get the syntactic rule at the root of the tree, you can use `tree.productions()[0]`

3. You'll want to know about Python's `*` operator. If you want to apply a function that takes multiple arguments and you have a list of its arguments, you can "unpack" the list using the `*` operator. For example, the following code is valid and works as expected:

   ```
   def f(a, b, c):
     return a + b + c
   my_arguments = [1,2,3]
   f(*my_arguments)
   ```

   This might be useful when calling a semantic composition function, since how many arguments it takes is only known at runtime.

4. If you want to check whether the root of a subtree `t` is a nonterminal or a terminal, you can use `if isinstance(t, nltk.Tree)`. This will return `True` for nonterminals, and `False` for terminals (because terminals are just strings in NLTK).

5. The solution is only a few lines of code.

[134]:
```
# TODO - write the `interpret` function
def interpret(tree, augmentations):
    """Returns a string containing an SQL query for the parse `tree`
    as interpreted by the grammar `augmentations`.
    """
```

```
    temp = []
    for child in tree:
      if isinstance(child, nltk.Tree):
        temp.append(interpret(child, augmentations))
    result = augmentations[tree.productions()[0]](*temp)
    return result
```

[135]:
```
grader.check("interpreting")
```

[135]:

```
All tests passed!
```

4 Putting it all together

Now we can put everything together to translate an NL query to SQL and execute the query against the database.

[136]:
```
def query_to_sql(nl_query, parser, augmentations):
    """Parses a natural language query `nl_query`, interprets it as SQL, and
    executes and returns the result of the query.
    """
    sentence = nl_query.split()
    parses = [p for p in parser.parse(sentence)]
    for tree in parses:
        tree.pretty_print()
    return [interpret(parse, augmentations) for parse in parses]


def query_to_answer(nl_query, parser, augmentations):
    """Parses and interprets a natural language query `nl_query` to a SQL query
    as per the provided `parser` and `augmentations` and executes the SQL
    query on the database, printing some useful information and returning
    the query results."""
    sql_queries = query_to_sql(nl_query, parser, augmentations)
    for query in sql_queries:
        print(f"SQL query: {query}")
        print("Result:")
        res = list(c.execute(query))
        for row in res:
            print(row)
    return res
```

You can now test your parser by running some queries all of the way through. The expected SQL query for `flights from Boston` is: `SELECT DISTINCT flightid FROM Flights WHERE origin == "Boston" AND 1` (or some such).

Note the quotation marks around `Boston`. These are required so that SQL interprets it as a field value rather than a field name.

[137]:
```
res1 = query_to_answer('flights from Boston', parser, augmentations)
```

```
         Q
         |
         NP
   _____|______
  |            PP
  |            |
  |         PP_PLACE
  |      ______|_______
  NP    |              LOC
  |     |               |
flights from          Boston

SQL query: SELECT DISTINCT flightid from Flights WHERE origin == "Boston" AND 1
Result:
('DL10',)
('DL11',)
```

[138]:
```
res2 = query_to_answer('flights from Boston to New York', parser, augmentations)
```

```
                        Q
                        |
                        NP
         ______________|___________
         NP                        |
   _____|______                    |
  |            PP                  PP
  |            |                   |
  |         PP_PLACE            PP_PLACE
  |      ______|_______       _____|______
  NP    |              LOC    |          LOC
  |     |               |     |      ______|___
flights from          Boston  to    New        York

SQL query: SELECT DISTINCT flightid from Flights WHERE origin == "Boston" AND
destination == "New York" AND 1
Result:
('DL10',)
```

[139]:
```
res3 = query_to_answer('flights from New York arriving in the evening', parser,␣
↪augmentations)
```

```
                               Q
                               |
                               NP
               ________________|___________________
              NP                                   |
    __________|_____                               |
   |                PP                             PP
   |                |                              |
   |             PP_PLACE                       PP_TIME
   |      __________|______            ____________|_____
   NP    |                LOC         |                  TIME
   |     |           ______|___       |        ___________|______
flights from       New        York arriving   in         the  evening

SQL query: SELECT DISTINCT flightid from Flights WHERE origin == "New York" AND
arrivalTime == "Evening" AND 1
Result:
('DL13',)
('LY12',)
```

```
[140]: res4 = query_to_answer('flights from New York to Tel Aviv departing in the␣
        ↪morning arriving in the evening', parser, augmentations)
```

```
                                                              Q
                                                              |
                                                              NP
__________________|______________________________________________
                                          NP
|
                                __________|_______________________________
|
                               NP                                          |
|
               ________________|__________                                 |
|
              NP                          |                                |
|
    __________|_____                      |                                |
|
   |                PP                    PP                               PP
PP
   |                |                     |                                |
|
   |             PP_PLACE              PP_PLACE                         PP_TIME
PP_TIME
   |      __________|______          _____|______              ____________|_____
____________|_____
   NP    |                LOC       |           LOC           |
TIME            |                 TIME
   |     |           ______|___     |      ______|___         |
___________|______        |       ___________|______
flights from       New        York  to   Tel        Aviv departing  in
the  morning arriving  in         the  evening

SQL query: SELECT DISTINCT flightid from Flights WHERE origin == "New York" AND
destination == "Tel Aviv" AND departureTime == "Morning" AND arrivalTime ==
"Evening" AND 1
Result:
('LY12',)
```

```
[141]: res5 = query_to_answer('flights from Tel Aviv arriving at New York leaving in␣
        ↪the evening arriving in the morning', parser, augmentations)
```

```
                                                              Q
                                                              |
                                                              NP
______________|____________________________________________________
                                               NP
|
_______________|_________________________________
|
                               NP
|                                 |
               ________________|_______________
|                                 |
              NP                               |
|                                 |
    __________|_____                           |
|                                 |
   |                PP                         PP
PP                                PP
   |                |                          |
|                                 |
   |             PP_PLACE                   PP_PLACE
PP_TIME                           PP_TIME
   |      __________|______            ________|__________
___________|_____                ____________|_____
   NP    |                LOC         |        |         LOC         |
TIME            |                 TIME
   |     |           ______|___       |        |       ___|___       |
___________|______        |       ___________|______
flights from       Tel        Aviv arriving    at     New     York leaving  in
the  evening arriving  in         the  morning

SQL query: SELECT DISTINCT flightid from Flights WHERE origin == "Tel Aviv" AND
destination == "New York" AND departureTime == "Evening" AND arrivalTime ==
"Morning" AND 1
Result:
```

```
('LY01',)
```

Write your own sentence (that our syntactic grammar can parse) and check its results:

```
[142]: #TODO - Write your own sentence that is parsable
       your_query = "flights"
       query_to_answer(your_query, parser, augmentations)
```

```
   Q
   |
   NP
   |
flights

SQL query: SELECT DISTINCT flightid from Flights WHERE 1
Result:
('DL10',)
('DL11',)
('DL13',)
('LY01',)
('LY12',)
```

```
[142]: [('DL10',), ('DL11',), ('DL13',), ('LY01',), ('LY12',)]
```

```
[143]: grader.check("your_own")
```

```
[143]:

     All tests passed!
```

Our semantic parser requires that the natural language input sentence be syntactically well-formed according to the grammar. But what happens if the sentence fails to parse syntactically?

For example, our parser can parse the sentence `flights from Boston`, but it cannot parse the sentence `show me flights from Boston`:

```
[144]: try:
           query_to_answer("show me flights from Boston", parser, augmentations)
       except ValueError as e:
           print(e)
```

```
Grammar does not cover some of the input words: "'show', 'me'".
```

If the sentence does not parse syntactically, our semantic parser won't be able to interpret it semantically.

One possible way to address this problem is to use a *partial* syntactic parser – a parser that provides a parse tree for the longest subsentence that it can parse. In the case of `show me flights from Boston`, such a partial parser will drop the words `show me` and provide the parse tree only for `flights from boston`.

Another possible solution is to use neural sequence-to-sequence methods, that will naturally address this problem by having an embedding for "unknown" words (usually referred to as `<UNK>`). We'll turn to those in later labs.

5 Lab debrief

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 might 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?

but you should comment on whatever aspects you found especially positive or negative.

Type your answer here, replacing this text.

6 End of Lab 4-2

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

[145]:
```
grader.check_all()
```

[145]:
```
add_flight:

    All tests passed!


interpreting:

    All tests passed!


your_own:

    All tests passed!
```