project4_semantics

November 12, 2020

[ ]:
```
# 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/cs187-2020/project4.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

1 Project 4: Semantic Parsing for Question Answering

The goal of semantic parsing is to convert natural language utterances to a meaning representation such as a *logical form* expression or a *SQL query*. In the previous project segment, you built a parsing system to reconstruct parse trees from the natural-language queries in the ATIS dataset. However, that only solves an intermediary task, not the end-user task of obtaining answers to the queries.

In this the final project segment, you will go further, building a semantic parsing system to convert the English queries to SQL queries, so that by consulting a database you will be able to answer those questions. You will implement both a rule-based approach and an end-to-end sequence-to-sequence (seq2seq) approach. Both algorithms come with their pros and cons, and by the end of this segment you should have a basic understanding of the characteristics of the rule-based computational linguistic approach and the neural approach.

1.1 Goals

1. Build a semantic parsing algorithm to convert text to SQL queries based on the syntactic parse trees from the last project.
2. Build an end-to-end seq2seq system to convert text to SQL.
3. Discuss the pros and cons of the rule-based system and the end-to-end system.

This will be a very challenging homework, so we recommend that you start early.

1.2 Setup

[ ]:
```
import collections
import pprint
import re
import warnings

import nltk
import sqlite3
import torch
import torch.nn as nn
import torchtext as tt

import scripts.transform as xform

from cryptography.fernet import Fernet
from nltk import treetransforms
from nltk.corpus import treebank
from nltk.grammar import ProbabilisticProduction, CFG, PCFG, induce_pcfg,␣
↪Nonterminal, Production
from nltk.parse import pchart
from nltk.tree import Tree
```

```
from torch.nn.utils.rnn import pack_padded_sequence as pack
from tqdm import tqdm
```

[ ]:
```
# Set random seeds
seed = 1234
torch.manual_seed(seed)

# GPU check, make sure to set runtime type to "GPU" where available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print (device)
```

[ ]:
```
## Download needed scripts and data
source_url = "https://raw.githubusercontent.com/nlp-course/data/master"

...
# Grammar to augment for this segment
if not os.path.isfile('data/grammar'):
  shell(f"wget -nv -N -P data {source_url}/ATIS/grammar_distrib4.crypt")

  # Decrypt the grammar file
  key = b'bfksTY2BJ5VKKK9xZb1PDDLaGkdu7KCDFYfVePSEfGY='
  fernet = Fernet(key)
  with open('./data/grammar_distrib4.crypt', 'rb') as f:
    restored = Fernet(key).decrypt(f.read())
  with open('./data/grammar', 'wb') as f:
    f.write(Fernet(key).encrypt(restored))

# Download scripts and ATIS database
shell(f"""
  wget -nv -N -P scripts {source_url}/scripts/trees/transform.py
  wget -nv -N -P data {source_url}/ATIS/atis_sqlite.db
  """)
```

1.3 Semantically augmented grammars

In the first part of this project segment, you'll be implementing a rule-based system for semantic interpretation of sentences. Before jumping into using such a system on the ATIS dataset – we'll get to that soon enough – let's first work with some trivial examples to get things going.

The fundamental idea of rule-based semantic interpretation is the rule of compositionality, that **the meaning of a constituent is a function of the meanings of its immediate subconstituents and the syntactic rule that combined them**. This leads to an infrastructure for specifying semantic interpretation in which each syntactic rule in a grammar (in our case, a context-free grammar) is associated with a semantic rule that applies to the meanings associated with the elements on the right-hand side of the rule.

As a first example, let's consider a grammar for arithmetic expressions, familiar from lab 3-1.

Instead of reconstructing the grammar using `nltk.Grammar.from_string`, we've provided a slightly more sophisticated specification using the function `xform.parse_augmented_grammar`. You can read more about it in the file `transform.py`, which we downloaded above.

[ ]:
```
arithmetic_grammar, arithmetic_augmentations = xform.parse_augmented_grammar(
    """
    ## Sample grammar for arithmetic expressions

    S -> NUM                                  : lambda Num: Num
       | S OP S                               : lambda S1, Op, S2: Op(S1, S2)

    OP -> ADD                                 : lambda Op: Op
        | SUB
        | MULT
        | DIV

    NUM -> 'zero'                             : lambda: 0
         | 'one'                              : lambda: 1
         | 'two'                              : lambda: 2
         | 'three'                            : lambda: 3
         | 'four'                             : lambda: 4
         | 'five'                             : lambda: 5
         | 'six'                              : lambda: 6
         | 'seven'                            : lambda: 7
         | 'eight'                            : lambda: 8
         | 'nine'                             : lambda: 9
         | 'ten'                              : lambda: 10

    ADD -> 'plus' | 'added' 'to'              : lambda: lambda x, y: x + y
    SUB -> 'minus'                            : lambda: lambda x, y: x - y
    MULT -> 'times' | 'multiplied' 'by'       : lambda: lambda x, y: x * y
    DIV -> 'divided' 'by'                     : lambda: lambda x, y: x / y
    """)
```

There are several things to note in this grammar specification format:

1. Comment lines can be added, starting with `#`.
2. Blank lines are ignored.
3. Alternative right-hand sides can be split onto successive lines that start with `|`.
4. Each rule can be given an *augmentation*, which is an arbitrary Python expression.
5. Rules that are not explicitly provided with an augmentation (like all the `OP` rules after the first) are associated with the textually most recent one.

The `parse_augmented_grammar` function returns both an NLTK grammar and a dictionary that maps from productions in the grammar to their associated augmentations. Let's examine the returned grammar.

```
[ ]: import inspect
     for production in arithmetic_grammar.productions():
       print(f"{repr(production):25}     {arithmetic_augmentations[production]}")
```

We can parse with the grammar using one of the built-in NLTK parsers.

```
[ ]: arithmetic_parser = nltk.parse.BottomUpChartParser(arithmetic_grammar)
     parses = [p for p in arithmetic_parser.parse('three plus one times four'.
      ↪split())]
     for parse in parses:
       parse.pretty_print()
       print(parse)
```

Now let's turn to the augmentations. Although augmentations can be arbitrary Python expressions evaluating to arbitrary Python values, the intention is that they will often be functions that are intended to be applied to the values associated with the right-hand side elements. This enables a simple inductive evaluation of trees according to the following pseudo-code:

```
to evaluate a tree:
  evaluate each of the nonterminal-rooted subtrees
  find the augmentation associated with the root production of the tree
   (it should be a function of as many arguments as there are nonterminals on the right-hand side)
  return the result of applying the augmentation to the subtree values
```

(The base case of this recursion occurs when the number of nonterminal-rooted subtrees is zero, that is, a rule all of whose right-hand side elements are terminals. We'll have more to say about dealing with terminal symols shortly.)

Suppose we had such a function, call it `eval-tree`. How would it operate on, for instance, the tree `(S (S (NUM three)) (OP (ADD plus)) (S (NUM one)))`?

```
evaluate (S (S (NUM three)) (OP (ADD plus)) (S (NUM one)))
    evaluate (S (NUM three))
        evaluate (NUM three)
            (no subconstituents to evaluate)
          apply the augmentation for the rule NUM -> three to the empty set of values
            (lambda: 3) () ==> 3
        ==> 3
    evaluate (OP (ADD plus))
        ...
        ==> lambda x, y: x + y
    evaluate (S (NUM one))
        ...
        ==> 1
   apply the augmentation for the rule S -> S OP S to the values 3, (lambda x, y: x + y), and 1
        (lambda S1, Op, S2: Op(S1, S2))(3, (lambda x, y: x + y), 1) ==> 4
    ==> 4
```

Thus, the string "three plus one" is semantically interpreted as the value 4.

Now, all you need to do is to write the `eval_tree` function.

```
[ ]: # TODO -- Write the `eval_tree` function
def eval_tree(tree, grammar, augmentations):
  """Returns the value associated with the `tree` by inductive evaluation
     according to the `grammar` and its `augmentations` """
  ...
```

Now we should be able to evaluate the arithmetic example from above.

```
[ ]: eval_tree(parses[0], arithmetic_grammar, arithmetic_augmentations)
```

And we can even write a function that parses and interprets a string. We'll have it evaluate each of the possible parses and print the results.

```
[ ]: def interpret(string, grammar, augmentations):
  parser = nltk.parse.BottomUpChartParser(grammar)
  parses = parser.parse(string.split())
  for parse in parses:
    print(parse, "==>", eval_tree(parse, grammar, augmentations))
```

```
[ ]: interpret("three plus one times four", arithmetic_grammar,␣
 ↪arithmetic_augmentations)
```

Before going on, it will be useful to have a few more conveniences in writing augmentations for rules. First, since the augmentations are arbitrary Python expressions, they can be built from and make use of other functions. For instance, you'll notice that many of the augmentations at the leaves of the tree took no arguments and returned a constant. We can define a function `constant` that returns a function that ignores its arguments and returns a particular value.

```
[ ]: def constant(value):
  return lambda *args: value
```

Similarly, several of the augmentations are functions that just return their first argument. Again, we can define a generic form `first` of such a function:

```
[ ]: def first(*args):
  return args[0]
```

We can now rewrite the grammar above to take advantage of these shortcuts.

> In the call to `parse_augmented_grammar` below, we pass in the global environment, extracted via a `globals` function call, via the argument `globals`. This allows the `parse_augmented_grammar` function to make use of the global bindings for `constant`, `first`, and the like when evaluating the augmentation expressions to their values. You can check out the code in `transform.py` to see how the passed in `globals` bindings are used.

```
[ ]: arithmetic_grammar_2, arithmetic_augmentations_2 = xform.
 ↪parse_augmented_grammar(
    """
```

```
    ## Sample grammar for arithmetic expressions

    S -> NUM                                  : first
       | S OP S                               : lambda S1, Op, S2: Op(S1, S2)

    OP -> ADD                                 : first
       | SUB
       | MULT
       | DIV

    NUM -> 'zero'                             : constant(0)
         | 'one'                              : constant(1)
         | 'two'                              : constant(2)
         | 'three'                            : constant(3)
         | 'four'                             : constant(4)
         | 'five'                             : constant(5)
         | 'six'                              : constant(6)
         | 'seven'                            : constant(7)
         | 'eight'                            : constant(8)
         | 'nine'                             : constant(9)
         | 'ten'                              : constant(10)

    ADD -> 'plus' | 'added' 'to'              : constant(lambda x, y: x + y)
    SUB -> 'minus'                            : constant(lambda x, y: x - y)
    MULT -> 'times' | 'multiplied' 'by'       : constant(lambda x, y: x * y)
    DIV -> 'divided' 'by'                     : constant(lambda x, y: x / y)
    """,
    globals=globals())
```

Finally, it might be occasionally useful to allow the augmentations to get ahold of the terminal symbols in the rule, in addition to the nonterminal-subtree values already available to it. Since it's only occasionally useful, we might place the list of right-hand-side terminal symbols in a named argument `terms`. This would allow us to write, for example,

[ ]:
```
def numeric(*args, terms):
  return {'zero':0, 'one':1, 'two':2, 'three':3, 'four':4, 'five':5,
          'six':6, 'seven':7, 'eight':8, 'nine':9, 'ten':10}[terms[0]]
```

and further simplify the grammar specification:

[ ]:
```
arithmetic_grammar_3, arithmetic_augmentations_3 = xform.
↪parse_augmented_grammar(
    """
    ## Sample grammar for arithmetic expressions

    S -> NUM                                  : first
       | S OP S                               : lambda S1, Op, S2: Op(S1, S2)
```

```
    OP -> ADD                               : first
       | SUB
       | MULT
       | DIV

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

    ADD -> 'plus' | 'added' 'to'            : constant(lambda x, y: x + y)
    SUB -> 'minus'                          : constant(lambda x, y: x - y)
    MULT -> 'times' | 'multiplied' 'by'     : constant(lambda x, y: x * y)
    DIV -> 'divided' 'by'                   : constant(lambda x, y: x / y)
    """,
    globals=globals())
```

[ ]:
```
interpret("six divided by three", arithmetic_grammar_3,␣
↪arithmetic_augmentations_3)
```

1.3.1 Another simple example

This stuff is tricky, so it's useful to see more examples before jumping in the deep end. In this simple GEaH fragment grammar, we use a larger set of auxiliary functions to build the augmentations.

[ ]:
```
def constant(value):
  """Return `value`, ignoring any arguments"""
  return lambda *args: value

def forward(F, A):
  """Forward application: Return the application of the first argument to the␣
↪second"""
  return F(A)

def backward(A, F):
  """Backward application: Return the application of the second argument to the␣
↪first"""
  return F(A)

def first_term(*args, terms=[]):
  """Return the first (and perhaps only) terminal symbol on the right-hand␣
↪side"""
  return terms[0]

def first(*args):
```

```
  """Return the value of the first (and perhaps only) subconstituent, ignoring␣
 ↪any others"""
  return args[0]

def second(*args):
  """Return the value of the second subconstituent, ignoring any others"""
  return args[1]

def ignore(*args):
  """Return `None`, ignoring everything about the constituent. (Good as a␣
 ↪placeholder until
     a better augmentation can be devised.)"""
  return None
```

Using these, we can build and test the grammar.

[ ]:
```
geah_grammar_spec = """
  ## Productions
  S -> NP VP          : backward
  VP -> V NP          : forward

  ## Lexicon
  V -> 'likes'        : constant(lambda Object: lambda Subject:␣
 ↪f"likes({Object})({Subject})")
  NP -> 'Sam' | 'sam' : first_term
  NP -> 'ham'
  NP -> 'eggs'
"""
```

[ ]:
```
geah_grammar, geah_augmentations = xform.
 ↪parse_augmented_grammar(geah_grammar_spec,
                                                    globals=globals())
```

[ ]:
```
geah_parser = nltk.parse.BottomUpChartParser(geah_grammar)
parses = [p for p in geah_parser.parse('Sam likes ham'.split())]
parses[0]
```

[ ]:
```
print(eval_tree(parses[0], geah_grammar, geah_augmentations))
```

Now you're in a good position to understand and add augmentations to a more comprehensive grammar, say, one that parses ATIS queries and generates SQL queries.

In preparation for that, we need to load the ATIS data, both NL and SQL queries.

1.4 Loading and preprocessing the corpus

To simplify things a bit, we'll only consider ATIS queries whose question type (remember that from project segment 1?) is `flight_id`. We download training, development, and test splits for this subset of the ATIS corpus, including corresponding SQL queries.

[ ]:
```
# Acquire the datasets -- training, development, and test splits of the
# ATIS queries and corresponding SQL queries
shell(f"""
  wget -nv -N -P data {source_url}/ATIS/test_flightid.nl
  wget -nv -N -P data {source_url}/ATIS/test_flightid.sql
  wget -nv -N -P data {source_url}/ATIS/dev_flightid.nl
  wget -nv -N -P data {source_url}/ATIS/dev_flightid.sql
  wget -nv -N -P data {source_url}/ATIS/train_flightid.nl
  wget -nv -N -P data {source_url}/ATIS/train_flightid.sql
""")
```

Let's take a look at the data: the NL queries are in `.nl` files, and the SQL queries are in `.sql` files.

[ ]:
```
shell("head -1 data/dev_flightid.nl")
shell("head -1 data/dev_flightid.sql")
```

1.4.1 Corpus preprocessing

We'll use `torchtext` to process the data. We use two `Fields`: `TEXT` for the questions, and `SQL` for the SQL queries.

[ ]:
```
## Turn off annoying torchtext warnings about pending deprecations
warnings.filterwarnings("ignore", module="torchtext", category=UserWarning)
```

[ ]:
```
TEXT = tt.data.Field(lower=True,                    # lowercased
                     sequential=True,               # sequential data
                     include_lengths=True,          # include lengths
                     batch_first=False,             # batches will be max_len X␣
 ↪batch_size
                     tokenize=lambda x: x.split(), # use split to tokenize
                     preprocessing=lambda lst: list(reversed(lst))) # reverse␣
 ↪the text before padding
SQL = tt.data.Field(sequential=True,
                    include_lengths=False,
                    batch_first=False,
                    tokenize=lambda x: x.split(),
                    init_token="<bos>",             # prepend <bos>
                    eos_token="<eos>")              # append <eos>
fields = [('text_reversed', TEXT), ('sql', SQL)]
```

Note that we reversed the tokens in question using the `preprocessing` argument. We

did that because in seq2seq (without attention) this trick improves performance. You can refer to Section 3.3 in the seminal seq2seq paper for more details. We also specify `batch_first=False`, so that the returned batched tensors would be of size `max_length X batch_size`, which facilitates seq2seq implementation.

Now, we load the data using `torchtext`. We use the `TranslationDataset` class here because our task is essentially a translation task: "translating" questions into the corresponding SQL queries. Therefore, we also refer to the questions as the *source* side and the SQL queries as the *target* side.

[ ]:
```
# Make splits for data
train_data, val_data, test_data = tt.datasets.TranslationDataset.splits(
    ('_flightid.nl', '_flightid.sql'), fields, path='./data/',
    train='train', validation='dev', test='test')

MIN_FREQ = 3
TEXT.build_vocab(train_data.text_reversed, min_freq=MIN_FREQ)
SQL.build_vocab(train_data.sql, min_freq=MIN_FREQ)

print (f"Size of English vocab: {len(TEXT.vocab)}")
print (f"Most comman English words: {TEXT.vocab.freqs.most_common(10)}\n")

print (f"Size of SQL vocab: {len(SQL.vocab)}")
print (f"Most comman SQL words: {SQL.vocab.freqs.most_common(10)}\n")

print (f"Index for start of sequence token: {SQL.vocab.stoi[SQL.init_token]}")
print (f"Index for end of sequence token: {SQL.vocab.stoi[SQL.eos_token]}")
```

Next, we batch our data to facilitate processing on a GPU. Batching is a bit tricky because the source and target will typically be of different lengths. Fortunately, `torchtext` allows us to pass in a `sort_key` function. By sorting on length, we can minimize the amount of padding on the source side, but since there is still some padding, we need to handle them with `pack` later on in the seq2seq part.

[ ]:
```
BATCH_SIZE = 32 # batch size for training/validation
TEST_BATCH_SIZE = 1 # batch size for test, we use 1 to make implementation easier

train_iter, val_iter = tt.data.BucketIterator.splits((train_data, val_data),
                                                     batch_size=BATCH_SIZE,
                                                     device=device,
                                                     repeat=False,
                                                     sort_key=lambda x: len(x.
 ↪text_reversed),
                                                     sort_within_batch=True)
test_iter = tt.data.BucketIterator(test_data,
                                   batch_size=1,
                                   device=device,
                                   repeat=False,
                                   sort=False,
```

```
                                        train=False)
```

Let's look at a single batch from one of these iterators.

[ ]:
```
batch = next(iter(train_iter))
text, text_lengths = batch.text_reversed
print (f"Size of text batch: {text.shape}")
print (f"Third sentence in batch: {text[:, 2]}")
print (f"Length of the third sentence in batch: {text_lengths[2]}")
print (f"Converted back to string: {' '.join([TEXT.vocab.itos[i] for i in text[:
↪, 2]])}")

sql = batch.sql
print (f"Size of sql batch: {sql.shape}")
print (f"Third label in batch: {sql[:, 2]}")
print (f"Converted back to string: {' '.join([SQL.vocab.itos[i] for i in sql[:,␣
↪2]])}")
```

Note that the question is reversed, and that the size of the batch is `max_length X batch_size`.

Alternatively, we can directly iterate over the raw examples in `train_data`, `val_data`, and `test_data`.

[ ]:
```
for example in train_iter.dataset: # val_iter.dataset is just val_data
  text_reversed = example.text_reversed
  text = ' '.join(reversed(text_reversed)) # detokenized question
  sql = ' '.join(example.sql) # detokenized sql
  print (f"Question: {text}")
  print (f"SQL: {sql}")
  break
```

1.5 Establishing a SQL database for evaluating ATIS queries

The output of our systems will be SQL queries. How to determine if the generated queries are correct? We can't merely compare against the gold SQL queries, since there are many ways to implement a SQL query that answers any given NL query.

Instead, we will execute the queries – both the predicted SQL query and the gold SQL query – on an actual database, and verify that the returned responses are the same. For that purpose, we need a SQL database server to use. We'll set one up here, using the Python `sqlite3` module.

[ ]:
```
conn = sqlite3.connect('data/atis_sqlite.db')  # establish the DB based on the␣
↪downloaded data
c = conn.cursor()                              # build a "cursor"
```

To run a query, we use the cursor's `execute` function, and retrieve the results with `fetchall`. Let's get all the flights that arrive at General Mitchell International – the query above. There's a lot, so we'll just print out the first few.

[ ]:
```
c.execute(sql)
predicted_ret = c.fetchall()

pprint.pprint(predicted_ret[:10])
len(predicted_ret)
```

1.6 Rule-based parsing and interpretation of ATIS queries

First, we will implement a rule-based semantic parser using a grammar like the one you completed in the third project segment. We've placed an initial grammar in the file `data/grammar`. We can build a parser with it:

[ ]:
```
atis_grammar, atis_augmentations = xform.read_augmented_grammar('data/grammar')
```

and parse the same sample query:

[ ]:
```
atis_parser = nltk.parse.BottomUpChartParser(atis_grammar)
parses = [p for p in atis_parser.parse(text.split())]
print(text)
print("Number of parses:", len(parses))
for parse in parses:
  parse.pretty_print()
```

Let's check the coverage of this grammar on the training set.

[ ]:
```
# Check coverage on training set
parsed = 0
with open("data/train_flightid.nl") as train:
  examples = train.readlines()[:]
for sentence in tqdm(examples):
  try:
    if len(list(atis_parser.parse(sentence.strip().split()))) > 0:
      parsed += 1
    else:
      next
  except:
    pass

print(f"Parsed {parsed} of {len(examples)} ({parsed/(len(examples))}%)")
```

The grammar that we've provided is able to parse about half of the queries in the training set.

1.7 Semantic interpretation for the ATIS grammar

2 *To be rewritten*

Recall that the in rule-based semantic parsing each syntactic rule is associated with a semantic rule. Given a sentence, we first construct its parse tree using the syntactic rules, then compose the corresponding semantic rules bottom-up, until eventually we arrive at the root node with a finished SQL statement.

We use the above parse tree as an example.

1. First, let the rule

FLIGHT -> flights

be accompanied by the semantic rule:

SELECT DISTINCT flight.flight_id FROM flight.

2. To handle origin/destination constraint 'boston', we associate

Place -> boston

with

(SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code FROM city WHERE city.city_name = 'boston')).

> Note that we look up the airport code instead of directly using the city code, because the flight table which we later use expects the airport code.

3. To distinguish destination from origin, we need to add a rule for:

PPLACE -> to.

We use lambda calculus here, since the SQL statement it produces is dependent on its siblings ('to boston' is different from 'to dallas'):

λ **x. "(flight.to_airport IN (" + x + "))"**.

4. Now we need to merge *PPLACE* and *PLACE* at node *PP*:

PP -> PPLACE PLACE.

We simply use:

left_child(right_child),

which denotes evaluating the left child to get a function, then applying that function with the right child as the input. In this case, this would evaluate to:

(flight.to_airport IN (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code FROM city WHERE city.city_name = 'boston')))

5. For the rule

PPS -> PP,

we simply copy the evaluation result of the child:

child.

6. Finally, the last piece to complete the puzzle is at the root node:

S -> FLIGHT PPS,

for which we only need to join the evaluation results of its left child and right child with a 'WHERE':

left_child WHERE right_child.

Putting all these together, the final SQL statement we get (at root 'S') is:

SELECT DISTINCT flight.flight_id FROM flight WHERE (flight.to_airport IN (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code FROM city WHERE city.city_name = 'boston'))),

which should return the answer to the original question when used to query a MySQL database containing relevant flight information.

In the ATIS grammar that we provide, as with the earlier toy grammars, the augmentation for a rule with n nonterminals and m terminals on the right-hand side is assumed to be called with n positional arguments (the values for the corresponding children), and a keyword argument `terms` whose value is a list of the m terminal symbols. The `eval_tree` function you've already defined should therefore work well with this grammar.

It makes use of a broader set of auxiliary functions as defined below.

[ ]:
```
# Some useful auxiliary functions
def constant(value):
  return lambda *args: value

def forward(F, A):
  return F(A)

def backward(A, F):
  return F(A)

def first_term(*args, terms=[]):
  return terms[0].upper()

def first(*args):
  return args[0]

def second(*args):
  return args[1]

def ignore(*args):
  return None
```

```
def concat(*args):
  return ' '.join(filter(None, args))

def select_flight():
  return lambda condition: f"""
    SELECT DISTINCT flight.flight_id FROM flight WHERE
      {condition}
    """.strip()

def no_change(*args, **kwargs):
  return lambda condition: condition

def either_condition(cond1, cond2):
   return lambda condition: f"""
     (f{cond1('True')} OR f{cond2('True')}) AND {condition}
     """.strip()

def both_condition(cond1, cond2):
   return lambda condition: cond1(cond2(condition))

def from_city(city):
  return lambda condition: f"""
    (flight.from_airport IN
     (SELECT airport_service.airport_code FROM airport_service WHERE␣
 ↪airport_service.city_code IN
        (SELECT city.city_code FROM city WHERE city.city_name = '{city}'))) AND
    {condition}
    """.strip()

def to_city(city):
  return lambda condition: f"""
    (flight.to_airport IN
     (SELECT airport_service.airport_code FROM airport_service WHERE␣
 ↪airport_service.city_code IN
        (SELECT city.city_code FROM city WHERE city.city_name = '{city}'))) AND
    {condition}
    """.strip()

def airline(aircode):
  return lambda condition: f"""
    (flight.airline_code = '{aircode}') AND
    {condition}
    """.strip()

def weekday(day):
  return lambda condition: f"""
```

```
    (flight.flight_days IN (SELECT days.days_code FROM days WHERE days.day_name␣
 ↪= '{day.upper()}')) AND
    {condition}
    """.strip()

def depart_around(time):
  return lambda condition: f"""
    (flight.departure_time >= {add_delta(miltime(time), -15).strftime('%H%M')}
     AND flight.departure_time <= {add_delta(miltime(time), 15).
 ↪strftime('%H%M')})
    AND {condition}
    """.strip()

def depart_at(time):
  return lambda condition: f"""
    (flight.departure_time = {miltime(time).strftime('%H%M')})
    AND {condition}
    """.strip()

def depart_before(time):
  return lambda condition: f"""
    (flight.departure_time <= {miltime(time).strftime('%H%M')})
    AND {condition}
    """.strip()

def depart_after(time):
  return lambda condition: f"""
    (flight.departure_time >= {miltime(time).strftime('%H%M')})
    AND {condition}
    """.strip()

def arrive_around(time):
  return lambda condition: f"""
    (flight.arrival_time >= {add_delta(miltime(time), -15).strftime('%H%M')}
     AND flight.arrival_time <= {add_delta(miltime(time), 15).strftime('%H%M')})
    AND {condition}
    """.strip()

def arrive_at(time):
  return lambda condition: f"""
    (flight.arrival_time = {miltime(time).strftime('%H%M')})
    AND {condition}
    """.strip()

def arrive_before(time):
  return lambda condition: f"""
    (flight.arrival_time <= {miltime(time).strftime('%H%M')})
```

```
    AND {condition}
    """.strip()

def arrive_after(time):
  return lambda condition: f"""
    (flight.arrival_time >= {miltime(time).strftime('%H%M')})
    AND {condition}
    """.strip()

def on_date(year=1991, month=4, day=1):
  return lambda condition: f"""
    (flight.flight_days IN
       (SELECT days.days_code FROM days
          WHERE days.day_name IN
          (SELECT date_day.day_name FROM date_day
             WHERE date_day.year = {year}
             AND date_day.month_number = {month}
             AND date_day.day_number = {day}))
     AND {condition})
    """.strip()

def month_name(month):
  return {'JANUARY' : 1,
          'FEBRUARY' : 2,
          'MARCH' : 3,
          'APRIL' : 4,
          'MAY' : 5,
          'JUNE' : 6,
          'JULY' : 7,
          'AUGUST' : 8,
          'SEPTEMBER' : 9,
          'OCTOBER' : 10,
          'NOVEMBER' : 11,
          'DECEMBER' : 12}[month.upper()]

def miltime(minutes):
  return datetime.time(hour=int(minutes/100), minute=(minutes % 100))

def add_delta(tme, delta):
    # transform to a full datetime first
    return (datetime.datetime.combine(datetime.date.today(), tme) +
            datetime.timedelta(minutes=delta)).time()
```

[ ]:
```
atis_grammar, atis_augmentations = xform.read_augmented_grammar("data/grammar",
                                                                globals=globals())
atis_parser = nltk.parse.SteppingChartParser(atis_grammar, strategy=nltk.parse.
 ↪chart.BU_STRATEGY, trace=0)
```

[ ]:
```
from nltk.tokenize import word_tokenize

def parse_tree(sentence):
  """Parse a sentence and return the parse tree, or None if failure."""
  try:
    parses = list(atis_parser.parse(word_tokenize(sentence.lower())))
    if len(parses) == 0:
      return None
    else:
      return parses[0]
  except Exception as e:
    if DEBUG:
      print(f"Error: {e}")
    return None
```

[ ]:
```
parse = parse_tree('tuesday flights')
parse
```

[ ]:
```
testing = eval_tree(parse, atis_grammar, atis_augmentations)
print(testing)
```

[ ]:
```
c.execute(testing)
c.fetchall()[:10]
```

2.0.1 Goal 1: Construct SQL queries from a parse tree and evaluate the results

Implement a rule-based semantic parsing system to successfully answer **at least 25%** of flight_id type questions in the test set.

Evaluation With a rule-based semantic parsing system, we can generate SQL queries given questions, and then execute those queries on a MySQL database to answer the given questions. To evaluate the performance of the system, we compare the returned results against the results of executing the ground truth queries. Note that we do not directly compare the predicted SQL queries to the gold SQL queries due to there being multiple ways of writing semantically equivalent queries.

We provide a function `evaluate_accuracy` to compare the results from our generated SQL to the ground truth SQL.

[ ]:
```
def evaluate_accuracy(predictions, sqls, questions=None):
  """
  Evaluate accuracy by executing predictions on a remote MySQL database
  and comparing returned results.
  Arguments:
      predictions: a list of predicted sqls or a single predicted sql.
      sqls: a list of gold sql statements or a single gold sql.
```

```
    questions: a list of questions or a single question. Optional.
Returns: accuracy.
"""
# Initial check for type of input
sqls = [sqls] if not isinstance(sqls, (list)) else sqls
predictions = [predictions] if not isinstance(predictions, (list)) else␣
↪predictions
if questions is not None:
  questions = [questions] if not isinstance(questions, (list)) else questions
else:
  questions = ['N/A',] * len(sqls)

# Connect to database
try:
  conn = sqlite3.connect('data/atis_sqlite.db')
except Exception as err:
  print(f"Something went wrong in establishing DB: {err}")
  return

c = conn.cursor()
#c.execute('USE atis;')

# Evaluate each query and compare results
correct = 0
total = len(sqls)
for gold_sql, predicted_sql, question in zip(sqls, predictions, questions):
  is_correct = True
  if len(predicted_sql) == 0:
    is_correct = False
  else:
    # Execute predicted sql
    try:
      c.execute(predicted_sql)
      predicted_ret = c.fetchall()
    except Exception as e:
      predicted_ret = 'Syntax Error!'
    # Execute gold sql
    try:
      c.execute(gold_sql)
      gold_ret = c.fetchall()
    except Exception as e:
      gold_ret = 'Syntax Error!'

    if gold_ret == predicted_ret:
      correct += 1
    else:
      is_correct = False
```

```
    if DEBUG and not is_correct:
      print (f"\nINCORRECT!")
      print (f"Question: {question}")
      print (f"Gold SQL: {gold_sql}")
      if len(gold_sql) > 0:
        print (f"Gold Result: {gold_ret[:50]}")
      print (f"Predicted SQL: {predicted_sql}")
      if len(predicted_sql) > 0:
        print (f"Predicted Result: {predicted_ret[:50]}")

  conn.commit()
  c.close()
  conn.close()
  return correct/total
```

To make development faster, we recommend starting with a few examples before running the full evaluation script.

[ ]:
```
DEBUG = False
```

[ ]:
```
# Example 1
question = 'flights from phoenix to milwaukee'
gold_sql = "SELECT DISTINCT flight_1.flight_id FROM flight flight_1 ,␣
 ↪airport_service airport_service_1 , city city_1 , airport_service␣
 ↪airport_service_2 , city city_2 WHERE flight_1.from_airport =␣
 ↪airport_service_1.airport_code AND airport_service_1.city_code = city_1.
 ↪city_code AND city_1.city_name = 'PHOENIX' AND flight_1.to_airport =␣
 ↪airport_service_2.airport_code AND airport_service_2.city_code = city_2.
 ↪city_code AND city_2.city_name = 'MILWAUKEE'"
tree = parse_tree(question)
tree.pretty_print()

predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

[ ]:
```
# Example 2
question = 'i would like a united flight'
gold_sql = "SELECT DISTINCT flight_1.flight_id FROM flight flight_1 WHERE␣
 ↪flight_1.airline_code = 'UA'"
tree = parse_tree(question)
tree.pretty_print()

predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

[ ]:
```
# Example 2
question = 'i would like a flight between boston and dallas'
```

```
gold_sql = "SELECT DISTINCT flight.flight_id FROM flight WHERE TRUE AND (flight.
 ↪from_airport IN (SELECT airport_service.airport_code FROM airport_service␣
 ↪WHERE airport_service.city_code IN (SELECT city.city_code FROM city WHERE␣
 ↪city.city_name = 'BOSTON'))) AND (flight.to_airport IN (SELECT␣
 ↪airport_service.airport_code FROM airport_service WHERE airport_service.
 ↪city_code IN (SELECT city.city_code FROM city WHERE city.city_name =␣
 ↪'DALLAS')))"
tree = parse_tree(question)
tree.pretty_print()

predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
print(predicted_sql)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

[ ]:
```
# Example 3
question = 'what flights are departing from houston or austin leaving at 7am␣
 ↪sunday'
gold_sql = "SELECT DISTINCT flight.flight_id FROM flight WHERE TRUE AND TRUE AND␣
 ↪((flight.from_airport IN (SELECT airport_service.airport_code FROM␣
 ↪airport_service WHERE airport_service.city_code IN (SELECT city.city_code␣
 ↪FROM city WHERE city.city_name = 'HOUSTON'))) OR (flight.from_airport IN␣
 ↪(SELECT airport_service.airport_code FROM airport_service WHERE␣
 ↪airport_service.city_code IN (SELECT city.city_code FROM city WHERE city.
 ↪city_name = 'AUSTIN')))) AND (flight.departure_time >= 630 AND flight.
 ↪departure_time <= 730) AND (flight.flight_days IN (SELECT days.days_code FROM␣
 ↪days WHERE days.day_name = 'sunday'))"
tree = parse_tree(question)
tree.pretty_print()

predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

[ ]:
```
# Example 4
question = 'can i have a flight from san francisco that stops in dallas going to␣
 ↪new york arriving before 6pm'
gold_sql = "SELECT DISTINCT flight.flight_id FROM flight WHERE TRUE AND (flight.
 ↪from_airport IN (SELECT airport_service.airport_code FROM airport_service␣
 ↪WHERE airport_service.city_code IN (SELECT city.city_code FROM city WHERE␣
 ↪city.city_name = 'SAN FRANCISCO'))) AND (flight_stop.stop_airport IN (SELECT␣
 ↪airport_service.airport_code FROM airport_service WHERE airport_service.
 ↪city_code IN (SELECT city.city_code FROM city WHERE city.city_name =␣
 ↪'DALLAS'))) AND (flight.to_airport IN (SELECT airport_service.airport_code␣
 ↪FROM airport_service WHERE airport_service.city_code IN (SELECT city.
 ↪city_code FROM city WHERE city.city_name = 'NEW YORK'))) AND (flight.
 ↪arrival_time <= 1800)"
tree = parse_tree(question)
```

```
tree.pretty_print()

predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

Below is the full evaluation code. You should be able to get correct results on at least 25% of `flight_id` type questions from the test set.

[ ]:
```
questions = []
predictions = []
gold_sqls = []

DEBUG = False
for example in tqdm(test_iter.dataset):
  # Input and output
  text_reversed = example.text_reversed
  question = ' '.join(reversed(text_reversed)) # detokenized question
  gold_sql = ' '.join(example.sql) # detokenized sql
  questions.append(question)
  gold_sqls.append(gold_sql)
  # Get parse tree
  tree = parse_tree(question)
  if tree is None:
    predictions.append('')
    continue
  # Predict
  try:
    predicted_sql = eval_tree(tree, atis_grammar, atis_augmentations)
  except Exception as e:
    predictions.append('')
    continue
  predictions.append(predicted_sql)

evaluate_accuracy(predictions, gold_sqls, questions)
```

2.1 End-to-End Seq2Seq Model

Nowadays neural networks dominate the field of NLP research. In this part, we investigate if it is possible to use an end-to-end system to directly learn the mapping from the natural language questions to the SQL queries.

2.1.1 Goal 2: Implement a seq2seq model

Model, Optimization and Decoding For the sequence-to-sequence model, you need to implement the class `EncoderDecoder`. We have provided starter code for performing optimization, but there are at least five methods that you need to implement:

1. `__init__`: an initializer where you can create network modules.
2. `forward`: given question word ids of size `batch_size X max_length`, question lengths of size `batch_size` and SQL word ids `batch_size X max_length_sql`, returns logits `batch_size X max_length_sql`. Note that here the batch size can be greater than 1.
3. `compute_loss`: computes loss by comparing output returned by forward to ground_truth which stores the true SQL word ids.
4. `evaluate_ppl`: evaluate the current model's perplexity on a given dataset iterator. Perplexity is defined as $\exp(-\frac{\text{total log likelihood)}}{\text{total number of words}})$, which can be roughly understood as how many random guesses the model needs to make to get a word correct.
5. `predict`: Generates the target sequence (SQL) given the source sequence (question). Note that here you can assume the batch size to be always 1 for simplicity. Besides, you can use greedy decoding here, i.e., predicting the word with the highest probability at any time step, although in practice researchers use more complicated decoding methods such as beam search.

This implementation is essentially building an entire neural seq2seq system, so expect it to be very challenging. The code you write here can also be used for other seq2seq tasks such as machine translation and document summarization.

*Hint: to handle source side paddings in `torch`, you can use somethine like `packed_src = pack(src, src_lengths)`. To handle target side paddings, you can use `ignore_index` when creating the loss function.

[ ]:
```
#TODO
class EncoderDecoder(nn.Module):
  def __init__(self, text, sql, embedding_size=512, hidden_size=512, layers=2,
               dropout=0, bidirectional=False,␣
↪share_decoder_input_output_embeds=False,
               add_encoder_out_to_decoder_input=False):
    """
    Initializer. Creates network modules and loss function. You do not need to
    implement all features as long as you can achieve 30%+ accuracy.
    Arguments:
        text: text field
        tag: sql field
        embedding_size: word embedding size
        hidden_size: hidden layer size
        layers: number of layers
        dropout: dropout
        bidirectional: use bidirectional RNN cells
        share_decoder_input_output_embeds: if True, set the weight matrix of the
            final projection layer to be the same as decoder word embeddings.
            This reduces the number of parameters and is found to improve␣
↪performance.
            See https://arxiv.org/pdf/1608.05859.pdf.
        add_encoder_out_to_decoder_input: if True, add encoder output to every
            step of decoder input. This trick keeps the decoder from forgetting
```

```
            encoder outputs as it decodes.
    """
    super(EncoderDecoder, self).__init__()
    self.text = text
    self.sql = sql
    # Keep the vocabulary sizes available
    self.V_src = len(text.vocab.itos)
    self.V_tgt = len(sql.vocab.itos)
    # Get special word ids or tokens
    self.padding_id_src = text.vocab.stoi[text.pad_token]
    self.padding_id_tgt = sql.vocab.stoi[sql.pad_token]
    self.bos_id = sql.vocab.stoi[sql.init_token]
    self.eos_id = sql.vocab.stoi[sql.eos_token]
    self.eos_token = sql.eos_token

    # Keep parameters available
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size
    self.layers = layers
    self.dropout = dropout
    self.share_decoder_input_output_embeds = share_decoder_input_output_embeds
    self.bidirectional = bidirectional
    self.add_encoder_out_to_decoder_input = add_encoder_out_to_decoder_input

    #TODO: implement this method
    # Create essential modules and loss function
    "your code here"

def forward(self, src_words, src_lengths, tgt_words):
    """
    Performs forward computation, returns logits.
    Arguments:
        src_words: question batch of size batch_size X max_length
        src_lengths: question lengths of size batch_size
        tgt_words: sql batch of size batch_size X max_length
    """
    #TODO: implement this method
    "your code here"
    return logits

def compute_loss(self, logits, targets):
    """
    Computes loss function with logits and target.
    Arguments:
        logits: tensor of size batch_size X max_length X V_tgt
        targets: tensor of size batch_size X max_length
    """
```

```
    #TODO: implement this method
    "your code here"
    return loss

  def evaluate_ppl(self, iterator):
    """
    Returns the model's perplexity on a given dataset `iterator`. We will
    use it for model selection.
    """
    # Switch to eval mode
    self.eval()
    #TODO: implement this method
    "your code here"
    return perplexity

  def predict(self, src_words, src_lengths, max_tgt_length=200):
    """
    Generates the target sequence (SQL) given the source sequence (question).
    You only need to implemnt greedy decoding, i.e., at each decoding step,
    find the word with the highest probability.
    Note that for simplicity, we only use batch size 1.
    Arguments:
        src_words: a tensor of size (max_length, 1) storing question word ids.
        src_lengths: a tensor of size (1) storing question length.
        max_tgt_length: at most proceed this many steps of decoding
    Returns:
        a string of the generated SQL.
    """
    # Switch to eval mode
    self.eval()
    #TODO: implement this method
    "your code here"
    decoded = 'SELECT DISTINCE * FROM flight'
    return decoded

  def fit(self, train_iter, val_iter, epochs=50, learning_rate=3e-4):
    """Train the model."""
    # Switch the module to training mode
    self.train()
    # Use Adam to optimize the parameters
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_ppl = float('inf')
    best_model = None
    # Run the optimization for multiple epochs
    for epoch in range(epochs):
      total_words = 0
      total_loss = 0.0
```

```
      for batch in tqdm(train_iter):
        # Zero the parameter gradients
        self.zero_grad()

        # Input and target
        text, text_lengths = batch.text_reversed # text: max_length_text, bsz
        sql = batch.sql # max_length_sql, bsz
        sql_in = sql[:-1] # Remove <eos> for decode input
        sql_out = sql[1:] # Remove <bos> as target
        batch_size = sql.size(1)

        # Run forward pass and compute loss along the way.
        logits = self.forward(text, text_lengths, sql_in)
        loss = self.compute_loss(logits, sql_out)

        # Training stats
        num_sql_words = sql_out.ne(self.padding_id_tgt).float().sum().item()
        total_words += num_sql_words
        total_loss += loss.item()

        # Perform backpropagation
        loss.div(batch_size).backward()
        optim.step()

      # Evaluate and track improvements on the validation dataset
      validation_ppl = self.evaluate_ppl(val_iter)
      self.train()
      if validation_ppl < best_validation_ppl:
        best_validation_ppl = validation_ppl
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = total_loss / total_words
      print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
             f'Validation Perplexity: {validation_ppl:.4f}')
```

Solution

[ ]:
```
#Solution
class EncoderDecoder(nn.Module):
  def __init__(self, text, sql, embedding_size=512, hidden_size=512, layers=2,
               dropout=0, bidirectional=False,␣
 ↪share_decoder_input_output_embeds=False,
               add_encoder_out_to_decoder_input=False):
    """
    Initializer. Creates network modules and loss function. You do not need to
    implement all features as long as you can achieve 30%+ accuracy.
    Arguments:
        text: text field
```

```
        tag: sql field
        embedding_size: word embedding size
        hidden_size: hidden layer size
        layers: number of layers
        dropout: dropout
        bidirectional: use bidirectional RNN cells
        share_decoder_input_output_embeds: if True, set the weight matrix of the
            final projection layer to be the same as decoder word embeddings.
            This reduces the number of parameters and is found to improve␣
↪performance.
            See https://arxiv.org/pdf/1608.05859.pdf.
        add_encoder_out_to_decoder_input: if True, add encoder output to every
            step of decoder input. This trick keeps the decoder from forgetting
            encoder outputs as it decodes.
    """
    super(EncoderDecoder, self).__init__()
    self.text = text
    self.sql = sql
    # Keep the vocabulary sizes available
    self.V_src = len(text.vocab.itos)
    self.V_tgt = len(sql.vocab.itos)
    # Get special word ids or tokens
    self.padding_id_src = text.vocab.stoi[text.pad_token]
    self.padding_id_tgt = sql.vocab.stoi[sql.pad_token]
    self.bos_id = sql.vocab.stoi[sql.init_token]
    self.eos_id = sql.vocab.stoi[sql.eos_token]
    self.eos_token = sql.eos_token

    # Keep parameters available
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size
    self.layers = layers
    self.dropout = dropout
    self.share_decoder_input_output_embeds = share_decoder_input_output_embeds
    self.bidirectional = bidirectional
    self.add_encoder_out_to_decoder_input = add_encoder_out_to_decoder_input

    # Create essential modules
    self.word_embeddings_src = nn.Embedding(self.V_src, embedding_size)
    self.word_embeddings_tgt = nn.Embedding(self.V_tgt, embedding_size)
    self.dropout_layer = nn.Dropout(dropout)

    # RNN cells
    self.encoder_rnn = nn.LSTM(
      input_size    = embedding_size,
      hidden_size   = hidden_size//2 if bidirectional else hidden_size,
      num_layers    = layers,
```

```
      dropout       = dropout,
      bidirectional = bidirectional
    )
    self.decoder_rnn = nn.LSTM(
      input_size    = embedding_size,
      hidden_size   = hidden_size,
      num_layers    = layers,
      dropout       = dropout,
    )

    # Final projection layer
    self.hidden2output = nn.Linear(hidden_size, self.V_tgt)
    if share_decoder_input_output_embeds:
      self.hidden2output.weight = self.word_embeddings_tgt.weight

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum',
                                             ignore_index=self.padding_id_tgt)

  def encode(self, src_words, src_lengths):
    """Encode source words into a vector"""
    # Compute word embeddings
    src = self.word_embeddings_src(src_words) # max_len, bsz, embedding_size
    if isinstance(src_lengths, torch.LongTensor) \
            or isinstance(src_lengths, torch.cuda.LongTensor):
      src_lengths = src_lengths.tolist()
    # Deal with paddings
    packed_src = pack(src, src_lengths)
    # Forward RNN and return final state
    encoder_out = self.encoder_rnn(packed_src)[-1] # num_layers*num_directions,␣
↪bsz, hidden_size/num_directions
    # Reshape encoder_out for bidirectional case
    if self.bidirectional:
      batch_size = len(src_lengths)
      h, c = encoder_out
      h = h.view(-1, 2, batch_size, self.hidden_size//2) \
           .transpose(1, 2) \
           .contiguous().view(-1, batch_size, self.hidden_size) # num_layers,␣
↪bsz, hidden_size
      c = c.view(-1, 2, batch_size, self.hidden_size//2) \
           .transpose(1, 2) \
           .contiguous().view(-1, batch_size, self.hidden_size) # num_layers,␣
↪bsz, hidden_size
      encoder_out = (h, c)
    return encoder_out

  def decode(self, tgt_words, encoder_out, feed_decoder_input):
```

```
    """Decode based on encoder output"""
    # Compute word embeddings
    tgt = self.word_embeddings_tgt(tgt_words) # len, bsz, hidden
    # Optionally add feed_decoder_input to every step
    if feed_decoder_input is not None: # bsz, hidden
      tgt = tgt + feed_decoder_input.unsqueeze(0) # unsqueeze to 1, bsz, hidden
    # Forward decoder RNN and return all hidden states
    return self.decoder_rnn(tgt, encoder_out)[0]

  def forward(self, src_words, src_lengths, tgt_words):
    """
    Performs forward computation, returns logits.
    Arguments:
        src_words: question batch of size batch_size X max_length
        src_lengths: question lengths of size batch_size
        tgt_words: sql batch of size batch_size X max_length
    """
    # Forward encoder
    encoder_out = self.encode(src_words, src_lengths) # tuple of (h_final,␣
↪c_final)
    if self.share_decoder_input_output_embeds:
      # h_final/c_final size: num_layers, bsz, hidden_size
      # We only take the last layer to match shape of decoder inputs
      feed_decoder_input = encoder_out[0][-1] + encoder_out[1][-1] # bsz,␣
↪hidden_size
    else:
      feed_decoder_input = None
    # Forward decoder
    decoder_out = self.decode(tgt_words, encoder_out, feed_decoder_input)
    # Final projection to target vocabulary
    logits = self.hidden2output(self.dropout_layer(decoder_out))
    return logits

  def compute_loss(self, logits, targets):
    """
    Computes loss function with logits and target.
    Arguments:
        logits: tensor of size batch_size X max_length X V_tgt
        targets: tensor of size batch_size X max_length
    """
    return self.loss_function(logits.view(-1, self.V_tgt), targets.view(-1))

  def evaluate_ppl(self, iterator):
    """Returns the model's perplexity on a given dataset `iterator`."""
    # Switch to eval mode
    self.eval()
    total_loss = 0
```

```
    total_words = 0
    for batch in iterator:
      # Input and target
      text, text_lengths = batch.text_reversed
      sql = batch.sql # max_length_sql, bsz
      sql_in = sql[:-1] # remove <eos> for decode input
      sql_out = sql[1:] # remove <bos> as target
      # Forward to get logits
      logits = self.forward(text, text_lengths, sql_in)
      # Compute cross entropy loss
      loss = self.compute_loss(logits, sql_out)
      total_loss += loss.item()
      total_words += sql_out.ne(self.padding_id_tgt).float().sum().item()
    return math.exp(total_loss/total_words)

  def predict(self, src_words, src_lengths, max_tgt_length=200):
    """
    Generates the target sequence (SQL) given the source sequence (question).
    You only need to implemnt greedy decoding, i.e., at each decoding step,
    find the word with the highest probability.
    Note that for simplicity, we only use batch size 1.
    Arguments:
        src_words: a tensor of size (max_length, 1) storing question word ids.
        src_lengths: a tensor of size (1) storing question length.
        max_tgt_length: at most proceed this many steps of decoding
    Returns:
        a string of the generated SQL.
    """
    # Switch to eval mode
    self.eval()
    # Forward encoder
    encoder_out = self.encode(src_words, src_lengths) # tuple of (h_final,␣
↪c_final)
    if self.share_decoder_input_output_embeds:
      # h_final/c_final size: num_layers, bsz, hidden_size
      # We only take the last layer to match shape of decoder inputs
      feed_decoder_input = encoder_out[0][-1] + encoder_out[1][-1] # bsz,␣
↪hidden_size
    else:
      feed_decoder_input = None

    batch_size = src_words.size(1)
    # Create initial decoder input
    initial_words = torch.zeros(1, batch_size, device=device).fill_(self.
↪bos_id).long()
    decoder_input = self.word_embeddings_tgt(initial_words) # 1, bsz,␣
↪embedding_size
```

```
    hidden = encoder_out # initialize decoder hidden state

    decoded = [] # stores partial decoding results
    # Forward one step at a time
    for _ in range(max_tgt_length):
      # Forward decoder for one step
      if self.add_encoder_out_to_decoder_input:
        decoder_input = decoder_input + feed_decoder_input.unsqueeze(0)
      output, hidden = self.decoder_rnn(decoder_input, hidden)
      # Forward final projection
      logits = self.hidden2output(self.dropout_layer(output)).squeeze(0) # bsz,␣
↪vocab
      # Take argmax to find the most probable word
      current_words = logits.argmax(1) # bsz
      # Set next step decoder inputs
      words = current_words.view(1, -1)
      decoder_input = self.word_embeddings_tgt(words)
      # Break if eos is encountered
      if current_words.item() == self.eos_id:
        break
      # Find the tokens
      decoded.append(self.sql.vocab.itos[current_words.item()])
    return ' '.join(decoded)

  def fit(self, train_iter, val_iter, epochs=10, learning_rate=3e-4):
    """Train the model."""
    # Switch the module to training mode
    self.train()
    # Use Adam to optimize the parameters
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_ppl = float('inf')
    best_model = None
    # Run the optimization for multiple epochs
    for epoch in range(epochs):
      total_words = 0
      total_loss = 0.0
      for batch in tqdm(train_iter):
        # Zero the parameter gradients
        self.zero_grad()

        # Input and target
        text, text_lengths = batch.text_reversed # text: max_length_text, bsz
        sql = batch.sql # max_length_sql, bsz
        sql_in = sql[:-1] # Remove <eos> for decode input
        sql_out = sql[1:] # Remove <bos> as target
        batch_size = sql.size(1)
```

```
        # Run forward pass and compute loss along the way.
        logits = self.forward(text, text_lengths, sql_in)
        loss = self.compute_loss(logits, sql_out)

        # Training stats
        num_sql_words = sql_out.ne(self.padding_id_tgt).float().sum().item()
        total_words += num_sql_words
        total_loss += loss.item()

        # Perform backpropagation
        loss.div(batch_size).backward()
        optim.step()

      # Evaluate and track improvements on the validation dataset
      validation_ppl = self.evaluate_ppl(val_iter)
      self.train()
      if validation_ppl < best_validation_ppl:
        best_validation_ppl = validation_ppl
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = total_loss / total_words
      print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
             f'Validation Perplexity: {validation_ppl:.4f}')
```

After implementing the `EncoderDecoder` class, you can use the below script to create the model and kick off training. You are free to tune the hyperparameters.

[ ]:
```
EPOCHS = 10 # epochs, we highly recommend starting with a smaller number like 1
LEARNING_RATE = 3e-4 # learning rate
# Instantiate and train classifier
model = EncoderDecoder(TEXT, SQL,
  embedding_size = 1024,
  hidden_size    = 1024,
  dropout        = 0.1,
  layers         = 3,
  bidirectional  = True,
  share_decoder_input_output_embeds = True,
  add_encoder_out_to_decoder_input = True,
).to(device)

model.fit(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEARNING_RATE)
model.load_state_dict(model.best_model)

# Evaluate model performance, the expected value shall be < 1.3
# We use validation set because this particular test set has a different␣
 ↪distribution
print (f'Validation perplexity: {model.evaluate_ppl(val_iter):.3f}')
```

Evaluation Now we are ready to run the full evaluation. For seq2seq, a proper implementation should reach at least 30% accuracy.

[ ]:
```
questions = []
predictions = []
gold_sqls = []

for example in test_iter.dataset: # val_iter.dataset is just val_data
  # Input and output
  text_reversed_str = example.text_reversed
  question = ' '.join(list(reversed(text_reversed_str))) # detokenized question
  gold_sql = ' '.join(example.sql) # detokenized sql
  questions.append(question)
  gold_sqls.append(gold_sql)
  # Predict
  text, text_lengths = TEXT.process([text_reversed_str])
  text = text.to(device)
  text_lengths = text_lengths.to(device)
  prediction = model.predict(text, text_lengths)
  print (prediction)
  predictions.append(prediction)

evaluate_accuracy(predictions, gold_sqls, questions)
```

2.2 Discussion

2.2.1 Goal 3: Compare the pros and cons of rule-based and neural approaches.

Compare the pros and cons of both approaches with relevant examples from your experiments above. Concerning the accuracy, which approach would you choose to be used in a product? Explain.

Solution For rule-based semantic parsing, as long as the written semantic rules consider all possible cases (which is a nontrivial task), it can solve this task nicely. We list some pros and cons of this approach, but our answer is by no means exhaustive.

Pros * Clearly interpretable. When the system makes a mistake we can easily pinpoint where the problem is, and write more rules to fix it. * Robust. For the cases that we considered, even if at test time there are examples with many constraints, the generated SQL would still be correct. * Low sample complexity. Developing the semantic rules does not need thousands of examples. We are very good at generalization and we only used dozens of examples to write those rules in the solution.

Cons * High develop cost. It is a lot of work to develop those semantic rules. * Poor transferability. For a new domain such as question answering in wikipedia, we need to develop a new set of rules to make this method work.

For the end-to-end seq2seq approach, as long as we have enough data (which is not always the case in reality), enough model capacity (limited by hardware and time), and if the test domain is similar

to the training domain, then the approach would be expeced to work well. Below lists some of its pros and cons.

Pros * High performance. With enough training data, this approach performs well as evidenced by this project. * Low develop cost. Developing the seq2seq model is much easier compared to writing semantic rules and does not require lingustic background. * High transferability if we have training data. For a new domain, as long as we have enough training instances, we can train the same model on the new training set to solve the problem. However, we do want to note that without training anew the model trained on one domain is unlikely to work on another.

Cons * Poor interpretability. When the model makes a mistake, there is no easy way of fixing it. The best we can do is to collect more data similar to the broken ones and add to the training set. * High sample complexity. We need a huge training set to make this approach work. There's no way it'd work using dozens of training examples. * Sensitive. By sensitive we meant if training set only contains compositions up to a certain level, then at test time the trained model is unlikely to work on any instance with a higher number of compositions. If we trained on sentences with length up to 100, then at test time it cannot work on sentences of length 150.

Best approach: If we only care about performance, it is most natural to select the seq2seq approach due to its higher performance. Though depending on the results from precision and recall, it may be best to choose the approach with the best precision scores when applying the approach for customer use (take in the case of Alexa).