project4_semantics

November 3, 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

Semantic parsing is an important task in Natural Language Processing (NLP), where the goal is to convert natural language to its logical form, such as SQL. In the last project, you have built a parsing system to extract parse trees from the questions in the ATIS dataset. However, that only solves an intermediary task, not any end-user task.

In this project, you will go one step further to build a semantic parsing system to convert the questions to SQL queries, such 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 homework you should have a basic understanding of the characteristics of the traditional computational lingustic approach and the recent 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 you to start early.

1.2 Setup

[ ]:
```
!pip install -q dateparser
!pip install -q nltk
!pip install -q cryptography
!pip install -qU torchtext
!pip install -q mysql-connector
```

[ ]:
```
import math
import copy
import requests
import datetime

import torch
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence as pack
import torchtext as tt

from tqdm import tqdm

import dateparser

import nltk
```

```
from nltk.tree import Tree
from nltk import treetransforms

from cryptography.fernet import Fernet

import mysql.connector
from mysql.connector import errorcode

# 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)
```

[ ]:
```
# Tree utils
!wget -nv -N -P scripts https://raw.githubusercontent.com/nlp-course/data/
 ↪master/scripts/trees/tree_utils.py
!wget -nv -N -P scripts https://raw.githubusercontent.com/nlp-course/data/
 ↪master/scripts/trees/tree_utils_private

# Add parse_tree function from the solutions to the last segment
key = '5_pggebiNGJfgNYJOlQiDRGfi1PCZeRuo6vBYDKtza8='
fernet = Fernet(key)
with open('scripts/tree_utils_private', 'rb') as fin:
  with open('scripts/tree_utils.py', 'ab') as fout:
    encrypted_data = fin.read()
    fout.write('\n'.encode())
    fout.write(fernet.decrypt(encrypted_data))

from scripts.tree_utils import parse_tree
```

1.2.1 Load data

In this segment, we only consider `flight_id`-type questions.

[ ]:
```
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/test_flightid.nl
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/test_flightid.sql

!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/dev_flightid.nl
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/dev_flightid.sql
```

```
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/train_flightid.nl
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/train_flightid.sql
```

Let's take a look at the data: the questions are in `.nl` files, and the SQL queries are in `.sql` files. The goal of this project is to convert a question to its corresponding SQL.

[ ]:
```
!head -1 data/dev_flightid.nl
!head -1 data/dev_flightid.sql
```

1.2.2 Data preprocessing

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

[ ]:
```
def reverse(tokens):
  """Reverse a list"""
  return list(reversed(tokens))

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=reverse)
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 by passing in `preprocessing=reverse`. We did that because in seq2seq (w/o attention) this trick improves performance. You can refer to Section 3.3 in the seminal seq2seq paper for more details. Another difference is that we use `batch_first=False`, such that the returned batched tensors would be of size `max_length X batch_size`, which facilitates seq2seq implementation.

Now, we load data using `torchtext`. We use `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, 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)}")

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

print (f"Start of sequence: {SQL.vocab.stoi[SQL.init_token]}") # word id for bos
print (f"End of sequence: {SQL.vocab.stoi[SQL.eos_token]}")   # word id for eos
```

Next, we batch our data to facilitate processing on GPU. Batching is a bit tricky because source/target will be of different lengths. Fortunately, `torchtext` allows us to pass in a `sort_key` function. This will 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(val_iter))
text, text_lengths = batch.text_reversed
print (f"Size of text batch: {text.size()}")
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.size()}")
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 val_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
```

1.2.3 Remote ATIS Database

The output of our systems are SQL queries, but to get the actual answer, we need to execute those queries on a database. We have set up a remote MySQL database, and we will connect to it using mysql-connector later.

1.3 Rule-based Semantic Parsing

First, we will implement a rule-based semantic parser using the parse trees from our last project.

1.3.1 CKY Parsing

We use our parse trees from the previous segment. We provide a function `parse_tree` which returns the parse tree as an `nltk.Tree` object. `parse_tree` is able to parse about 50% of ATIS questions, and it returns `None` if a question is not parsable. For higher coverage, feel free to use your own implementation.

```
[ ]: question = 'flights to boston'
     tree = parse_tree(question)
     tree.pretty_print()
```

1.3.2 Semantic Parsing: The Basics

The high-level idea of rule-based semantic parsing is to associate each grammar rule with a semantic rule. Given a sentence, we first construct its parse tree, then compose 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 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.

1.3.3 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.

Starter Code We provide starter code for some functions that you will implement.

HINT: You may find it useful to use `WHERE TRUE AND (condition)` instead of `WHERE (condition)` in your queries. This way, if you want to add more conditions you can write it as such: `WHERE TRUE AND (condition1) AND (condition2)...`

First, we provide a lexicon from our grammar.

[ ]:
```
#TODO
# Lexicon
lexicon = {
  'ADJ': {
    "days": set(
      [
        "monday's",
        "tuesday's",
        "wednesday's",
        "thursday's",
        "friday's",
        "saturday's",
        "sunday's",
      ]
    ),
    "availability": set(["available", "possible"]),
    "seat_types": set(["first class", "economy", "thrift economy"]),
    "price": set(["cheapest", "lowest cost", "least expensive", "most␣
 ↪expensive"]),
    "time": set(["weekday", "daily", "last", "first"]),
    "attributes": set(["dinner", "transcontinental"]),
  },
  'PDAY': {
    "arrive_on": set(
      [
        "returning on",
        "arriving",
        "arriving on",
        "that arrive on",
        "which arrive on",
      ]
    ),
    "depart_on": set(
      [
        "on",
```

```
      "of",
      "for",
      "next",
      "the next",
      "in the next",
      "of next",
      "leaving",
      "which leave",
      "leaving on",
    ]
  ),
},
'PPLACE': {
  "dest": set(
    [
      "to",
      "that arrive at",
      "that arrives in",
      "coming back to",
      "that go to",
      "and then to",
      "arriving in",
      "and arriving in",
      "and arrive in",
      "to arrive in",
      "arrive in",
      "going to",
      "into",
      "for",
      "with the destination city of",
      "arriving",
      "goes to",
      "flying into",
      "goes on to",
      "reaching",
      "in",
      "and then",
      "arriving to",
    ]
  ),
  "source": set(
    [
      "from",
      "leaving",
      "return from",
      "leaving from",
      "departing from",
```

```
      "are departing from",
      "departing",
      "go from",
      "going from",
      "back from",
      "that goes from",
      "that departs",
      "which leaves from",
      "which leave",
      "that leave",
      "originating in",
      "leave",
      "out of",
      "leaves from",
      "to get from"
    ]
  ),
  "through": set(
    [
      "via",
      "with a stopover in",
      "with a layover in",
      "with a stopover at",
      "and a stopover in",
      "stop in",
      "stopping in",
      "make a stop in",
      "with a stop in",
      "with one stop in",
      "go through",
      "which go through",
      "makes a stopover in",
      "that stops in",
      "that stops over in",
      "by way of",
      "connecting through",
      "that will stop in",
      "which connects in",
    ]
  ),
},
'PTIME': {
  "arrive_by": set(
    [
      "that arrive before",
      "that arrives before",
      "arriving before",
```

```
      "arrival by",
      "arrives",
      "before",
      "departing before",
      "that leaves before",
      "which arrive before",
      "by",
    ]
  ),
  "arrive_at": set(
    [
      "around",
      "that return around",
      "that gets in around",
      "at",
      "arriving around",
      "arriving about",
    ]
  ),
  "arrive_after": set(["that arrive soon after", "arriving after"]),
  "depart_at": set(
    [
      "leaving at",
      "leaving",
      "which leave after",
      "leaving after",
      "after",
      "departing after",
      "that depart after",
      "departing at",
      "are departing at",
    ]
  ),
  "depart_in": set(["in", "departing in", "on", "that leaves in"]),
},
'TIME': {
  "morning": set(
    [
      "the morning",
      "the early am",
      "mornings",
      "as early as possible",
      "earliest possible time",
      "as soon thereafter as possible",
    ]
  ),
  "afternoon": set(
```

```
      ["the afternoon", "the late afternoon", "the day", "afternoons"]
    ),
    "evening": set(["the evening", "evenings"]),
  },
}
```

In addition to the provided lexicon, we also provide some helper functions. You will need to implement `eval_S`, which returns the SQL query based on a parse tree.

[ ]:
```
#TODO
def eval_S(tree):
  """
  Construct the SQL query based on a parse tree.
  Arguments:
      tree: an nltk.Tree.
  Returns:
      a string of the corresponding SQL query
  """
  #TODO: implement this method.
  PREJ = None
  DET = None
  ADJS = None
  FLIGHT = None
  PPS = None

  for child in tree:
    if child.label() == "PREJ":
      PREJ = child
    elif child.label() == "DET":
      DET = child
    elif child.label() == "ADJS":
      ADJS = child
    elif child.label() == "FLIGHT":
      FLIGHT = child
    elif child.label() == "PPS":
      PPS = child

  ### Implement these Rules
  # S -> (PREJ) (DET) ADJS FLIGHT PPS
  # S -> (PREJ) (DET) ADJS FLIGHT
  # S -> (PREJ) (DET) FLIGHT PPS
  # S -> (PREJ) (DET) FLIGHT
  ### YOUR CODE HERE
  raise NotImplementedError

def eval_FLIGHT(tree):
  ### Implement these Rules
```

```
  # FLIGHT -> 'flights' | 'flight' | 'to' 'fly'
  ### YOUR CODE HERE
  raise NotImplementedError

def eval_PPS(tree):
  ### Implement these Rules
  # PPS -> PP
  # PPS -> PP PPS
  ### YOUR CODE HERE
  raise NotImplementedError

def eval_PP(tree):
  # List of the labels of the children (e.g. ['PPLACE', 'PLACE'])
  child_labels = [child.label() for child in tree]
  ### Implement these Rules
  # PP -> PPLACE PLACE OR PLACE
  # PP -> PPLACE EITHER PLACE OR PLACE
  # PP -> PPLACE PLACE
  # PP -> BETWEEN PLACE AND PLACE
  ### YOUR CODE HERE
  raise NotImplementedError

def eval_PPLACE(tree):
  PPLACE_lexicon = lexicon['PPLACE']
  # Join multiword phrases
  val = ' '.join(tree).strip()
  ### Implement these Rules
  # PPLACE -> <departing>
  # PPLACE -> <arriving>
  # PPLACE -> <layover>
  ### YOUR CODE HERE
  raise NotImplementedError

def eval_PLACE(tree):
  # Join multiword phrases
  val = ' '.join(tree)
  ### Implement these Rules
  # PLACE -> <city_name>
  ### YOUR CODE HERE
  raise NotImplementedError
```

Solution

[ ]:
```
#Solution
# Lexicon
lexicon = {
  'ADJ': {
```

```
    "days": set(
      [
        "monday's",
        "tuesday's",
        "wednesday's",
        "thursday's",
        "friday's",
        "saturday's",
        "sunday's",
      ]
    ),
    "availability": set(["available", "possible"]),
    "seat_types": set(["first class", "economy", "thrift economy"]),
    "price": set(["cheapest", "lowest cost", "least expensive", "most␣
↪expensive"]),
    "time": set(["weekday", "daily", "last", "first"]),
    "attributes": set(["dinner", "transcontinental"]),
  },
  'PDAY': {
    "arrive_on": set(
      [
        "returning on",
        "arriving",
        "arriving on",
        "that arrive on",
        "which arrive on",
      ]
    ),
    "depart_on": set(
      [
        "on",
        "of",
        "for",
        "next",
        "the next",
        "in the next",
        "of next",
        "leaving",
        "which leave",
        "leaving on",
      ]
    ),
  },
  'PPLACE': {
    "dest": set(
      [
        "to",
```

```
        "that arrive at",
        "that arrives in",
        "coming back to",
        "that go to",
        "and then to",
        "arriving in",
        "and arriving in",
        "and arrive in",
        "to arrive in",
        "arrive in",
        "going to",
        "into",
        "for",
        "with the destination city of",
        "arriving",
        "goes to",
        "flying into",
        "goes on to",
        "reaching",
        "in",
        "and then",
        "arriving to",
      ]
    ),
    "source": set(
      [
        "from",
        "leaving",
        "return from",
        "leaving from",
        "departing from",
        "are departing from",
        "departing",
        "go from",
        "going from",
        "back from",
        "that goes from",
        "that departs",
        "which leaves from",
        "which leave",
        "that leave",
        "originating in",
        "leave",
        "out of",
        "leaves from",
        "to get from"
      ]
```

```
    ),
    "through": set(
      [
        "via",
        "with a stopover in",
        "with a layover in",
        "with a stopover at",
        "and a stopover in",
        "stop in",
        "stopping in",
        "make a stop in",
        "with a stop in",
        "with one stop in",
        "go through",
        "which go through",
        "makes a stopover in",
        "that stops in",
        "that stops over in",
        "by way of",
        "connecting through",
        "that will stop in",
        "which connects in",
      ]
    ),
  },
  'PTIME': {
    "arrive_by": set(
      [
        "that arrive before",
        "that arrives before",
        "arriving before",
        "arrival by",
        "arrives",
        "before",
        "departing before",
        "that leaves before",
        "which arrive before",
        "by",
      ]
    ),
    "arrive_at": set(
      [
        "around",
        "that return around",
        "that gets in around",
        "at",
        "arriving around",
```

```
        "arriving about",
      ]
    ),
    "arrive_after": set(["that arrive soon after", "arriving after"]),
    "depart_at": set(
      [
        "leaving at",
        "leaving",
        "which leave after",
        "leaving after",
        "after",
        "departing after",
        "that depart after",
        "departing at",
        "are departing at",
      ]
    ),
    "depart_in": set(["in", "departing in", "on", "that leaves in"]),
  },
  'TIME': {
    "morning": set(
      [
        "the morning",
        "the early am",
        "mornings",
        "as early as possible",
        "earliest possible time",
        "as soon thereafter as possible",
      ]
    ),
    "afternoon": set(
      ["the afternoon", "the late afternoon", "the day", "afternoons"]
    ),
    "evening": set(["the evening", "evenings"]),
  },
}
```

[ ]:
```
#Solution
def eval_S(tree):
  """
  Construct the SQL query based on a parse tree.
  Arguments:
      tree: an nltk.Tree.
  Returns:
      a string of the corresponding SQL query
  """
  PREJ = None
```

```
  DET = None
  ADJS = None
  FLIGHT = None
  PPS = None

  for child in tree:
    if child.label() == "PREJ":
      PREJ = child
    elif child.label() == "DET":
      DET = child
    elif child.label() == "ADJS":
      ADJS = child
    elif child.label() == "FLIGHT":
      FLIGHT = child
    elif child.label() == "PPS":
      PPS = child

  # S -> (PREJ) (DET) ADJS FLIGHT PPS
  if ADJS and PPS:
    return "{} {} {}".format(eval_FLIGHT(FLIGHT), eval_ADJS(ADJS), eval_PPS(PPS))
  # S -> (PREJ) (DET) ADJS FLIGHT
  elif ADJS:
    return "{} {}".format(eval_FLIGHT(FLIGHT), eval_ADJS(ADJS))
  # S -> (PREJ) (DET) FLIGHT PPS
  elif PPS:
    return "{} {}".format(eval_FLIGHT(FLIGHT), eval_PPS(PPS))
  # S -> (PREJ) (DET) FLIGHT
  else:
    return eval_FLIGHT(FLIGHT)

def eval_FLIGHT(tree):
  # FLIGHT -> 'flights' | 'flight' | 'to' 'fly'
  return 'SELECT DISTINCT flight.flight_id FROM flight WHERE TRUE'

def eval_PREJ(tree):
  # PREJ -> JUNK PREJ | JUNK
  # Not relevant for semantics
  return ''

def eval_ADJS(tree):
  # ADJS -> ADJ
  if len(tree) == 1:
    return eval_ADJ(tree[0])
  # ADJS -> ADJ ADJS
  else:
    return "{} {}".format(eval_ADJ(tree[0]), eval_ADJS(tree[1]))
```

```
def eval_ADJ(tree):
  # ADJ -> <anything>
  return ''

def eval_DET(tree):
  # DET -> 'all' 'the' | 'all' | A | 'an' | THE | 'any' | 'all' 'of' 'the'
  # All words in lexicon should show all flights
  # Might consider `THE` or `A` to only get one...
  return ''

def eval_PPS(tree):
  # PPS -> PP
  if len(tree) == 1:
    return 'AND {}'.format(eval_PP(tree[0]))
  # PPS -> PP PPS
  else:
    return "AND {} {}".format(eval_PP(tree[0]), eval_PPS(tree[1]))

def eval_PP(tree):
  child_labels = [child.label() for child in tree]
  # PP -> PPLACE PLACE OR PLACE
  if child_labels == ['PPLACE', 'PLACE', 'OR', 'PLACE']:
    return '({} OR {})'.format(eval_PPLACE(tree[0])(eval_PLACE(tree[1])),␣
↪eval_PPLACE(tree[0])(eval_PLACE(tree[3])))
  # PP -> PPLACE EITHER PLACE OR PLACE
  elif child_labels == ['PPLACE', 'EITHER', 'PLACE', 'OR', 'PLACE']:
    return '({} OR {})'.format(eval_PPLACE(tree[0])(eval_PLACE(tree[2])),␣
↪eval_PPLACE(tree[0])(eval_PLACE(tree[4])))
  # PP -> PPLACE PLACE
  elif child_labels == ['PPLACE', 'PLACE']:
    return eval_PPLACE(tree[0])(eval_PLACE(tree[1]))
  # PP -> BETWEEN PLACE AND PLACE
  elif child_labels == ['BETWEEN', 'PLACE', 'AND', 'PLACE']:
    return '(flight.from_airport IN {}) AND (flight.to_airport IN {})'.
↪format(eval_PLACE(tree[1]), eval_PLACE(tree[3]))
  # PP -> BETWEEN TIME AND TIME
  elif child_labels == ['BETWEEN', 'TIME', 'AND', 'TIME']:
    return '(flight.departure_time >= {} AND flight.departure_time <= {})'.
↪format(eval_TIME(tree[1]), eval_TIME(tree[3]))
  # PP -> WEEKDAY
  elif child_labels == ['WEEKDAY']:
    return eval_WEEKDAY(tree[0])
  # PP -> TIME
  elif child_labels == ['TIME']:
    return eval_TIME(tree[0])
  # PP -> PTIME TIME
  elif child_labels == ['PTIME', 'TIME']:
```

```
    return eval_PTIME(tree[0])(eval_TIME(tree[1]))
  # PP -> EITHER PLACE OR PLACE | PLACE OR PLACE | PLACE | PDAY WEEKDAY |␣
↪WEEKDAY TIME | PDAY WEEKDAY TIME | PDAY DATE | DATE | PAIRLINE AIRLINE |␣
↪AIRCRAFT | FLIGHTTYPE | FARETYPE | PRICE | FOOD | AVAIL | POSTJ
  else:
    return 'TRUE'

def eval_PPLACE(tree):
  lex = lexicon['PPLACE']
  val = ' '.join(tree).strip()
  if val in lex['dest']:
    return lambda dest_sql: '(flight.to_airport IN {})'.format(dest_sql)
  elif val in lex['source']:
    return lambda source_sql: '(flight.from_airport IN {})'.format(source_sql)
  elif val in lex['through']:
    return lambda through_sql: '(flight_stop.stop_airport IN {})'.
↪format(through_sql)
  else:
    return lambda place: 'TRUE'

def eval_PLACE(tree):
  val = ' '.join(tree)
  # Currently assumes PLACE is a city... may want to query for it and check and␣
↪handle other cases
  return "(SELECT airport_service.airport_code FROM airport_service WHERE␣
↪airport_service.city_code IN (SELECT city.city_code FROM city WHERE city.
↪city_name = '{}'))".format(val.upper())

def eval_PTIME(tree):
  lex = lexicon['PTIME']
  val = ' '.join(tree)
  if val in lex['arrive_by']:
    return lambda time: '(flight.arrival_time <= {})'.format(int(time.
↪strftime('%H%M')))
  elif val in lex['arrive_at']:
    return lambda time: '(flight.arrival_time >= {} AND flight.arrival_time <=␣
↪{})'.format(int((time - datetime.timedelta(minutes=30)).strftime('%H%M')),␣
↪int((time + datetime.timedelta(minutes=30)).strftime('%H%M')))
  elif val in lex['arrive_after']:
    return lambda time: '(flight.arrival_time >= {})'.format(int(time.
↪strftime('%H%M')))
  elif val in lex['depart_at']:
    return lambda time: '(flight.departure_time >= {} AND flight.departure_time␣
↪<= {})'.format(int((time - datetime.timedelta(minutes=30)).strftime('%H%M')),␣
↪int((time + datetime.timedelta(minutes=30)).strftime('%H%M')))
  # Try to handle "depart in" by querying time
```

```
  else:
    return lambda time: ''

def eval_TIME(tree):
  child_labels = [child.label() for child in tree]
  if child_labels == ['SIMPLETIME']:
    return eval_SIMPLETIME(tree[0])
  else:
    return 'TRUE'

def eval_SIMPLETIME(tree):
  # uses natural langage parser for lexicon
  val = ' '.join(tree)
  try:
    time = dateparser.parse(val)
    return time
  except:
    return 'TRUE'

def eval_WEEKDAY(tree):
  child_labels = [child.label() for child in tree]
  if child_labels == ['SIMPLEWEEKDAY'] or child_labels == ['A', 'SIMPLEWEEKDAY']:
    return eval_SIMPLEWEEKDAY(tree[-1])
  else:
    return 'TRUE'

def eval_SIMPLEWEEKDAY(tree):
  val = ' '.join(tree)
  return "(flight.flight_days IN (SELECT days.days_code FROM days WHERE days.
 ↪day_name = '{}'))".format(val)
```

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 remote database
  try:
    conn = mysql.connector.connect(host='54.202.209.190', user='CS187',␣
↪password='007')
  except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
      print("Something is wrong with your user name or password")
    else:
      print(err)

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

  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.

[ ]:
```
# 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_S(tree)
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_S(tree)
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_S(tree)
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_S(tree)
print (f"Accuracy: {evaluate_accuracy(predicted_sql, gold_sql, question)}")
```

Below is the full evaluation code. Note that you are required to get correct results on **at least 25%** of flight_id type questions from the test set.

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

```
gold_sqls = []

for example in 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_S(tree)
  except Exception as e:
    predictions.append('')
    continue
  predictions.append(predicted_sql)

evaluate_accuracy(predictions, gold_sqls, questions)
```

1.4 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.

1.4.1 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)
```

1.5 Discussion

1.5.1 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).