CS187 Project Segment 2: Sequence labeling – The slot filling task

September 2, 2021

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

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

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

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/project2.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 2: Sequence labeling – The slot filling task

The second segment of the project involves a sequence labeling task, in which the goal is to label the tokens in a text. Many NLP tasks have this general form. Most famously is the task of *part-of-speech labeling* as you explored in lab 2-4, where the tokens in a text are to be labeled with their part of speech (noun, verb, preposition, etc.). In this project segment, however, you'll use sequence labeling to implement a system for filling the slots in a template that is intended to describe the meaning of an ATIS query. For instance, the sentence

```
What's the earliest arriving flight between Boston and Washington DC?
```

might be associated with the following slot-filled template:

```
flight_id
    fromloc.cityname: boston
    toloc.cityname: washington
    toloc.state: dc
    flight_mod: earliest arriving
```

You may wonder how this task is a sequence labeling task. We label each word in the source sentence with a tag taken from a set of tags that correspond to the slot-labels. For each slot-label, say `flight_mod`, there are two tags: `B-flight_mod` and `I-flight_mod`. These are used to mark the beginning (B) or interior (I) of a phrase that fills the given slot. In addition, there is a tag for other (O) words that are not used to fill any slot. (This technique is thus known as IOB encoding.) Thus the sample sentence would be labeled as follows:

Token	Label
BOS	O
what's	O
the	O
earliest	B-flight_mod
arriving	I-flight_mod
flight	O
between	O
boston	B-fromloc.city_name
and	O
washington	B-toloc.city_name
dc	B-toloc.state_code
EOS	O

See below for information about the `BOS` and `EOS` tokens.

The template itself is associated with the question type for the sentence, perhaps as recovered from the sentence in the last project segment.

In this segment, you'll implement two methods for sequence labeling: a hidden Markov model (HMM) and two recurrent neural networks, a simple RNN and a long short-term memory network (LSTM). By the end of this homework, you should have grasped the pros and cons of both approaches.

1.1 Goals

1. Implement an HMM-based approach to sequence labeling.
2. Implement an RNN-based approach to sequence labeling.
3. Implement an LSTM-based approach to sequence labeling.
4. (Optional) Compare the performances of HMM and RNN/LSTM under different amount of training data. Discuss the pros and cons of the HMM approach and the neural approach.

1.2 Setup

[ ]:
```
import copy
import math
import matplotlib.pyplot as plt
import random
import warnings

import torch
import torch.nn as nn
import torchtext.legacy as tt

from tqdm import tqdm

# Set random seeds
seed = 1234
random.seed(seed)
torch.manual_seed(seed)

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

1.2.1 Load Data

First, we download the ATIS dataset, already presplit into training, validation (dev), and test sets.

[ ]:
```
shell("""
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/atis.train.txt
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/atis.dev.txt
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/atis.test.txt
""")
```

1.2.2 Data preprocessing

We again use `torchtext` to load data and convert words to indices in the vocabulary. We use one field `TEXT` for processing the question, and another field `TAG` for processing the sequence labels.

We treat words occurring fewer than three times in the training data as *unknown words*. They'll be replaced by the unknown word type `<unk>`.

[ ]:
```
MIN_FREQ = 3

TEXT = tt.data.Field(init_token="<bos>")
TAG = tt.data.Field(init_token="<bos>")
fields=(('text', TEXT), ('tag', TAG))

train, val, test = tt.datasets.SequenceTaggingDataset.splits(
  fields=fields,
  path='./data/',
  train='atis.train.txt',
  validation='atis.dev.txt',
  test='atis.test.txt'
)

TEXT.build_vocab(train.text, min_freq=MIN_FREQ)
TAG.build_vocab(train.tag)
```

We can get some sense of the datasets by looking at the size and some elements of the text and tag vocabularies.

[ ]:
```
print(f"Size of English vocabulary: {len(TEXT.vocab)}")
print(f"Most common English words: {TEXT.vocab.freqs.most_common(10)}\n")

print(f"Number of tags: {len(TAG.vocab)}")
print(f"Most common tags: {TAG.vocab.freqs.most_common(10)}")
```

1.2.3 Special tokens and tags

You'll have already noticed the `BOS` and `EOS`, special tokens that the dataset developers used to indicate the beginning and end of the sentence; we'll leave them in the data.

We've also passed in `init_token="<bos>"` for both torchtext fields. Torchtext will prepend these to the sequence of words and tags. This relieves us from estimating the initial distribution of tags and tokens in HMMs, since we always start with a token `<bos>` whose tag is also `<bos>`. We'll be able to refer to these tags as exemplified here:

[ ]:
```
print(f"""
Initial tag string: {TAG.init_token}
Initial tag id:     {TAG.vocab.stoi[TAG.init_token]}
""")
```

Finally, since torchtext will be providing the sentences in the training corpus in "batches", torchtext will force the sentences within a batch to be the same length by padding them with a special token. Again, we can access that token as shown here:

```
[ ]: print(f"""
     Pad tag string: {TAG.pad_token}
     Pad tag id:     {TAG.vocab.stoi[TAG.pad_token]}
     """)
```

Now, we can iterate over the dataset using `torchtext`'s iterator. We'll use a non-trivial batch size to gain the benefit of training on multiple sentences at a shot. This is different from how you've previously used torch, with a batch size of 1, and you'll need to be careful about the shapes of the various tensors that are being manipulated.

```
[ ]: BATCH_SIZE = 20

     train_iter, val_iter, test_iter = tt.data.BucketIterator.splits(
         (train, val, test),
         batch_size=BATCH_SIZE,
         repeat=False,
         device=device)
```

Each batch will be a tensor of size `max_length x batch_size`. Let's examine a batch.

```
[ ]: # Get the first batch
     batch = next(iter(train_iter))

     # What's its shape? Should be max_length x batch_size.
     print(f'Shape of batch text tensor: {batch.text.shape}\n')

     # Extract the first sentence in the batch, both text and tags
     first_sentence = batch.text[:, 0]
     first_tags = batch.tag[:, 0]

     # Print out the first sentence, as token ids and as text
     print("First sentence in batch")
     print(f"{first_sentence}")
     print(f"{' '.join([TEXT.vocab.itos[i] for i in first_sentence])}\n")

     print("First tags in batch")
     print(f"{first_tags}")
     print(f"{[TAG.vocab.itos[i] for i in first_tags]}")
```

The goal of this project is to predict the sequence of tags `batch.tag` given a sequence of words `batch.text`.

1.3 Majority class labeling

As usual, we can get a sense of the difficulty of the task by looking at a simple baseline, tagging every token with the majority tag. Here's a table of tag frequencies for the most frequent tags:

```
[ ]: tag_counts = torch.zeros(len(TAG.vocab.itos), device=device)

for batch in train_iter:                # for each batch
  for sent_id in range(len(batch)):     # ... each sentence in the batch
    for tag in batch.tag[:, sent_id]:   # ... each tag in the sentence
      tag_counts[tag] += 1              # bump the tag count

for tag_id in range(len(TAG.vocab.itos)):
  print(f'{tag_id:3}  {TAG.vocab.itos[tag_id]:30}{tag_counts[tag_id].item():3.
 ↪0f}')
```

It looks like the `'O'` (other) tag is, unsurprisingly, the most frequent tag (except for the padding tag). The proportion of tokens labeled with that tag (ignoring the padding tag) gives us a good baseline accuracy for this sequence labeling task. If we were to just label every token with the `'O'` tag, we'd expect to be right on this proportion of the tokens.

```
[ ]: majority_baseline_accuracy = (
  tag_counts[TAG.vocab.stoi['O']]
  / (sum(tag_counts)
     - tag_counts[TAG.vocab.stoi[TAG.pad_token]])
)
print(f'Baseline accuracy: {majority_baseline_accuracy:.3f}')
```

1.4 HMM for sequence labeling

Having established the baseline to beat, we turn to implementing an HMM model.

1.4.1 Notation

First, let's start with some notation. We use $Q = \langle Q_1, Q_2, \ldots, Q_N \rangle$ to denote the possible tags, which is the state space of the HMM, and $\mathcal{V} = \langle v_1, v_2, \ldots v_V \rangle$ to denote the vocabulary of word types. We use $q_t \in Q$ to denote the state at time step t (where t varies from 1 to T), and $o_t \in \mathcal{V}$ to denote the observation (word) at time step t.

1.4.2 Training an HMM by counting

Recall that an HMM is defined via a transition matrix A which stores the probability of moving from one state Q_i to another Q_j, that is,

$$A_{ij} = P(q_{t+1} = Q_j \,|\, q_t = Q_i)$$

and an emission matrix B which stores the probability of generating word $\mathcal{V}_j$ given state Q_i, that is,

$$B_{ij} = P(o_t = \mathcal{V}_j \,|\, q_t = Q_i)$$

As is typical in notating probabilities, we'll use abbreviations

$$P(q_{t+1} \mid q_t) \equiv P(q_{t+1} = Q_j \mid q_t = Q_i) \tag{1}$$
$$P(o_t \mid q_t) \equiv P(o_t = \mathcal{V}_j \mid q_t = Q_i) \tag{2}$$

where the i and j are clear from context.

In our case, since the labels are observed in the training data, we can directly use counting to determine (maximum likelihood) estimates of A and B.

Goal 1(a): Find the transition matrix The matrix A contains the transition probabilities: A_{ij} is the probability of moving from state Q_i to state Q_j in the training data, so that $\sum_{j=1}^{N} A_{ij} = 1$ for all i.

We find these probabilities by counting the number of times state Q_j appears right after state Q_i, as a proportion of all of the transitions from Q_i.

$$A_{ij} = \frac{\sharp(Q_i, Q_j) + \delta}{\sharp(Q_i) + \delta N}$$

(In the above formula, we also used add-δ smoothing.)

Using the above definition, implement the method `train_A` in the `HMM` class, which calculates and returns the A matrix as a tensor of size $N \times N$.

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.
>
> Remember that the training data is being delivered to you batched.

Goal 1(b): Find the emission matrix B Similar to the transition matrix, the emission matrix contains the emission probabilities such that B_{ij} is probability of word $o_t = \mathcal{V}_j$ conditioned on state $q_t = Q_i$.

We can find this by counting as well.

$$B_{ij} = \frac{\sharp(Q_i, \mathcal{V}_j) + \delta}{\sharp(Q_i) + \delta V}$$

Using the above definitions, implement the `train_B` method in the `HMM` class, which calculates and returns the B matrix as a tensor of size $N \times V$.

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.

1.4.3 Sequence labeling with a trained HMM

Now that you're able to train an HMM by estimating the transition matrix A and the emission matrix B, you can apply it to the task of sequence labeling. Our goal is to find the most probable sequence of tags $\hat{q} \in Q^T$ given a sequence of words $o \in \mathcal{V}^T$.

$$\begin{aligned}\hat{q} &= \underset{q \in Q^T}{\text{argmax}}(P(q \mid o)) \\ &= \underset{q \in Q^T}{\text{argmax}}(P(q, o)) \\ &= \underset{q \in Q^T}{\text{argmax}} \left(\Pi_{t=1}^{T} P(o_{t+1} \mid q_{t+1}) P(q_{t+1} \mid q_t)\right)\end{aligned}$$

where $P(o_{t+1} = \mathcal{V}_j \mid q_{t+1} = Q_i) = B_{ij}$, $P(q_{t+1} = Q_j \mid q_t = Q_i) = A_{ij}$.

Goal 1(c): Viterbi algorithm Implement the `predict` method, which should use the Viterbi algorithm to find the most likely sequence of tags for a sequence of `words`.

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.
>
> Warning: It may take up to 30 minutes to tag the entire test set depending on your implementation (a fully tensorized implementation can be much faster tho). We highly recommend that you begin by experimenting with your code using a *very small subset* of the dataset, say two or three sentences, ramping up from there.
>
> Hint: Consider how to use vectorized computations where possible for speed.

1.4.4 Evaluation

We've provided you with the `evaluate` function, which takes a dataset iterator and uses `predict` on each sentence in each batch, comparing against the gold tags, to determine the accuracy of the model on the test set.

[ ]:
```
class HMMTagger():
  def __init__ (self, text, tag):
    self.text = text
    self.tag = tag
    self.V = len(text.vocab.itos)     # vocabulary size
    self.N = len(tag.vocab.itos)      # state space size
    self.initial_state_id = tag.vocab.stoi[tag.init_token]
    self.pad_state_id = tag.vocab.stoi[tag.pad_token]
    self.pad_word_id = text.vocab.stoi[text.pad_token]

  def train_A(self, iterator, delta):
    """Stores A for training dataset `iterator` using add-`delta` smoothing."""
    # Create A table
    A = torch.zeros(self.N, self.N, device=device)

    #TODO: Add your solution from Goal 1(a) here.
    #      The returned value should be a tensor for the A matrix
    #      of size N x N.

    ...
```

```
    return A

  def train_B(self, iterator, delta):
    """Stores B for training dataset `iterator` using add-`delta` smoothing."""
    # Create B
    B = torch.zeros(self.N, self.V, device=device)

    #TODO: Add your solution from Goal 1 (b) here.
    #      The returned value should be a tensor for the $B$ matrix
    #      of size N x V.

    ...

    return B

  def train_all(self, iterator, delta=0.01):
    """Stores A and B for training dataset `iterator`."""
    self.log_A = self.train_A(iterator, delta).log()
    self.log_B = self.train_B(iterator, delta).log()

  def predict(self, words):
    """Returns the most likely sequence of tags for a sequence of `words`.
    Arguments:
      words: a tensor of size (seq_len,)
    Returns:
      a list of tag ids
    """
    #TODO: Add your solution from Goal 1 (c) here.
    #      The returned value should be a list of tag ids.

    ...

    return bestpath

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      for sent_id in range(len(batch)):
        words = batch.text[:, sent_id]
        words = words[words.ne(self.pad_word_id)] # remove paddings
        tags_gold = batch.tag[:, sent_id]
        tags_pred = self.predict(words)
        for tag_gold, tag_pred in zip(tags_gold, tags_pred):
          if tag_gold == self.pad_state_id:  # stop once we hit padding
            break
```

```
        else:
          total += 1
          if tag_pred == tag_gold:
            correct += 1
    return correct/total
```

Putting everything together, you can expect a correct implementation to reach above **90% test set accuracy** after running the following cell.

[ ]:
```
# Instantiate and train classifier
hmm_tagger = HMMTagger(TEXT, TAG)
hmm_tagger.train_all(train_iter)

# Evaluate model performance
print(f'Training accuracy: {hmm_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {hmm_tagger.evaluate(test_iter):.3f}')
```

1.4.5 RNN for Sequence Labeling

HMMs work quite well for this sequence labeling task. Now let's take an alternative (and more trendy) approach: RNN/LSTM-based sequence labeling. Similar to the HMM part of this project, you will also need to train a model on the training data, and then use the trained model to decode and evaluate some testing data.

After unfolding an RNN, the cell at time t generates the observed output o_t based on the input x_t and the hidden state of the previous cell h_{t-1}, according to the following equations.

$$h_t = \sigma(Ux_t + Vh_{t-1})$$
$$o_t = \text{softmax}(Wh_t)$$

The parameters here are the elements of the matrices U, V, and W, and the bias terms b_h and b_y which we omitted for simplicity. Similar to the last project segment, we will perform the forward computation, calculate the loss, and then perform the backward computation to compute the gradients with respect to these model parameters. Finally, we will adjust the parameters opposite the direction of the gradients to minimize the loss, repeating until convergence.

You've seen these kinds of neural network models before, for language modeling in lab 2-3 and sequence labeling in lab 2-5. The code there should be very helpful in implementing an `RNNTagger` class below. Consequently, we've provided very little guidance on the implementation. We do recommend you follow the steps below however.

Goal 2(a): RNN training Implement the forward pass of the RNN tagger and the loss function. A reasonable way to proceed is to implement the following methods:

1. `forward(self, text_batch, hidden_0)`: Performs the RNN forward computation over a whole `text_batch` (`batch.text` in the above data loading example) starting with the starting hidden state `hidden_0` of size, say, `1 x batch_size x hidden_size`. The `text_batch` will

be of shape `max_length x batch_size`. You might run it through the following layers: an embedding layer, which maps each token index to an embedding of size `embedding_size` (so that the size of the mapped batch becomes `max_length x batch_size x embedding_size`); then an RNN, which maps each token embedding to a vector of `hidden_size` (the size of all outputs is `max_length x batch_size x hidden_size`); then a linear layer, which maps each RNN output element to a vector of size N (which is commonly referred to as "logits", recall that $N = |Q|$, the size of the tag set).

This function is expected to return `logits`, which provides a logit for each tag of each word of each sentence in the batch (which is a tensor of size `max_length x batch_size x N`).

You might find the following functions useful:

- `nn.Embedding`
- `nn.Linear`
- `nn.RNN`

2. `compute_loss(self, logits, ground_truth)`: Computes the loss for a batch by comparing `logits` of a batch returned by `forward` to `ground_truth`, which stores the true tag ids for the batch. Thus `logits` is a tensor of size `max_length x batch_size x N`, and `ground_truth` is a tensor of size `max_length x batch_size`. Note that the criterion functions in `torch` expect outputs of a certain shape, so you might need to perform some shape conversions.

   You might find `nn.CrossEntropyLoss` from the last project segment useful. Note that if you use `nn.CrossEntropyLoss` then you should not use a softmax layer at the end since that's already absorbed into the loss function. Alternatively, you can use `nn.LogSoftmax` as the final sublayer in the forward pass, but then you need to use `nn.NLLLoss`, which does not contain its own softmax. We recommend the former, since working in log space is usually more numerically stable. For reshaping tensors, check out the `torch.Tensor.view` method.

3. `train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001)`: Trains the model on training data generated by the iterator `train_iter` and validation data `val_iter`.The `epochs` and `learning_rate` variables are the number of epochs (number of times to run through the training data) to run for and the learning rate for the optimizer, respectively. You can use the validation data to determine which model was the best one as the epocks go by. Notice that our code below assumes that during training the best model is stored so that `rnn_tagger.load_state_dict(rnn_tagger.best_model)` restores the parameters of the best model.

Goal 2(b) RNN decoding Implement a method to predict the tag sequence associated with a sequence of words:

1. `predict(self, text_batch)`: Returns the batched predicted tag sequences associated with a batch of sentences.
2. `def evaluate(self, iterator)`: Returns the accuracy of the trained tagger on a dataset provided by `iterator`.

[ ]:
```
class RNNTagger(nn.Module):

  ...
```

Now train your tagger on the training and validation set. Run the below cell to train an RNN, and evaluate it. A proper implementation should reach about **95%+ accuracy**.

[ ]:
```
# Instantiate and train classifier
rnn_tagger = RNNTagger(TEXT, TAG, embedding_size=36, hidden_size=36).to(device)
rnn_tagger.train_all(train_iter, val_iter, epochs=10, learning_rate=0.001)
rnn_tagger.load_state_dict(rnn_tagger.best_model)

# Evaluate model performance
print(f'Training accuracy: {rnn_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {rnn_tagger.evaluate(test_iter):.3f}')
```

1.5 LSTM for slot filling

Did your RNN perform better than HMM? How much better was it? Was that expected?

RNNs tend to exhibit the vanishing gradient problem. To remedy this, the Long-Short Term Memory (LSTM) model was introduced. In PyTorch, we can simply use `nn.LSTM`.

In this section, you'll implement an LSTM model for slot filling. If you've got the RNN model well implemented, this should be extremely straightforward. Just copy and paste your solution, change the call to `nn.RNN` to a call to `nn.LSTM`, and make any other minor adjustments that are necessary. In particular, LSTMs have *two* recurrent bits, `h` and `c`. You'll thus need to initialize both of these when performing forward computations.

[ ]:
```
class LSTMTagger(nn.Module):


  ...
```

Run the cell below to train an LSTM, and evaluate it. A proper implementation should reach about **95%+ accuracy**.

[ ]:
```
# Instantiate and train classifier
lstm_tagger = LSTMTagger(TEXT, TAG, embedding_size=36, hidden_size=36).
 ↪to(device)
lstm_tagger.train_all(train_iter, val_iter, epochs=10, learning_rate=0.001)
lstm_tagger.load_state_dict(lstm_tagger.best_model)

# Evaluate model performance
print(f'Training accuracy: {lstm_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {lstm_tagger.evaluate(test_iter):.3f}')
```

1.6 (Optional) Goal 4: Compare HMM to RNN/LSTM with different amounts of training data

Vary the amount of training data and compare the performance of HMM to RNN or LSTM (Since RNN is similar to LSTM, picking one of them is enough.) Discuss the pros and cons of HMM and RNN/LSTM based on your experiments.

This part is more open-ended. We're looking for thoughtful experiments and analysis of the results, not any particular result or conclusion.

The below code shows how to subsample the training set with downsample ratio `ratio`. To speedup evaluation we only use 50 test samples.

[ ]:
```
ratio = 0.1
test_size = 50

# Set random seeds to make sure subsampling is the same for HMM and RNN
random.seed(seed)
torch.manual_seed(seed)

train, val, test = tt.datasets.SequenceTaggingDataset.splits(
            fields=fields,
            path='./data/',
            train='atis.train.txt',
            validation='atis.dev.txt',
            test='atis.test.txt')

# Subsample
random.shuffle(train.examples)
train.examples = train.examples[:int(math.floor(len(train.examples)*ratio))]
random.shuffle(test.examples)
test.examples = test.examples[:test_size]

# Rebuild vocabulary
TEXT.build_vocab(train.text, min_freq=MIN_FREQ)
TAG.build_vocab(train.tag)
```

[ ]:
```
...
```

[ ]:
```
...
```

[ ]:
```
...
```

Type your answer here, replacing this text.

1.7 Debrief

Question: We're interested in any thoughts you have about this project segment so that we can improve it for later years, and to inform later segments for this year. Please list any issues that arose or comments you have to improve the project segment. Useful things to comment on include the following:

- Was the project segment clear or unclear? Which portions?
- Were the readings appropriate background for the project segment?
- Are there additions or changes you think would make the project segment better?

Type your answer here, replacing this text.

1.8 Instructions for submission of the project segment

This project segment should be submitted to Gradescope at http://go.cs187.info/project2-submit-code and http://go.cs187.info/project2-submit-pdf, which will be made available some time before the due date.

Project segment notebooks are manually graded, not autograded using otter as labs are. (Otter is used within project segment notebooks to synchronize distribution and solution code however.) **We will not run your notebook before grading it.** Instead, we ask that you submit the already freshly run notebook. The best method is to "restart kernel and run all cells", allowing time for all cells to be run to completion. You should submit your code to Gradescope at the code submission assignment at http://go.cs187.info/project2-submit-code.

We also request that you **submit a PDF of the freshly run notebook**. The simplest method is to use "Export notebook to PDF", which will render the notebook to PDF via LaTeX. If that doesn't work, the method that seems to be most reliable is to export the notebook as HTML (if you are using Jupyter Notebook, you can do so using `File -> Print Preview`), open the HTML in a browser, and print it to a file. Then make sure to add the file to your git commit. Please name the file the same name as this notebook, but with a `.pdf` extension. (Conveniently, the methods just described will use that name by default.) You can then perform a git commit and push and submit the commit to Gradescope at http://go.cs187.info/project2-submit-pdf.

2 End of project segment 2