Assignment 1

Due: Friday 02/14/2025 @ 11:59pm EST

Disclaimer

I encourage you to work together, I am a firm believer that we are at our best (and learn better) when we communicate with our peers. Perspective is incredibly important when it comes to solving problems, and sometimes it takes talking to other humans (or rubber ducks in the case of programmers) to gain a perspective we normally would not be able to achieve on our own. The only thing I ask is that you report who you work with: this is **not** to punish anyone, but instead will help me figure out what topics I need to spend extra time on/who to help.

Setup: The Data

Some settings, like typing on a phone, writing Chinese/Japanese characters, etc. are slow (even by human standards). It can be helpful for the machine to guess what the next character(s) the user will type. In this assignment, you will build statistical/neural character-level language models and test how well they can predict the next character.

Included with this file are some directories and a few files. The first notable element is the `data` directory. This contains both data files and code files. Here are the data files:

- `data/small`: A small training data corpus. Each line is a separate example.
- `data/large`: A much larger training data corpus. Each line is a separate example.
- `data/dev`: The development data. Each line is a separate example.
- `data/test`: The test data. Each line is a separate example.

These data files come from the NUS SMS Corpus, which is a collection of real text messages sent (mostly) by students at the National University of Singapore. More specifically, these data files come from the English portion of the dataset, although there are some examples of Singlish, a language that is a mixture of English, Malay, and other various Chinese languages.

The code files look like this:

- `data/charloader.py`: This file contains some utility functions for loading data from files sequences of sequences of characters (one sequence of characters per example).

Section: The API

In this assignment, all models you build will follow a specific api. First, every model you build should have a field called `vocab` of type `Vocab`, which contains the vocabulary that the model is aware of. Every model should also have the following methods:

- method `start()`. Some models have a "state" that must be maintained as the model processes a sequence. For n-gram models, the state is the current gram that the model is conditioning on (if applicable). For neural models, the state of the model is (at least one) real-valued vector. The `start()` method should return the initial state of the model.

- method `step(q, w_idx)`. This method takes two arguments: `q` is the prior state of the model, and `w_idx` is the token index of the current token being read by the model. This method should process the new token and calculate the new state of the model after reading the new token. This method should return two things: the new state of the model and a mapping of word indices to log probabilities. This log probability mapping is a probability mass function over the entire vocabulary (i.e. there should be an entry for every token in the vocabulary), and the values are calculated as the log probability of emitting a token vocabulary after reading `w_idx` with state `q`.

For instance, the code to compute the log-probability of `foo` would be (using model `m`):

```
q: StateType = m.start()
lp_foo: float = 0f

q, p = m.step(q, m.vocab.numberize("<BOS>"))
lp_foo += p[m.vocab.numberize("f")]

q, p = m.step(q, m.vocab.numberize("f"))
lp_foo += p[m.vocab.numberize("o")]

q, p = m.step(q, m.vocab.numberize("o"))
lp_foo += p[m.vocab.numberize("o")]

q, p = m.step(q, m.vocab.numberize("o"))
lp_foo += p[m.vocab.numberize("<EOS>")]
```

Section: Starter Code

In addition to the `data` directory, there are a few code files included with this pdf. These files are:

- `vocab.py`: This file contains the `Vocab` class. This class acts as a set of tokens, and will allow you to convert between the raw token and its *index* into the set. While converting between string tokens and their index in the vocabular is not explicitly required for statistical models, is it necessary for neural models. Since we are working with both in this assignment, we will always be working with the index of a token in the vocabulary to keep the api consistent.

- `models/ngram/unigram.py`: This file contains a statistical `Unigram` class which follows the required api.

- `models/nn/unigram.py`: This file contains a neural `Unigram` class which follows the required api. Note that this class, since it implements a neural network, has the code to train the model in the constructor.

- `lm.py`: This file contains the base class of our statistical language models. It also contains some userful types like `StateType` that are used in the earlier example of model usage. If you want a hint as how to design your model states, please take a look at the typing of this value.

- `predict.py`: This file will train (at the moment) a unigram model the training data and eval the unigram model to predict the next character you type into the console based on the characters you have types previously.

- `debug_statistical.py`: This file trains a Bigram language (which does not exist yet) on the example string from the lectures. I would really recommend using examples like this to help debug your code.

- `baseline_statistical.py`: This file trains a statistical unigram model and evaluates its performance on the development data. This file should give you an example of how I expect you to design your statistical code in this assignment.
- `baseline_nn.py`: This file trains a neural unigram model and evaluates its performance on the development data. This file should give you an example of how I expect you to design your neural code.

The file `predict.py` uses the `argparse` module, so you will need to supply arguments to this file if you choose to run it.

Baseline (10 points)

As mentioned previously, the file `models/*/unigram.py` implements a unigram language model (with the above interface) in the `Unigram` class. This class is used in the `baseline_[statistical, nn].py` file. Your first task is to run both of these files, and verify that the accuracy of a unigram model on the development data is $6620/40176 \approx 16.477\%$. These file will show you how to do a few things: how to separate your code into functions where path values and hyperparameters are hardcoded, how to load the data, and where/when the data should be padded with `<BOS>` and `<EOS>` during evaluation. Report your printouts in your report under a section titled `Baseline`.

n-gram Language Modeling (45 points)

Now you will try to improve the quality of your character predictions:

(20 points) Create a file called `models/ngram/ngram.py`. In this file you should implement a generic n-gram language model with the class `Ngram`. Your class should follow the same api as `models/ngram/unigram.Unigr` and it should take the value n as the first input to the constructor. I highly suggest you use the `debug_statistical.py` file to help debug your solution.

(3 points) Create a file called `experiment_ngram.py`. This file will work similarly to `baseline_statistical.py`. Create two functions: `train_ngram` and `dev_ngram`. These methods should perform the same experiment (i.e. train on `data/large`) you did with the `Unigram` model, but instead now use a 5-gram character-level language model. Your accuracy on the development set should be at least 49%. Report how many grams your 5-gram model has seen during training and your development accuracy in your report.

(20 points) Now try to make your language model better. In your report, briefly describe what modifications you tried, and for each modification, what accuracy on the development set was. You must try at least one modification, and the accuracy on the development set should be better than the accuray of your unmodified model.

(2 points) In your `experiment_ngram.py` file, write a function called `test_ngram`. This function should take a fully-trained instance of your best model (which `train_ngram` should now produce) as an argument, load in the test data (`data/test`, and, much like `dev_ngram`, should predict the test data and measure accuracy. To get full credit, you must score at least 55% on the test set.

Neural Language Modeling (45 points)

Let's see if a neural language model can do better:

(20 points) Create a file called `models/nn/rnn.py`. In this file you should implement a generic RNN language model with the class `RNN`. Your class should follow the same api as `models/nn/unigram.Unigram`. Your model should contain an instance of the `RNNCell` class in Pytorch.

(2 points) Create a file called `experiment_rnn.py`. This file will work similarly to `baseline_nn.py`. Create two functions: `train_rnn` and `dev_rnn`. These methods should perform the same experiment (i.e. train on `data/large`) you did with the `Unigram` model, but instead now uses your RNN model. Report how you decided when to stop training your RNN model, what hidden size you used, all other training parameters and decisions, and how many epochs your model took during training as well as your RNN development accuracy in your report. Your accuracy on the development set should be at least 34%.

(20 points) It turns out that RNNs aren't very sensitive to the dependencies in language. Add a class in your `models/nn/rnn.py` file called `LSTM` which uses a `LSTMCell` Pytorch class instead of a `RNNCell`. A LSTM (Long Short-Term Memory) RNN works similarly to a vanilla RNN, albeit with some changes. Check out this link for a description of how LSTMs are different from vanilla RNNs. Your `LSTM` class should follow the same api as the other neural classes.

(3 points) In your `experiment_rnn.py` file, write functions called `train_lstm` and `dev_lstm`. These functions should behave identically to `train_rnn` and `dev_rnn` except they train an instance of your `LSTM` rather than your `RNN`. Report how you decided when to stop training your LSTM model, what hidden size you used, all other training parameters and decisions, and how many epochs your model took during training as well as your LSTM development accuracy in your report. Your accuracy on the development set should be better than your RNN development accuracy%. Finally, create a function called `test_lstm`. This function should take a fully-trained instance of your best model (which `train_lstm` should now produce) as an argument, load in the test data (`data/test`, and, much like `dev_lstm`, should predict the test data and measure accuracy. To get full credit, you must score at least 53% on the test set.

Submission

Please turn in `models/ngram/ngram.py`, `models/nn/rnn.py`, `experiment_ngram.py`, and `experiment_rnn.py`. Note that the autograders will execute your code, and they will time out after 40min. To make sure that your submission doesn't time out the autograder, be sure to save the neural models to disk, and to turn these files in as well. The `train_rnn` and `train_lstm` functions that you write should, instead of training a model from scratch, instead load and return the trained models from disk. The model files you turn in to gradescope will not be put into any special directory, they will be put in the same directory as your `experiment_[ngram,rnn].py` files. Please also turn in the pdf report. There will be one gradescope entry for your code and another for your report.