Assignment 2

Due: Friday 03/26/2025 @ 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. Reminder, you are **not** to share code with others nor should you use virtual assistants to help in the development of your code in this assignment.

Setup: The Data

When processing languages without standardized spelling rules or historical language that followed rules that are no longer standard, spelling normalization is a necessary first step. In this assignment, you will build a model that learns how to convert Shakespearean original English spelling to modern Engligh spelling. The data in this assignment comes from the text of Hamlet from Shakespeare's First Folio in original and modern spelling. The training data contains all text up to where Hamlet dies (spoilers!), and the test data is the last 50 or so lines afterwards.
Included with this pdf is a directory called `data`. Inside this directory, you will find four files:

- `data/train.old`. This file contains training sequences (one sequence per line) of language that is spelled according to Shakespearean English spelling rules.
- `data/train.new`. This file contains the same training sequences (one sequence per line) of language that is spelled according to modern English spelling rules.
- `data/test.old`. This file contains test sequences (one sequence per line) of language that is spelled according to Shakespearean English spelling rules.
- `data/test.new`. This file contains the same test sequences (one sequence per line) of language that is spelled according to modern English spelling rules.

Section: Starter Code

In addition to the `data` directory, there are quite a few code files included with this pdf. Some of these files are organized into sub-directories for organization. The files and directories are:

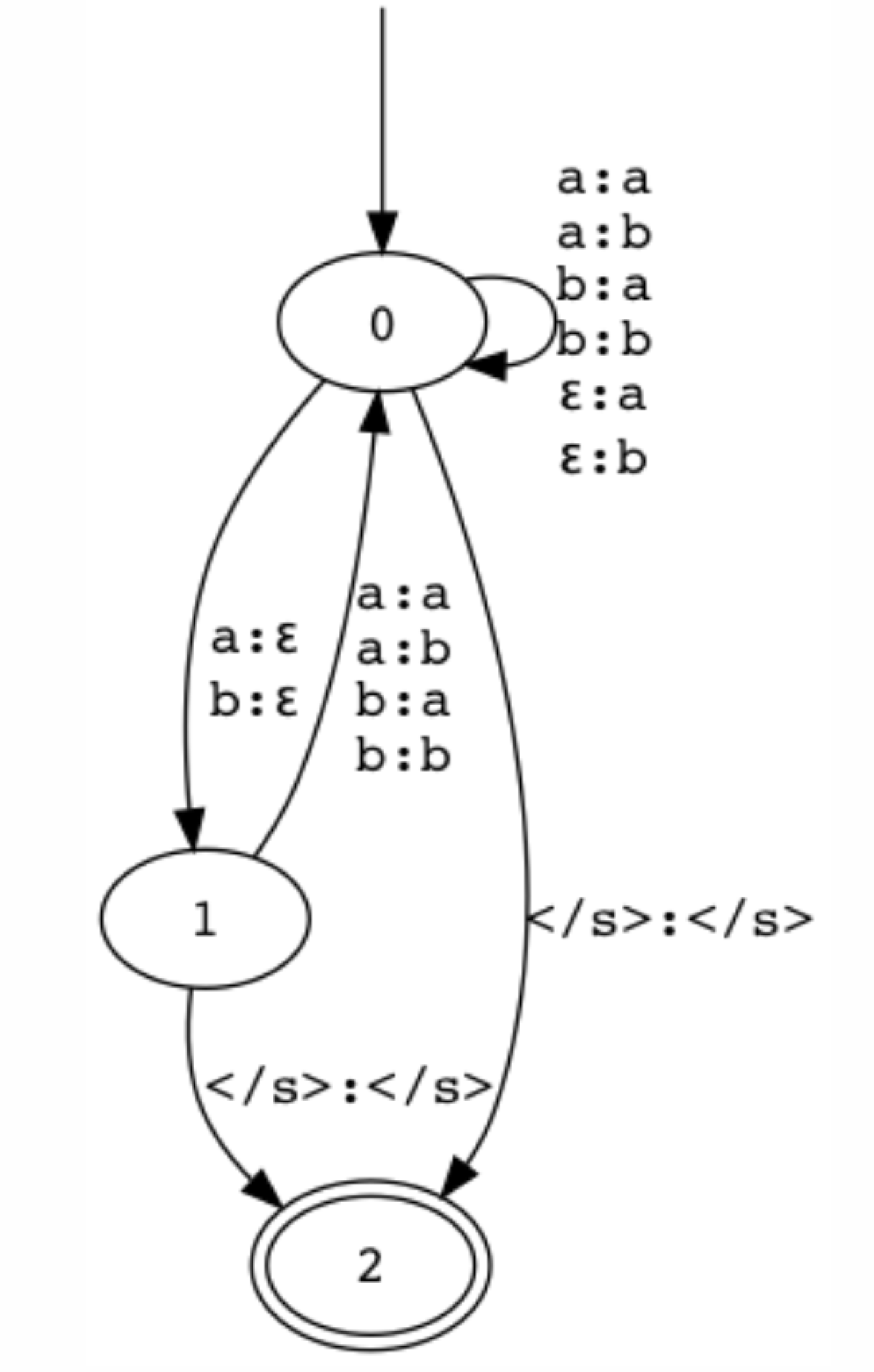
- `eval/`: This directory contains all code files used to evaluate a model's performance. In this assignment we will use just one flavor of performance, Character Error Rate:
  - `eval/cer.py`: This file contains a metric called "Character Error Rate". This metric measures the average Levenshtein distance between n sequence pairs. Lower is better.
- `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. This will become important in the future when we work with neural networks, but for now this functionality isn't strictly necessary but I want you to get used to seeing it.

- `from_file.py`: This file contains some useful functions for loading data from files.
- `fst.py`: This file contains a few types, but two most important are the `Transition` and `FST` class. The tl;dr of this file is that I have implemented Finite-State Transducers for you, including normalization and composition. If you want to create a fst in code, you should instantiate the `FST` class. You will then need to set the start and accept states, and provide transitions expressed as `Transition` objects for the `FST` instance to update. Whenever you want to lookup transitions, you will need to examine one of three mappings in your `FST` instance (lets call the instance `x`):
  - `x.transitions_from`: This mapping contains all outgoing edges from a state. The key is a state, and the value is a mapping of `Transition` objects to their weights.
  - `x.transitions_to`: This mapping contains all incoming edges to a state. The key is a state, and the value is a mapping of `Transition` objects to their weights.
  - `x.transitions_on`: This mapping contains all edges that use the same input symbol. The key is an input symbol, and the value is a mapping of `Transition` objects to their weights.

  One final helpful function in this file is called `create_seq_fst`. This function will, given a sequence, convert that sequence into a `FST` object.
- `topsort.py`: This file contains the code necessary to perform a topological ordering of the states of a `FST` instance. The entry-point function is called `fst_topsort`.
- `models/`: This directory contains the meat of this assignment. It contains three files, two of which you will need to complete:
  - `models/lm.py`: This file contains an implementation of a smoothed KneserNey language model and has code to convert such a language model into a fst.
  - `models/modernizer.py`: This file contains the model you will be completing in this assignment.

The Typo Model (10 points)

In this section you will learn about the typo model FST we will use to modernize Shakespearean spelling. Specifically, I want you to complete the `_create_typo_model` and `init_typo_model` methods in the `Modernizer` class. The typo model you will build looks like this (shown for a language with only two tokens, yours will have more edges but the same states and topology):



You can use the visualize method in the FST class to check that your typo-model is constructed with the correct topology. In the _create_typo_model I only want you to focus on creating an **unweighted** typo-model FST (with the correct topology), and then **setting** the weights of your typo-model FST in the **init_typo_model**. When you initialize the weights of your typo model, most letters in Shakespearean English will stay the same in modern spelling, so for transitions that consume symbol a and emit symbol a, you should assign a rather large (positive) weight compared to transtions of the form $a : b$ by like a factor of 100 or something. Additionally you should prefer transitions of the form $a : b$ over ϵ transitions, although the magnitude of this preference isn't as critical as the preference for $a : a$ transitions to $a : b$ transitions. When you are done assigning weights (remember they must all be positive values), you can call the normalize_cond method on a FST object to normalize all the pmfs.

Viterbi and Decoding (20 points)

Inside the Modernizer class, there is a method stub for a method called viterbi_traverse. In this assignment you will implement multiple flavors of the viterbi algorithm, but all flavors will share the same graph traversal. As discussed in class, you can separate the functionality into a traversal method (i.e. viterbi_traverse) and a suite of different viterbi wrapper methods that perform the various flavors of viterbi. Please see the method stub for viterbi_traverse to see a description of the function pointer you should pass as an argument.

Once your viterbi_traverse method is functioning, I want you to complete the viterbi_decode method. This method is where you will implement viterbi decoding, and it should use the your viterbi_traverse method.newline
Finally, to get decoding working, you should complete the decode_seq method. In this method, you will use your viterbi_decode method to find the largest weighted path (along with the weight which is a log-probability) of a FST. This FST you will construct using the following logic:

1. Build a FST for the sequence w that is given as the argument to the decode_seq method. Call this FST M_w
2. Compose M_{LM} (i.e. the FST for the language model), M_{TM} (i.e. the FST for the typo model) and M_w together to produce FST M. The ordering of the machines is important, but not the order of operations. See which one uses less RAM and time, and go with that order of operations. See the method description for more details.
3. Run your viterbi decoding algorithm on FST M.

Create a script called init.py that constructs your Modernizer, creates and initializes your typo model on the training data, and using the decode method, prints the first 10 decodings on the test set along with their log-probabilities in a method called decode. This method should take no arguments and should produce your fully-created Modernizer. Include these printouts in your report. Include in your report the Character Error Rate from decoding the entire test set. For full credit you should get at most 10%.

Forward and Brittle Training (35 points)

Inside the `Modernizer` class, there is a method stub for a method called `viterbi_forward`. This is where you will implement the forward algorithm described for HMMs, but works identically on FSTs. The good news is that it is only slightly different from `viterbi_decode`: while decoding calculates the largest-weighted path through the graph, the forward algorithm calculates the sum of all paths through the graph starting at the start state of the FST. This means that your `viterbi_forward` algorithm will be implemented almost identically to your `viterbi_decode`.newline
With your `viterbi_forward` algorithm working, you can use it to complete the `loglikelihood` method. This method calculates the log-likelihood of a collection of samples that are assumed to be distributed i.i.d. To calculate this, you should use your forward algorithm on each sequence w in the dataset to calculate $Pr[w]$, log these values and add up the log-probabilities. This sum of log-probabilities is what your `loglikelihood` method should return.

Once you can calculate the loglikelihood of the dataset, it is time to train the typo model using hard (brittle) expectation maximization. I am only asking you to implement the e-step, which is contained within the `brittle_estep` method. In this method, you need to calculate the "hard" counts for each (s, w) token pair, where s is a token in the modern English vocabulary, and w is a token in the Shakespearean English vocabulary. To speed up this process (and also to get better results), we are going to leverage parallel data. Therefore, during your E-step, the FST you decode is constructed differently than the one in your `decode_seq` method In your `brittle_estep` method, when considering Shakespearean sequence w, you also have its modern spelling s:

1. First build M_w like before (i.e. convert w into a FST).
2. Now convert s into a FST to create M_s.
3. Compose together M_s with M_{TM} and M_w. Note that in `decode_seq`, you used M_{LM} in place of M_s.

These hard counts are, as discussed in lecture, calculated by decoding a Shakespearean sequence w to get its most likely correction s^* (at least according to the model right now), and treating that s^* as if it was ground truth. The mapping of (s, w) to their counts is what your `brittle_estep` method should return.

You are now ready to train the model using brittle-EM. Write a small script called `hard_em.py` that constructs your Modernizer, creates and initializes your typo and language model on the training data, and then trains your typo model on the training data until the log-likelihood converges in a function called `train`. This function should take no arguments and produce a fully trained `Modernizer`. You will likely want to do some add-δ smoothing in your m-step (which I have already implemented for you, you just need to set the argument). I have already implemented the `brittle_em` method for you, so you don't need to implement it from scratch. Report a graph of the log-likelihood as a function of iteration (you may have to modify `brittle_em` to record the log-likelihoods as a function of iteration) in your report. What is the Character Error Rate of your model on the test set? For full credit it should be better than 7.5%.

Backward and Flexible Training (35 points)

Inside the **Modernizer** class, there is a method stub for a method called `viterbi_backward`. This method is where you will implement the backward algorithm, which is a mirror of the forward algorithm. While the forward algorithm calculates the sum of all paths from the start state to a node in the graph, the backward algorithm calculates the sum of all paths starting from a node and ending at the accept state. We calculate this by implementing the forward algorithm but in the reverse direction: instead of traversing the vertices in topological order, we do so in reverse topological order, and instead of examining incoming edges of a state, we examine outgoing edges of a state. As such, your backward algorithm should look suspiciously similar to your forward algorithm. Remember, they should both calculate $Pr[w]$, so you can easily check correctness!

Once your backward algorithm is finished, you can complete the `flexible_estep` method. In this method, you still have access to parallel data just like in `brittle_estep`, so you should still build the same composed FST as in that method. The difference between the two is how you calculate your counts. The mapping you produce should have the same structure, but the values of the counts will differ, as in this method you should calculate "soft" counts instead of "hard" ones.

You are now ready to train the model using flexible-EM. Create a script called `soft_em.py` almost identical to the last one, but train a model with soft-EM instead of hard-EM. Report a graph of the log-likelihood as a function of iteration in your report. Which flavor of EM performs better with respect to our Character Error Rate metric? Which flavor of EM performs better with respect to log-likelihood? For full credit, your soft-EM model should also be better than 7.5% according to Character Error Rate.

Submission

Please turn in all of the files in your repository to gradescope. Due to a student in HW1, you have lost the ability to self-report your performance. Training a model will take tens of minutes apiece (mine takes around 30min for hard-EM and longer for soft-EM), so please do **not** train on the autograder. Instead, you should save your model to disk (I recommend using `pickle`). When you turn in your code, please adjust `soft_em.train` and `hard_em.train` so that they load these files from disk into fully-trained `Modernizers` instead of training them from scratch. Please also submit the serialized models. Assume that on gradescope they will be placed in the same directory as `soft_em.py` and `hard_em.py`. In your report you should include console output of having run your code as well as any observations you made during your experiments. If an instruction in this prompt says to report on something, you should include those elements in your pdf. Please do not handwrite these: typeset pdfs only. There will be separate links on gradescope for your code and your pdf. Your code will be autograded, while your report will be graded by us.