Assignment 2

Due: Friday 10/11/2024 @ 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

In 2005, a viral blog post told the story of how a person bought a bootleg copy of *Star Wars Episode 3: Revenge of the Sith* in Shanghai with English subtitles that were often hilariously wrong. These subtitles were generated by some form of (low-quality) machine translation. Can you do better?
In this assignment, you will be translating the Mandarin script for *Revenge of the Sith* back into English and comparing against the English transcript of the movie. Included with this pdf is a directory called `data` that contains all of the training/testing data we will need for this task:

- `train.zh-en`: This file contains a parallel corpus (i.e. sentence pairs) of the prequels (except episode 3) and the original triology (i.e. the training data contains episodes 1,2, and 4-6).
- `train.align`: This file contains word alignments for every sentence pair in `train.zh-en`. These alignments are not perfect: they were generated using IBM Model 4 (a better model), so we will use them as a "silver" standard to evaluate the alignments produced by your model.
- `test.zh`: *Star Wars Episode 3: Revenge of the Sith* Mandarin transcript.
- `test.en`: *Star Wars Episode 3: Revenge of the Sith* English transcript.

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 two flavors of performance: **f1** score and **bleu**:
  - `eval/align_f1.py`: This file will be used to evaluate the alignments produced by your IBM Model 1 by comparing against the "silver" alignments produced by IBM Model 4. Performance is calculated as a **f1** score, which is the balance point between precision and recall. **f1** produces a score between 0 and 1, and larger is better.
  - `eval/bleu.py`: This file will be used to evaluate the translations produced by your model by comparing against the "gold" English script of epsiode 3. Performance is calculated using **bleu** score, which is a weighted average of n-gram accuracies. **bleu** produces a score between 0 and 1, and larger is better.

- `utils.py`: This file contains a few useful functions for reading/writing to files, but most importantly it 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. Even though they are duplicated with some of the functionality in `utils.py`, I find them to be a little easier to use.

- `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`.

- `[part1/part2].py`: These files correspond to part1 and part2 of this assignment. I have implemented some of it for you so you can get the feel of how using the starter code should go. I also implemented these parts to provide more structure to the assignment. Some of the functions in these files I have left up to you. See the instructions below for more details.

- `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/ibm1.py`: This file contains a partial implementation of IBM Model 1. You will be completing this implementation as part of your assignment.
  - `models/translator.py`: This file contains a partial implementation of an object that will use some of the transitions learned by IBM Model 1 (also referred to as "rules") to decode a Mandarin sequence into an English sequence.

Word Alignment (45 points)

As mentioned previously, in this part you will work on two files: `models/ibm1.py` and `part1.py`. Please complete the following steps:

a I have already written a function in `part1.py` that will load in the aligned corpus from `data/train.zh-en`. This function will return a pair of double-layered list objects. The first element of the tuple is the Mandarin training data and the second element of the tuple is the English training data.

b In the file `models/ibm1.py`, please complete the following two methods:

- `estep`. This method should compute a mapping of "soft" counts from the two training corpi (included as arguments in the method). Remember that the e-step for IBM1 uses the following equations for every sentence pair and for each Mandarin position $j = 1, \cdots, m$ and English position $i = 0, \cdots, l$:

$$c(f_j, e_i) \leftarrow c(f_j, e_i) + \frac{t(f_j|e_i)}{\sum_{k=0}^{l} t(f_j|e_k)}$$

  Note: In order to save time and memory, I have implemented a method called `_init_tm` within the IBM1 class. This method examines all sentence pairs, and creates transitions in the FST for every Mandarin word f and English word e if those two words appear in the same line. So, if a word pair is never observed occuring within the same sentence pair, a transition for that pair will never be created. You are welcome to change this, but it will increase the runtime and memory usage of your code (and gradescope only uses around 6GB of RAM and 40min max runtime).

- `mstep`. This method should use the counts calculated in the `estep` method (included as an argument to the method) to update the weights of the fst object stored in the `self.tm` field. In this step, all we need to do is normalize the counts back into probabilities. So, for every mandarin word f and english word e (including the NULL word):

$$t(f|e) \leftarrow \frac{c(f, e)}{\sum_{f'} c(f', e)}$$

  Reminder: you can do this by reweighting transitions and then asking the fst to normalize a conditonal probability itself for you.

I have already written a function in `part1.py` that will instantiate your IBM1 model and call the training infrastructure you completed in this part. One way you can check the success of your model is that the log-likelihood of the training data should monotonically increase. If it does not, you for sure have an error in either your estep or mstep methods. Additionally, your log-likelihood should eventually get better than -155000. This function will also lookup and display the probabilities of some correct English and Mandarin word pairs. Please include this printout in your report. On my machine, this entire process took around 6min to run and used around 3GB of RAM.

c In the file `part1.py`, please complete the function `partc`. I have already written code that will use the fst your EM methods trained to produce the most probably token alignment. Your function should take this alignment, display the first 5 alignment sequences, write all of the alignments to a file at the provided path, and then calculate and display the performance of your alignments against the "silver" IBM Model 4 alignments. Your f1 score should be at least 50%. Please include these printouts in your report. This process took around 2min for me and used around 3GB of RAM.

Monotone Decoding (45 points)

True IBM Model 1 decoding should search through all possible reorderings of the input sequence. Since this is somewhat involved, we'll try something simpler, which is to translate Mandarin sequences word-for-word without any reordering:

a In the file `part2.py`, I have created a function that will instantiate a `Translator` object and train it on the training corpus. The `Translator` class is mostly a wrapper around your `IBM1` class with some notable exceptions. When we want to actually perform a translation, we need to compose three FSTs together to get a machine that assigns probability to every possible translation for an observed input sequence (in this case a Mandarin test sequence). As described in lecture, we would normally compose a language model (in this case a KneserNey-smoothed bigram model) with IBM1, and then take the resulting machine and compose it with a FST built from the input sequence. This process, while effective, is inefficient. A more efficient approach is to do the composition in the reverse order: take the FST built from the input sequence and compose it with the "translation model" (i.e. IBM1), and then take that result and compose it with the language model. This process will compute a FST that defines the same probabilities, however it will have significantly smaller runtime and memory overhead due to the intermediary FST being smaller.

   Additionally, we don't take IBM1 directly and use it as a translation model. Instead, we will be copying some of the transitions from IBM1 and building a new FST from them that we will use in our compositions. This translation model has two states, `q0` and `q1`. The state `q0` serves the same purpose as the sole state of IBM1: to house transitions that read in a symbol, and produce a symbol. The state `q1` serves as the accept state, which will take a single transition from `q0` that consumes and emits `<EOS>`. Here is how we build this translation model (don't worry I already did this for you in the `Translator.use_pretrained_tm` method:

   - Create a fst with two states: a start state `q0` and an accept state `q1`.
   - Add a transition from `q0` to `q1` that consume and produces `<EOS>`.
   - For each observed Mandarin token f, for each of its top k (default of 10) most probable English translations e, add a transition from `q0` to `q0` that consumes f and produces e with probability $t(f|e)$. If e = NULL, make the transition output ϵ (the empty token).
   - In this assignment we will handle unknown Mandarin symbols by simply deleting them. For each Mandarin token f in the **test set**, add a transition from `q0` to `q0` that consumes f and produces ϵ with a really small probability (default of 10^{-100}).

b In the file `part2.py`, I have created a function that will use a trained `Translator` object to decode each sequence of the test set (i.e. episode 3). The trouble is that this code is incomplete: in `models/translator.py` you need to finish the `viterbi` method, which should calculate the most probable path through the composed FST, and return a few things in a specific format. I refer you to the comments I left at this method for more details. When your code works, the function in `part2.py` will print out the first 10 translations, write all of the translations to a file, and then use `eval/bleu.py` to calculate and print your bleu score. On my machine, this took around 3GB of RAM, yet took around 20min to fully decode the entirety of episode 3. Your bleu score should be at least 0.01. Please include your printouts in your report.

Translation Improvement (10 points)

In this section, you should play with the hyperparameters of this model to try and improve the bleu score of your translations. Some low-hanging fruit are the gram size of the language model and the number of transitions for each Mandarin token f that are kept from IBM1 when constructing your translation model (called *rules*). For full points, you should be able to achieve a bleu score of at least 0.02.

Submission

Please turn in all of the files in your repository to gradescope, whether they are generated content or otherwise. In addition, please submit your report on gradescope. 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.