Rust-Python HMM

For this assignment I decided to focus on the model computational performance. Specifically, I implemented the assignment also in Rust to see what the performance gain will be. The reasons are that (1) I want to learn Rust but every class is Python + PyTorch and (2) current NLP research is all done by prototyping in Python, yet there is virtue in experience of programming usable deployable and fast solutions. Note: I don't want to be mean to Python, it certainly has its place and advantages.

I understand that the task was to programm this in Python, which I hope I fulfilled. Yet I also hope that you will find these comparison interesting.

The Rust code is twice as large* and took much longer to complete. The benefits are, however, that once it compiled, I was convinced of its functionality, which was not the case with Python. *Running `grep -r -E "^\s*[{}]\s*$" rust/src/ | wc -l` reveals that more than 100 lines are just opening or closing brackets, so the code size is not that significant.

Project structure

```
data/                   # not supplied, paste the files here for reproducibility
 - de-{eval,train}.tt
 - de-test.t
data_measured/
 - {r,p}-de-eval-{,smooth}.tt # model outputs
 - time-{1,2,3}         # measured results for graphs
 - time-{1,2,3}.png     # exported graphs
meta/                   # scripts for measuring performance and accuracy
 - graph.py             # produce graphs given logs time-{1,2,3} in data_measured
 - run_times.py         # measure performance from r-build-time and p-run-time recipes
rust/                   # Rust source code
python/                 # Python source code
Makefile                # Makefile for common recipes
```

Makefile and reproducing results

`make r-print-eval` trains two models on the data and outputs the CONLL-U file to `data_measured/p-de-eval.tt` and `data_measured/p-de-eval-smooth.tt`, similarly `make r-print-eval` produces `data_measured/r-de-eval.tt` and `data_measured/r-de-eval-smooth.tt` (assuming stable Rust compiler in path). The semantics of the rest of the command line arguments is intuitive from the Makefile: `print_acc` self-reports the accuracy on anything it computes (`comp_test`, `comp_train` or `comp_eval`).

File paths are relative hardcoded, because there are no plans to make this portable and there were already too many switches. Both versions assume that they are run from the top-level directory (the directory thie `README.md` is in). If `print_pred` is present, the program outputs predictions to stdout. Progress is outputed to stderr.

Correctness

Even though the Viterbi algorithm should be mostly deterministic, there is a big issue with number representation and rounding. There appears to be a big difference in accuracy based on the underlying numeric type used (f32 vs f64). All parameters were multiplied by `4096` in both versions, because this maximized the performance (possibly striking the sweet spot between diminishing and exploding values). In trellis computation, the layers are all normalized to sum to one after every step.

Despite my best efforts, the two versions produce slightly different results. This may be due to different corner-case numeric handling in the two systems.

Unseen tokens were dealt with by substituting the emission probability with 1, thus relying on the surrounding transition probabilities.

I tried to use the same algorithmic steps in both solutions, so that they are comparable. It is, however, still possible, that I mistakenly used some other data structure, assuming it was the same.

Log space

Another solution to the issue of storing very small probabilities would be to work in log space. One of the issues is that it no longer supports the computation of cummulative probability (because the probabilities there are summed) and also it had a negative effect on performance relative to the current solution: for (train, eval) accuracy, the new results in Rust were (89.16%, 78.96%) and in Python (66.67%, 66.98%).

Code structure

Structures in both versions follow the same naming scheme. The programs function as follows:

1. Train `Loader` is created, which also creates a `Mapper` objects (see Note)
2. HMM Model parameters are estimated from the training data.
3. Eval or Test `Loader` is created, reusing Training `Mapper`.
4. Based on the arguments, datasets are evaluated (`comp_test`, `comp_train` or `comp_eval`).

The `HMM` class contains code for initialization and Viterbi and can be used generically. `HMMTag` inherits from this class and adds specific functions for initialization from `Loader` and evaluation. Both implementations start with `main.{rs,py}`.

Performance Graphs

The performance was measured with respect to changing training data size (steps of 10000 tokens). The task was (1) train, (2) train + evaluate on eval, (3) train + evaluate on train and eval. Accuracy of these models was also measured. The measured times are without writing to files. Rust version is compiled with the `--release` flag and Python is run with `-O`.

Figure 1 shows simply that in training, the Rust implementation seem to be faster by the factor of ~7.

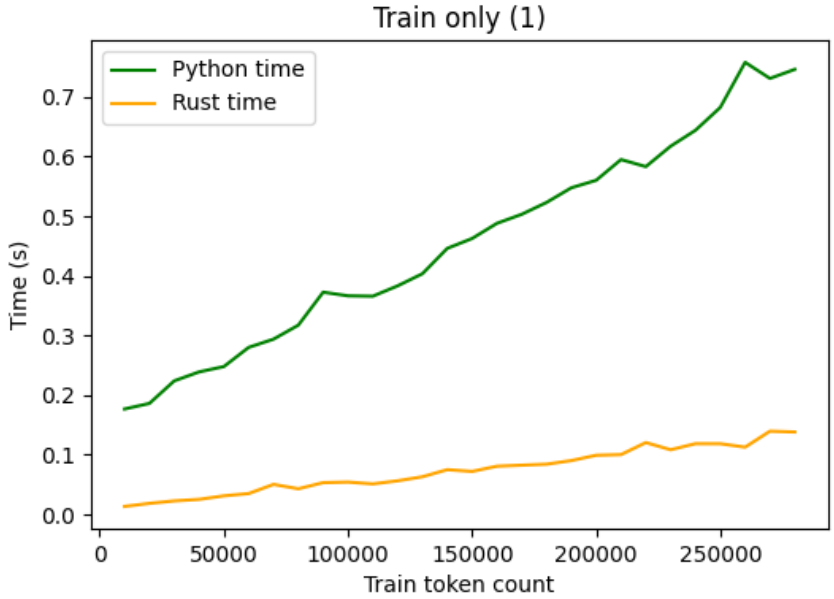
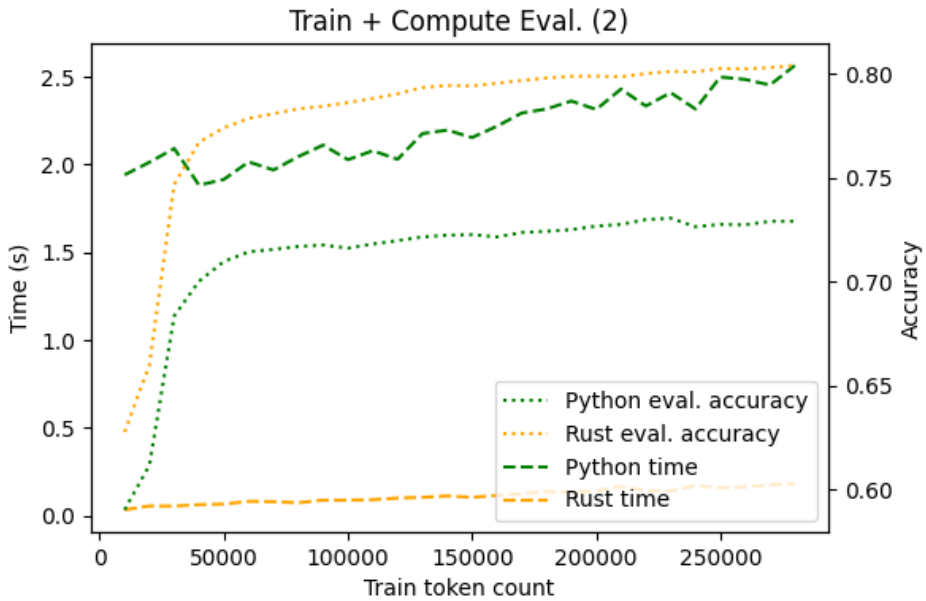
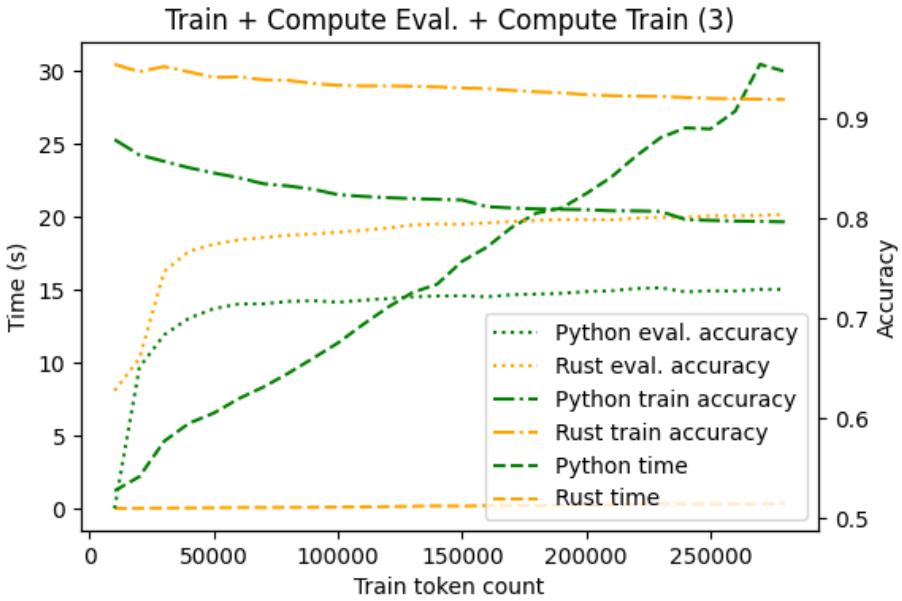


Figure 2 shows also that the Rust implementation is more stable (possibly because of the lack of runtime). We also see that there seem to be diminishing return in performance after we pass 50k train tokens. Python ends at 2.56s and Rust on 0.18s.



Evaluating the whole data proved to be the most difficult task. This is shown on Figure 3. While Python ends at 29.95s, for Rust it is 0.39s. The training accuracy is also decreasing, because the capacity of the model is getting shared with larger amount of examples. Train accuracies were 91.89% and 79.65% for Rust and Python, respectively. Evaluation accuracies were 80.40% and 72.91%.



Note on Performance

I did not try to especially optimize the algorithmic performance. For example the trellis is allocated and cleared for every sentence in the data. This could be done much more efficiently by creating one static one (the size of the longest sentence) and reusing that for the computation. It does not need to be cleared, because every cell is first written to and only then read.

One of the biggest performance boosts was gained by creating a hashmap mapping from string (both for words and for tags), convert everything to numbers (Rust version uses 8bytes, which is unnecesary), manipulate just these numbers and only when printing revert back. This is done by the `Loader` and `Mapper` classes in both versions.

Also both versions contain code for computing sequence observation probability in trellis (`sum` instead of `max`), but is turned off in both versions. The Rust version gets an unfair advantage in this, because it is removed compile time, while in Python, the interpreter has a bit more work to do.

Additional

Smoothing

I also experimented with rudimentary smoothing. This can be done easily by changing the initial probabilities in constructor (class `HMM`) to some parameter `alpha` instead of zeroes. Since probabilities are scaled up by the factor of `4096`, it makes sense to use higher values.

Interestingly enough, the performance increased by tinkering with start and transition probabilities and not emission probabilities. Furthermore, setting initial transition probability to a negative number `-128` and the start probability to `64` resulted in the best results (I did not employ gridsearch, so there surely exists a better set of parameters. The resulting (train, eval) accuracies were (92.58%, 80.86%) and (81.72%, 73.98%) for Rust and Python respectively. This is an improvement of (+0.69%, +0.46%) and (+2.07%, +1.07%) for Rust and Python. The resulting inferences are stored in `data_measured/{p,r}-de-eval-{,smooth}.tt`.

Ice cream

The Rust code also contains the toy ice-cream X weather example. It can be run from the `rust` directory with `cargo test -- --nocapture`.

Unknown word handling by subwords

This is an idea beyond the scope of this homework, but I would nevertheless like to see it implemented (and especially to see the performance) or any comments that show the caviats of this approach.

In order to better handle unknown word handling, all tokens could be split into subword units, e.g. by Byte Pair Encoding. This would allow the splitting to be trained not only on annotated data, but also on unanotated. The HMM parameters could be then estimated as follows:

Assume the sequence `SENT A-B C-B` (BPE compound `A-B` at the beginning of the sentence, followed by `C-B`). Since individual subwords have the same POS tags, the starting and transition probabilities can be computed in almost normal way: both `A` and `B` are starting and both `A`, `B` are followed by `C, D` (4 transitions). Furthermore, emission probabilities can also remain unchanged. This is counterintuitive, because it will lead to affixes with POS tags as the same word (e.g. `un-do-able -> (un, ADJ), (do, ADJ), (able, ADJ)`.) To avoid this, I would suggest early stopping of the BPE algorithm.

Further assume, that we trained two sets of HMM parameters: in the standard way `(E, T, P)` and also with subword units `(E', T', P')`. The main difference would be in inference. If the next token to be processed is present in the training data, the standard parameters and approach would be used. If it is however not in the training data, it is split to subwords: `c = A1-A2-..-An`. The starting and transition probability would be estimated from `P'` and `T'`. Emission probability would then be the average of the parameters for individual subwords: `E''(c, s) = [E'(A1, s)+E'(A2, s)+..+E'(An, s)]/n`.

The emission probability function can be extended to convex interpolate between the standard and subword version: `E'''(c, s) = a * E''(c, s) + (1-a) * E(UNK, s)`. Here `a` is a parameter, which can be estimated from heldout data.

Unknown word handling by Stemming

Completely another approach would be some sort of sensitive stemming, which would remove affixes that do not change the part of speech. This would reduce the amount of word forms, while preserving correctness in annotation.