nlp 2019 course

Assignment 1: Sequence Tagging

In this assignment you will implement a sequence tagger using the Viterbi algorithm.

You will implement 2 versions of the tagger:

1. An HMM tagger.
2. A MEMM tagger.

You will evaluate your taggers on two datatsets.

In addition to the code, you should also submit a short writeup, see below.

At the time of the assignment is published, we did not cover the material for all of the parts, so the submission date is somewhat far into the future. But you can definitely start now, and advised to do so: things take time.

Software

Use Python 3 for this exercise.

Personal Details

include inside the submission a text file named `details.txt`, with your details. The format of this file is: `first-name SPACE family-name SPACE id`. If you submit in pairs, write the details of both of you, each one in a separate line. For example:

```
Shira Cohen 12345678
```

```
Or Levi 87654321
```

Without the details files, we cannot grade your assignment.

Data

The training and test files (tagged corpora) needed for this assignment are available on piazza

We will be using two datasets: one for part-of-speech tagging, based on newswire text, and the other for named-entity recognition based on Twitter data.

The instructions below assume the part-of-speech tagging data, which is the main task. However, as sequence tagging is a generic task, you will also train and test your taggers on the named entities data.

HMM Tagger

When implementing the HMM tagger, there are two tasks: (a) computing the MLE estimates q (transition probabilities) and e (emission probabilities), (b) finding the best sequence based on these quantities using the Viterbi algorithm.

HMM task 1: MLE estimates (20 pts)

In this part you need to compute the estimates for e and q quantities based on the training data. In class 3, we refer to the quantities `q` as **transition probabilities** and the quantities `e` as **emission probabilities**.

The input of this part is a tagged corpus. The output is two files: `e.mle` will contain the info needed for computing the estimates of e, and `q.mle` will contain the info needed for computing the estimates of q. The files will be in a human-readable format.

As discussed in class, the estimates for q should be based on a weighted linear interpolation of p(c|a,b), p(c|b) and p(c) (see slide 120 in class 3).

The estimates for `e` should be such that they work for words seen in training, as well as words not seen in training. That is, you are advised to think about good "word signatures" for use with unknown words that may appear at test time.

The format of `e.mle` and `q.mle` files should be the following: Each line represents an event and a count, separated by a tab. For example, lines in `e.mle` could be:

```
dog NN<TAB>345
*UNK* NN<TAB>939
```

While lines in `q.mle` could be:

```
DT JJ NN<TAB>3232
DT JJ<TAB>3333
```

(the numbers here are made up. `<TAB>` represents a TAB character, i.e. `"\t"`)

Word-signature events in `e.mle` should start with a `^` sign, for example:

```
^Aa NN<TAB>354
```

This indicates a signature for a word starting with upper-case letter and continuting with lowercase. Beyond starting with `^`, you are free to use whatever format you choose for the word signatures. Lines for actual words should not start with `^`.

Beyond saving the corpus-based quantities in `q.mle` and `e.mle` you code should also contain methods for computing the actual q and e quantities. For example, a method called `getQ(t1,t2,t3)` (or another name) will be used to compute the quantity `q(t3|t1,t2)` based on the different counts in the `q.mle` file. You will use them in the next sections when implementing the actual tagging algorithm.

What to submit:

A program called `MLETrain` taking 3 commandline parameters, `input_file_name`, `q.mle`, `e.mle`

The program will go over the training data in `input_file_name`, compute the needed quantities, and write them to the filenames supplied for `q.mle` and `e.mle`.

Your program should be run as:

```
python3 MLETrain.py input_file_name q.mle e.mle
```

Note: If this part takes more than a few seconds to run, you are doing something very wrong.

HMM task 2: Greedy decoding (5 pts)

Implement a greedy tagger. You will assign scores for each position based on the q and e quantities, using the greedy algorithm presented in class.

See "what to submit" instruction in task 3 below.

HMM task 3: Viterbi decoding (20 pts)

Implement the Viterbi algorithm. In this part, your Viterbi algorithm will use scores based on the e and q quantities from the previous task, but it is advisable (though not required) to write it in a way that will make it easy to re-use the Vietrbi code in the MEMM part, by switching to the scores derived form a log-linear model (that is, the Viterbi code could use a `getScore(word,tag,prev_tag,prev_prev_tag)` function that can be implemented in different ways, one of them is based on `q` and `e`).

You need to implement the Viterbi algorithm for Trigram tagging. The straight-forward version of the algorithm can be a bit slow, it is better if you add some pruning to not allow some tags at some positions, by restricting the tags certain words can take, or by considering impossible tag

sequences. This is necessary for a reasonable running time. ***If your code runs for more than 4 minutes, it will not be graded (i.e., will receive a 0 score).***

What to submit:

Two text files named `hmm-viterbi-predictions.txt` and `hmm-greedy-predictions.txt` with your predictions on the development file `ass1-tagger-dev`.

A program called `GreedyTag` (implementing greedy tagging) taking 5 commandline parameters, `input_file_name`, `q.mle`, `e.mle`, `greedy_hmm_output.txt`, `extra_file.txt`.

A program called `HMMTag` (implementing viterbi tagging) taking 5 commandline parameters, `input_file_name`, `q.mle`, `e.mle`, `viterbi_hmm_output.txt`, `extra_file.txt`.

`q.mle` and `e.mle` are the names of the files produced by your MLETrain from above. `input_file_name` is the name of an input file. The input is one sentence per line, and the tokens are separated by whitepsace. The format of `viterbi_hmm_output.txt` and `greedy_hmm_output.txt` should be the same as the input file for `MLETrain`. `extra_file.txt` this is a name of a file which you are free to use to read stuff from, or to ignore. Even if you do not use an extra file, you should still accept it as an argument.

The purpose of the `extra_file.txt` is to allow for extra information, if you need them for the pruning of the tags, or for setting the lambda values for the interpolation. If you use the extra_file in your code, you need to submit it also together with your code.

The program will tag each sentence in the `input_file_name` using the tagging algorithm and the MLE estimates, and output the result to `out_file_name`.

Your program should run as follows:

```
python3 GreedyTag.py input_file_name q.mle e.mle greedy_hmm_output.txt extra_file.txt
```

```
python3 HMMTag.py input_file_name q.mle e.mle viterbi_hmm_output.txt extra_file.txt
```

Your tagger should achieve a dev-set accuracy of at leat 95\% on the provided POS-tagging dataset.

We should be able to train and test your tagger on new files which we provide. We may use a different tagset.

MEMM Tagger

Like the HMM tagger, the MEMM tagger also has two parts: training and prediction.

Like in the HMM, we will put these in different programs. For the prediction program, you can likely re-use large parts of the code you used for the HMM tagger.

MEMM task 1: Model Training (20 pts)

In this part you will write code that will extract training data from the the training corpus in a format that a log-linear trainer can understand, and then train a model.

This will be a three-stage process.

1. Feature Extraction: Read in the train corpus, and produce a file of feature vectors, where each line looks like:

   `label feat1 feat2 ... featn`

   where all the items are stings. For example, if the features are the word form, the 3-letter suffix, and the previous tag, you could have lines such as

   `NN form=walking suff=ing pt=DT`,

   indicating a case where the gold label is `NN`, the word is `walking`, its suffix is `ing` and the previous tag is `DT`.

2. Feature Conversion and training: Read in the features file produced in stage (1), produce a file in the format the log-linear trainer can read, and train the model, to produce a trained model file. The classification model we will use goes by various names, including `multinomial logistic regression`, `multiclass logistic regression`, `maximum entropy`, `maxent`.

For feature conversion and model training, we will use scikit-learn python package (also called `sklearn`). The sklearn package contains many machine learning algorithms, and is worth knowing. To convert the features you extracted to a format than can be fed to the model, use sklearn DictVectorizer. Alternatively, you can convert the features file using load*svmlight*file(). Then, train a logisitc regression model (make sure to use the softmax classifier, and not the "one vs. rest" classifer).

To be precise, you need create and submit the following programs, that run as follows:

1. `python3 ExtractFeatures.py corpus_file features_file`
2. `python3 TrainSolver.py features_file model_file`

You can save the mapping you create between features and integers to a file named `feature_map_file` (to be use in test time).

The first parameter to each program is the name of an input file, and the rest are names of output files. `model_file` is your trained model (***Do not submit it***)

Implement the features from table 1 in the MEMM paper (see link in the course website)

What to submit:

Submit the code for the programs in stages (1) and (2). You also need to submit a file named `features_file_partial`. This file contains the first 100 and last 100 lines in the `features_file`.

You can create the partial file on a unix commandline using:

```
cat features_file | head -100 > features_file_partial
```

```
cat features_file | tail -100 >> features_file_partial
```

You can verify the number of lines using:

```
cat features_file_partial | wc -l
```

(the result should be `200`)

MEMM task 2: Greedy decoding (5 pts)

Implement a greedy tagger. You will assign scores for each position based on the max-ent model, in a greedy fashion.

See "what to submit" instruction in task 3 below.

MEMM task 3: Viterbi Tagger (15 pts)

Use the model you trained in the previous task inside your vieterbi decoder.

What to submit:

Two text files named `memm-viterbi-predictions.txt` and `memm-greedy-predictions.txt` with your predictions on the test file `ass1-tagger-test-input`.

A program called `GreedyMaxEntTag` taking 4 commadnline parameters, `input_file_name`, `modelname`, `feature_map_file`, `out_file_name`.

A program called `MEMMTag` taking 4 commadnline parameters, `input_file_name`, `modelname`, `feature_map_file`, `out_file_name`.

- `input_file_name` is the name of an input file. The input is one sentence per line, and the tokens are separated by whitepsace.
- `modelname` is the names of the MaxEnt trained model file.
- `feature_map_file` is the name of the features-to-integers file, which can also contain other information if you need it to.
- `out_file_name` is the name of the output file. The format here should be the same as the input file for `MLETrain`.

The purpose of the `feature_map_file` is to contain the string-to-id mapping, as well as other information which you may need, for example for pruning of the possible tags, like you did in the HMM part.

The programs will tag each sentence in the `input_file_name` using using either the MaxEnt (logistic-regression) predictions and greedy-decoding, or using the MaxEnt predictions and Viterbi decoding, and output the result to `out_file_name`.

Note: If memory usage is an issue, you can (1) use a sparse representation in DictVectorizer, and (2) use SGDClassifier with loss='log', instead of `linear_model.LogisticRegressio`. This model has a methdo `partial_fit()` that can be run on a subset of the dataset. ***If the program runs for more than 10 minutes, it will not be graded.***

Your program should be run as:

`python3 GreedyMaxEntTag.py input_file_name model_file_name feature_map_file output_file`

`python3 MEMMTag.py input_file_name model_file_name feature_map_file output_file`

Your tagger should achieve a dev-set accuracy of at leat 95\% on the provided POS-tagging dataset.

Clarification: In class (and in the MEMM paper) the features are written as `form=dog&label=NN`, and we assume a feature for each label, that is, we will also have `form=dog&label=DT`, `form=dog&label=VB` etc. That is, the label is part of the feature. In the solvers input, we take a somewhat different approach, and do not include the label as part of the feature. Instead, we write the correct label, and all the label-less features. The solver produces the conjunctions with all the different labels internally. Over the years, this is something that was very hard for some students to grasp. I made this illustration which I hope will help to explain this.

The Named Entities Data (15 pts)

Train and test your taggers also on the named entities data.

Note that there are two ways to evaluate the named entities data. One of them is to look at the per-token accuracy, like in POS tagging (what is your per-token accuracy?).

The NER results are lower than the POS results (how much lower?), look at the data and think why.

Another way is to consider precision, recall and F-measure for correctly identified spans. (what is your spans F-measure?).

You can perform span-level evaluation using this script: ner_eval.py, which should be run as: `python ner_eval.py gold_file predicted_file`

make sure this program runs without exceptions; it will be used for grading.

The span-based F scores are lower than the accuracy scores. Why?

Can you improve the MEMM tagger on the NER data?

(maybe these lexicons can help)

What to submit:

1. An ascii text file named `ner.txt` containing the per-token accuracy on the ner dev data, and the per-span precision, recall and F-measure on the ner dev data. You should include numbers based on the HMM and the MEMM taggers. Include also a brief discussion on the "Why"s above, and a brief description of your attempts to improve the MEMM tagger for the NER data.
2. Files named `ner.memm.pred` and `ner.hmm.pred`, containing the predictions of your HMM and your MEMM models on the `test.blind` file.

Misc

Use python 3.x. You can use packages such as `sklearn` or `numpy`.

The code should be able to run from the commandline, without using PyCharm or any other IDE.

The code for all assignments should be submitted in a single `.zip` or `.tar.gz` file.

The files for all the tasks should be inside a single directory (this is the directory you have to submit), **named** `assignment1`)

The writeup should be in the same directory, in a `pdf` file called `writeup.pdf` .

You code should run from the commandline on a unix system according to the instruction provided in each section.

For task Memm-1, please provide instructions on how to run your code, in a `README` file in the `memm1` folder.

If your code cannot be run, you will receive a grade of 0.

Writeup

Your writeup should include the following:

1. Describe how you handled unknown words in hmm1.
2. Describe your pruning strategy in the Viterbi hmm.
3. Report your test scores when running the each tagger (hmm-greedy, hmm-viterbi, maxent-greedy, memm-viterbi) on each dataset. For the NER dataset, report token accuracy accuracy, as well as span precision, recall and F1.
4. Is there a difference in behavior between the hmm and maxent taggers? discuss.
5. Is there a difference in behavior between the datasets? discuss.
6. What will you change in the hmm tagger to improve accuracy on the named entities data?
7. What will you change in the memm tagger to improve accuracy on the named entities data, on top of what you already did?
8. Why are span scores lower than accuracy scores?