Assignment 1: Sequence Tagging

NLP-BIU | Spring 2022

Due: 5 April 2022

Goal: In this assignment you will implement two sequence taggers: a greedy tagger based on local classification, and a tagger that uses global search, using the *Viterbi* algorithm. The global search tagger will be based on the *trigram HMM model*, and the local classifier tagger will be based on a classifier of your choosing (we recommend a logistic-regression classifier). You will run the taggers on two datasets, *part-of-speech* (POS) tags and *named entity recognition* (NER).

Guidelines: In your submission, please follow these guidelines:

- General instructions for all assignments (follow this link).
- **Software:** Use Python 3 for this assignment.
- **Data:** The training and test files (tagged corpora) needed for this assignment are available on moodle.
- **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 will not grade your assignment.
- **Writeup:** In addition to the code, you should also submit a short writeup, as specified below. **Your writeup will be graded as well.**

At the time of publishing this assignment, we did not cover all the required materials in class, in particular, we did not cover the NER part. *But* we covered all the critical parts, and you can definitely start now. In fact, you are strongly advised to do so; things take time.

Resources: In this assignment we will be using two datasets:

1. A **part-of-speech tagging** dataset, based on newswire text
2. A **named-entity recognition** dataset.

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.

1 HMM Tagger

In this work, your main task is to implement an HMM tagger. When implementing the HMM tagger, there are two tasks:

(a) **Training:** computing the MLE estimates **q** (transition probabilities) and **e** (emission probabilities)

(b) **Prediction:** finding the best sequence of tagged 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 to this part is a tagged corpus. The output consists of two files:

- `e.mle` will contain the info needed for computing the estimates of **e**
- `q.mle` will contain the info needed for computing the estimates of **q**

The files will be in a human-readable format, as specified below.

What To Do: Your code should contain methods for computing the actual q and e quantities based on the corpus, and save the corpus-based quantities in *q.mle* and *e.mle* to files as specified below. For example, a method called $getQ(t1, t2, t3)$ 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.

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 $p(c)$. 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.

File Format: The format of *e.mle* and *q.mle* files should be as follows. Each line represents an event and a count, separated by a tab (marked here as `<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
```

Word-signature events in e.mle should start with a ^ sign (this is the "shift-6" sign on your keyboard), for example:

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

This indicates a signature for a word starting with an upper-case letter and continuing 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 ^.

What to submit: A program called `MLETrain` taking 3 command-line 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, and it will not be graded.

HMM task 2: Greedy decoding (5 pts)

What To Do: 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.

What To Submit: See "what to submit" section in task 3 below.

HMM task 3: Viterbi decoding (25 pts)

What To Do: Implement the Viterbi algorithm. Your Viterbi algorithm will use scores based on the **e** and **q** quantities from the previous tasks. 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` (for greedy tagging) taking 5 command line parameters: `input_file_name, q.mle, e.mle, greedy_hmm_output.txt, extra_file.txt`.
- A program called `HMMTag` (for viterbi tagging) taking 5 command line parameters: `input_file_name, q.mle, e.mle, viterbi_hmm_output.txt, extra_file.txt`.

Notes: `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 `input_file_name` that goes as input for `MLETrain`.

`extra_file.txt` 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 λ values for the interpolation. If you use the `extra_file` in your code, you need to submit it also together with your code, and use that name for this file.

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
```

Requirements

- 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**[1] **use a different tagset**, so make sure your program supports it.
- Your code should run in a reasonable time. If it takes more than 5 minutes to tag the dev set, something is wrong (and you will not be graded, meaning getting 0 points).

2 Local Feature-based Tagger

Like the HMM tagger, a feature-based tagger also has two parts: **Training** and **Prediction**. Like in the HMM, we will put these in different programs.

We advise you to use a classifier from python's `scikit-learn` package (`sklearn` for short), but you can also use other classifiers if you wish.

[1] We will.

Local Feature-based Tagger Task 1: Training (20 pts)

In this part you will write code that will extract training data from the training corpus in a format that the feature-based machine-learning trainer can understand, and then train a model. This will be a two-stage process.

Stage 1: Feature Extraction. Read in the train corpus, and produce a file of labels and features, 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 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`.

This should be implemented in a program called `ExtractFeatures` that takes in a corpus file, and produces a features file.

```
python3 ExtractFeatures.py corpus_file features_file
```

You can choose whatever features you want, but your final tagger's accuracy on the dev set of the POS task should be above 95%. You can find a decent set of features in Table 1 of this paper.

We advise you to write a function `extract(sent,i,...)` where `sent` is a list of tokens, `i` is a token index, and `...` are additional parameters you may need (for example, previously predicted tags) which returns a list of features for the ith word of a sentence, and use it when extracting features. You can later re-use this function in the prediction code.

Stage 2: Feature vectorization and training. Now, write a program that reads the output of the previous step (`features_file`) and trains a model. This entails reading lines of the features file, converting the features into vectors, and feeding them to a classifier trainer.

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 a vector format than can be fed to the model, use `DictVectorizer` class. Then, train a multi-class classifier based on the output of the `DictVectorizer`. You can train whatever classifier you want, but we recommend to train a `LogisticRegression` model (if you do so,

make sure to use the softmax classifier, and not the "one vs. rest" classifier).[2] Save the trained model to file.

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

```
python3 ExtractFeatures.py corpus_file features_file
```

```
python3 TrainModel.py features_file model_file
```

You can save the mapping between features and integers (produced by `DictVectorizer`) 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 seconds are names of output files. `model_file` is your trained model (**Do not submit it**)

What to submit: Submit the code for the programs in stages (1) and (2) (`ExtractFeatures.py` and `TrainModel.py`). 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)

Local Feature-based Tagger Task 2: Decoding (5 pts)

Implement a tagger that takes as input a model file (produced in the previous task) and un-tagged text, and produces a tagged file. Your code should produce the features for each sentence position, pass it to the DictVectorizer, and then to the prediction model.

The input and output format should be the same as the for the HMM tagger.

[2] Training this classifier requires fitting all features in memory, which may be challenging. If memory usage is an issue, you can: (1) use a sparse representation in DictVectorizer, and (2) use `SGDClassifier` with `loss='log'`, instead of using `LogisticRegression`. This model has a method `partial_fit()` that can be run on a subset of the dataset.

This program should be called `FeaturesTagger.py`, and it should take 4 commandline paramters: `input_file_name`, model_name, `feature_map_file`, `out_file_name`, in this order.

`input_file_name` is the name of an input file. The input is one sentence per line, and the tokens are separated by whitepsace. `model_name` is the names of the trained model file. `feature_map_file` is the name of the features-names-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`.

Your program should be run as:

```
python3 FeaturesTagger.py input_file_name model_file_name feature_map_file output_file
```

Requirements

- Your tagger should achieve a dev-set accuracy of at least 95% on the provided POS-tagging dataset.
- If your tagger runs for more than 10 minutes on the test or dev datasets, it will not be graded (it should not be hard to make it substantially faster).

Two ways to improve your accuracy are: (a) explore better / different features; (b) tune the hyper-parameters of the training, in particular the regularization parameter is important, and the learning rate can play a role as well.

What to submit:

- A program called `FeaturesTagger.py` taking 4 commandline parameters.
- A text file named `feats-predictions.txt` with your predictions on the test file, `ass1-tagger-test-input`.

3 Named Entity Recognition Tagger (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; we will use it for grading.

The span-based F scores are lower than the accuracy scores. Why? (write in the writeup).

What can you say about the accuracy of the HMM-tagger compared to the features-based tagger? (write in the writeup).

How far can you push the accuracy on the NER data? In particular, consider the features-based tagger. Maybe you can use better features? Or maybe the problem is with the greedy nature of the decoding?

For features inspiration, you can look at this paper. And maybe these lexicons can be of help?

If you think greedy-decoding is an issue, you can adapt the viterbi code to search over the classifier's score for each tag, rather than over the product of **q** and **e** estimates.

Do your best to improve the F_1 score on the NER data.

What to submit:

- 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 tagger, and numbers based on the Features-tagger (for the features tagger, use the best model you achieved). It should be clear which numbers are which.
- File named `ner.hmm.pred`, containing the predictions of your HMM model on the `test.blind` file.
- File named `ner.feats.pred`, containing the predictions of **your best features based model** on the **test.blind** file.

4 Writeup (10 points)

Along with your submission you should deliver a `writeup.pdf` file where you critically discuss your main experimental outcomes and insights, based on the sequences of items below. This file will be graded too, based on both the form and the content of the file.

Your writeup should include the following contents:

1. Describe how you handled unknown words in the HMM.
2. Describe your pruning strategy in the Viterbi HMM.
3. What is the runtime complexity of the trigram Tagger?
4. On each dataset, report your test scores when running each tagger (hmm-greedy, hmm-viterbi, feature-based-classifier)

5. For the NER dataset, report token accuracy *as well as* span precision, recall and F1.
6. Is there a difference in behavior between the greedy and viterbi taggers? Discuss.
7. Is there a difference in behavior between the datasets? Discuss.
8. Why are span scores lower than accuracy scores?
9. What would you change in the HMM tagger to improve accuracy on the named entities data?
10. How did you improve the accuracy of the NER tagger? what did you try? what worked? what didn't work? What does your best NER model include?

Submission Checklist:

In order for your assignment to be graded, please take care to comply with the following:

- Use python 3.x. You can use the packages as sklearn and numpy. If you want to use other packages that do not come pre-installed with python, please verify with the lecturers § first.
- The code should be able to run from the command line, 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`.
- Your code should run from the command line on a unix system, according to the instruction provided in each section. If your code cannot be run, for whatever reason, you will receive a grade of 0.

Good Luck!