Assignment 3 – POS Tagging on Irish and Welsh

CS499 - Intro to NLP
Antonis Anastasopoulos

February 2021

Due Date: 3/12/2021 (eod)
Submission instructions: see at the end of the assignment.

Make sure to download the assignment materials from the class page.

Part-of-Speech Tagging

Part-of-Speech (POS) Tagging is the task of annotating every token in a sentence with its grammatical category. For example, a tagging of English sentence ”the cat sat on the mat” would be `DET NOUN VERB PREP DET NOUN`, where `DET` stands for ‘determiner’ and `PREP` stands for ‘preposition’.

Although the proper categorization for some languages is still debatable, in this assignment we will work within the Universal Dependencies (`https://universaldependencies.org/`) formalism, which uses the Google Universal part-of-speech tags defined by Petrov et al.

Data We will work with the Irish (`https://universaldependencies.org/treebanks/ga_idt/index.html`) treebank (and the Welsh one (`https://universaldependencies.org/treebanks/cy_ccg/index.html`) in the Bonus question) from the Universal Dependencies project. The `data` directory of the assignment package includes training, development, and test files, following the following format:

- Each line corresponds to a sentence
- Each sentence is already tokenized and for each token the corresponding POS tag is provided in the following format: `word|tag`

The Irish train file includes 2019 sentences, the development and test sets have 451 and 454 respectively. The Welsh training file has 614 sentences, while the dev file has 500 sentences. The Welsh test set has 453 sentences but does not provide the correct POS tags.

Code The assignment package provides the following scripts:

- A utilities `utils.py` script with basic data reading functions that are used by the other scripts.
- A basic LSTM model `pytorch_tagging.py` implemented in PyTorch (based on this tutorial: `https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html`).
- A simple script to compute accuracy given two files in the above format. Run as: `python compute_accuracy.py [your_output] [gold_file]`.
- A python script `tagging_with_CRF.py` that uses the crfsuite that we used in the CRF class exercise.

Prerequisites You should be able to easily install PyTorch locally by following the instructions here: `https://pytorch.org/get-started/locally/`.

1 Neural Baseline Results on Irish (3 credits total)

The example script has toy data hard-coded into it. You can modify it to read the provided data in the desired format by doing something like the following:

```
TRAINING_FILE = "data/irish.train"
training_data = utils.convert_data_for_training(utils.read_data(
    TRAINING_FILE))
```

For this part, use the pre-defined hyperparameters as they are in the code!

[**1 credit**] **(a) Model Read/Write** The baseline script does not implement storing and loading the trained models. Implement this functionality, so that you don't have to train your models from scratch every time you run your script (and so that I can load your models and immediately obtain outputs when running your code).

[**1 credit**] **(b) Unknown Words** The baseline script has a very severe limitation: it can only handle input words that appear in the vocabulary (the dictionary `word_to_ix` in the script). However, there's no guarantee that the test set will only include these words! In fact, you can be sure that there are words in the test set that do not appear in the training set. One way to deal with this is to use an `UNK` token that we will use to represent rare words. The `utils.py` includes the skeleton for a `substitute_with_unk()` function, that you can use to modify the corpus. After training with the rare words substituted with `UNK`, we can easily use our model on the test set (as long as we also substitute the unknown words – the words that are not in our training vocabulary – with `UNK`). Implement this functionality, and train and evaluate an **Irish** POS tagging model using UNK for words that appear only once in the training set.

1. What accuracy does your model achieve on the Irish development set after training for 20 epochs?
2. What happens if you use two copies of the training set, one with UNKs and one without, as your training set?

You can evaluate how good your model is by running the `compute_accuracy.py` script as described above.

[1 credit] (c) Early Stopping The baseline script does not utilize the Irish development set. As a result, we face the unfortunate potential of *overfitting* on the training set, which can lead to much worse performance on the evaluation set. One way to combat overfitting is through *early stopping.* The idea is simple: while training, we will periodically[1] check the performance of our current model on the development set. If the accuracy is better than anything we have achieved before, then we keep training. If a few epochs pass and we still are not improving (e.g. after 8 epochs of no improvement. This is called "patience" in most toolkits) then we stop training and we use the checkpoint that achieved the best dev accuracy as our final model. Implement early stopping with a patience of 6 epochs.

What is the highest accuracy you achieve on the development set? For that model, what is the accuracy on the test set? After how many epochs did your model achieve its best performance?

Note: A good practice is to keep track of both the training loss and the development set loss. The training loss should generally always decrease. The development set loss will eventually plateau and then start increasing again, as you start overfitting. Visualizing the training process by plotting the losses is a great way to keep track of what's going on.

Hint: Without any additional modifications, and using an embedding and hidden dimension of 32 and training for 20 epochs, I got a test set accuracy of around 77% on Irish.

To submit: Code that implements the above three items, and any development set or test set outputs, under a directory named "part1". Make sure to answer all questions in your PDF report.

[1]e.g. after every epoch or after every 100 update steps or something like that.

2 Improving the Neural Baseline (5 credits total)

We will now improve upon this simple baseline for POS tagging in Irish:

[**2 credits**] **(a) Incorporate context from both sides** The current recurrent model only passes through the sequence in the left-to-right manner. This means that when making a prediction for position j, we have only taken into account the left context from previous positions $0 \dots j$. However, incorporating the right context from the positions $j+1 \dots$ can be very informative and aid the model in making a more informed prediction. This can be done by running another LSTM over the sequence in a right-to-left manner, and then combining the outputs of the two LSTMs. Implement this in your model by modifying the forward function, either by:

- making a copy of the input sequence, reversing it, passing it through a second LSTM, and then adding (or concatenating) the outputs of the left-to-right and right-to-left LSTMs, OR by
- taking a look into the LSTM PyTorch module (`https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`) which already implements it.

Note: the second option is very, very easy. I strongly suggest you try the first option first!

What accuracy do you get on the Irish development set by incorporating the right-hand-side context into your models?

To submit: code that implements the above, and any applicable dev/test set outputs, under a directory named "part2a". Make sure to answer all questions in your PDF report.

[**1 credit**] **(b) Optimizing training and hyper-parameters** The baseline model is very simple, and small. What happens if you increase your model's size? Try out the following, and report and changes in the Irish development set performance:

1. Stack two or more BiLSTM layers instead of using only one layer.
2. Change the size of the embeddings or the size of the hidden states of the LSTMs.
3. Substitute the simple SGD optimizer (line 80 in the original script) with the Adam optimizer (`https://pytorch.org/docs/stable/optim.html?highlight=adam#torch.optim.Adam`)

Note: no need to perform extensive hyper-parameter search; 2-3 different settings should be enough.

What is the best performance you achieve on the development set? What is your takeaway of the best hyper-parameter setting?

To submit: No need to submit specific code for this part. You can include your best Irish dev set outputs under a directory named "part2b". Discuss the changes you made to the model and the respective resulting development set accuracy in your PDF report.

[**2 credits**] **(c) Using pre-trained embeddings** Facebook has publicly released `fastText` embeddings (which incorporate the character n-grams we discussed above) that have been trained on large amounts of data. Details here: `https://fasttext.cc/docs/en/crawl-vectors.html`; the Irish (and Welsh) pre-trained embeddings can be downloaded in the same webpage. Use the fastText utilities to obtain vectors for all words in your train, dev, and test sets, including the out-of-vocabulary ones from the development and the test set (see instructions in the fasttext page on how to do that). Modify your code to read in these pre-trained embeddings, instead of using random initialization. Do you see any improvements in your dev and test set performance?

To submit: code that implements the above, and any applicable dev/test set outputs, under a directory named "part2c". Make sure to answer all questions in your PDF report.

3 Tagging with CRFs (2 credits total)

The code provided in the `tagging_with_CRF.py` script converts the words in every sentence to a bunch of features (through the `word2features()` function), and then uses these features to train a CRF to perform POS tagging.

[**0.5 credits**] **(a)** Run the code as is and report the accuracy on the Irish test set.

[**1.5 credits**] **(b)** The only features extracted by the provided code are the following:

- the word itself (lowercased)
- whether the word is uppercase
- whether the word is a digit
- the immediate previous word (lowercased)
- whether the immediate previous word is uppoercase

- whether the immediate previous word is the beginning of sentence (BOS)

Modify the `word2fearures()` function to also extract the following features:

- the immediate next word (lowercased)
- whether the immediate next word is uppoercase
- whether the immediate next word is the end of sentence (EOS)
- the last character of the word
- the last two characters of the word
- the last three characters of the word

What accuracy do you get now on the Irish test set?
To submit: code that implements the above, the Irish and Welsh dev and test set outputs, under a directory named ”part3”.

[**Bonus 1 credit**] **(c)** Add any other features you can think of, and see if you can further improve the model's accuracy. For instance, you can expand the features to include the word's prefixes, suffixes, or character n-grams. Or features that capture longer context like going beyond the immediate previous/next words, etc. Anyone who beats the best performance that Antonis got on the Irish test set (91.7%) gets the bonus credit. In your report, describe explicitly what additional features you added, and submit the code that implements them under a directory named “part3c”.

4 Bonus: Multilingual Models (2 (+1 or +2) credits total)

[**2 credit**] Irish and Welsh are related to each other. They both belong in the Celtic branch (“genus”) of the Indo-European language family. What happens if you train a multilingual model that can perform POS tagging on both languages? Does the performance improve on Irish? What is the accuracy on the Welsh dev set? Describe, in your report, which model you use and any other observations you think are worth discussing.

Hint 1: The simplest way is to concatenate the Irish and Welsh training data and train a single model.
Hint 2: Note the imbalance in training data sizes.

[**Bonus Bonus**] Tag the Welsh test set and write the output in a file named `welsh.test.hyp` following the same conventions as the other files (one sentence per line, and `word|tag` with your predicted tags for each word. We will compare your test set output against the gold file. The best performing submission for this exercise will receive an additional +2 credits, and the second best performing submission will receive an additional +1 credit.

To submit: code that implements the above, the Irish and Welsh dev and test set outputs, under a directory named ”part4”.

Submission Instructions

This assignment must be done individually. Discussing the assignment on the class forum is absolutely ok – posting the solutions is not. If you use someone else’s code (e.g. taken from a book or a website), make sure to properly cite it in your report.

Please submit all of the following in a gzipped tar archive (.tgz or .tar.gz; not .zip or .rar) via Blackboard. If you’re making a full submission, please name your file `gmuid-hw3.tgz` (for example, the instructor’s submission would be `antonis-hw3.tgz`). If you’re making a partial submission, please name your file `gmuid-hw3-part.tgz` where part is the part (1, 2, 3, or 4) that you’re submitting. Make sure that your .tgz file expands into a self-contained folder with your gmuid in it.

Your submission should contain:

- A PDF file (not .doc or .docx) with your responses to the instructions/questions above. The report should include your name and gmuid, and it should be self-sufficient (as in, we shouldn’t have to look elsewhere to grade you). If your name or an answer does not appear in the report, you will receive 0 credits for that part.

- All of the code that you wrote.

- A brief README file with instructions on how to build/run your code. We **will** run the code and check whether it produces all required outputs and if these outputs match the results in your report.