Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Alan Akbik

Berlin, 10.05.2024
Lehrstuhl Maschinelles Lernen

Introduction to Natural Langauge Processing

Exercise 3

Submission: Hand your homework until 20.05.2024 (Monday), 23:59 via Github Classroom. Each exercise is worth a total of **10 points** and you need **80%** (of total points over all exercises) to be admitted to the final exam. Sometimes, there are optional tasks that will give you extra points. Exercises in Github Classroom should be completed and submitted individually. However, we encourage you to work on the problems in teams. Please note the information available in Moodle: https://hu.berlin/nlp24-moodle.

Github Classroom: https://classroom.github.com/a/9ZkftzNQ

Task 0 (Update Python Environment) — 0 points

We observed an increasing number of questions in exercise 2. In the following lectures we want to avoid unclear instructions by providing your basic test cases to test your implementation. We use the `pytest` module to test your code.
You can install the package with:

```
# If you have followed the instructions from exercise 1:
conda activate nlpcourse
# Otherwise activate the virtual environment of your choice.

pip install pytest
# Or if you have already cloned the exercise repository:
# pip install -r requirements.txt
```

Task 1 (Bonus Task: Annotate Sentences with POS-Tags) — 5 points

Part-of-speech tagging (PoS-tagging), also called grammatical tagging, is the process of assigning a single word in a text to a particular part-of-speech class, based on both its definition and its context. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.
You can check the universal dependencies page for a full list of PoS-tags and their corresponding explanations.

(a) You are required to **annotate 5 sentences**. This exercise is done in Moodle https://moodle.hu-berlin.de/mod/quiz/view.php?id=4461590. The tags must be uppercase, you have unlimited attempts.

Task 2 (PoS Model) — 1 + 1 + 2 + 2 + 2 + 2 = 10 points

In this task, we are building a part-of-speech tagger in PyTorch. The goal of this model is to annotate each word in a sentence with its part-of-speech tag. The main difference to previous

exercises is that we now classifying each token instead of the entire sequence. However, the main ideas should be already familiar to you.
We provide you with tests to test all implementations in this task. You can find them in the `tests` directory. There you find a test script for each subtask (which are named accordingly). You can test your implementations with:

```
python -m pytest
```

(a) Complete the data loading function `read_pos_from_file` in `task_2/data_util.py`. As in previous exercises, you need to read-in a .txt-file. The format of the file is as follows:

```
so RB
what WP
happened VBD
? .
```

As you can see, each line consists of a word and a PoS-tag. Sentences are split by blank lines. The function should return a list of tuples with each tuple consisting out of two lists: one for the words in the sentence and one for the respective PoS-tags.

(b) Complete the function `make_tag_dictionary` in `task_2/data_util.py`. The inputs to the function are the generated tuples from previous task. The function should return a dictionary that has all PoS-tags as keys and a unique class ID as value (start counting at 0).

(c) Complete the function `prepare_train_sample` in `task_2/train.py`. The function creates vectors (which serve as your model inputs) for a given sentence and corresponding PoS tags.

The function takes several inputs: `tokens` (a list of strings), `pos_tags` (a list of strings), `unk_token` (a string), `vocab` (a dictionary that has strings as keys and integers as values) and `tag_dictionary` (a dictionary that has strings as keys and integers as values). You now need to create a vectorized representation of your `tokens` and `pos_tags` as in previous exercise by looking up the corresponding ID of each token and tag in the respective dictionary (`vocab` and `tag_dictionary`). If a word is not present in `vocab`, take the ID for `unk_token`. We add the `unk_token` for you to `vocab`, so just need to find the corresponding ID. Your function needs to return a tuple of lists with each list containing the IDs (integers) the respective word/tag.

(d) Train a simple part-of-speech tagger using your implementations of previous tasks. The class `SingleWordTagger` is already implemented. As described previously, the model now learns to assign each token of sentencen to a part-of-speech class. Perform a training run with and without pretrained embeddings.

1. Learning Rate: 0.3 (command line argument `--learning_rate X`)
2. Embedding Dimension: 50 (command line argument `--embedding_dim X`)
3. Train Epochs: 40 (command line argument `--epochs X`)
4. Pretrained Embeddings: [Yes/No] (command line argument `--pretrained_embedding`)
5. Seed: 42 (command line argument `--seed X`)

Check the argument parser in `run_task_2.py` for details. Example execution:

```
python run_task_2.py --lr 0.3 --pretrained_embedding
```

(e) Complete the `forward` function of `FixedContextWordTagger` in `task_2/tagger.py`. The class `SingleWordTagger` in the same file is already implemented and takes two lists of integers as input (what you return from previous task). Compared to previous exercises, we now classify each word instead of the sentence. You can see that we obtain the log-probabilities for each token for each possible PoS-tag.

As indicated in task 1, the PoS-tag may also depend on the surrounding context. Your task is now extend the representation of word by its surrounding context. To do so, you need to do the following steps:

1. Embed all tokens that are passed to the function (the function signature is identical to the one of `SingleWordTagger`) using the embedding layer (`self.embedding`).
2. For each token embeddings, compute the fixed context window size using `self.context_window`. For example, if the current word is at position $i = 3$ and $context\_window = 2$, then you would consider all embeddings at positions $[1, 2, 3, 4, 5]$. If the left context is negative or the right context is larger than the actual length of the sentence, use a zero-vector of `self.hidden_size`.
   a) Once you obtained your list of embeddings for the context window, use `torch.cat` to concatenate the embeddings into one vector.
   b) Pass the concatenated representation through the linear layer and obtain the log-probabilities for the token with `torch.nn.functional.log_softmax`.
3. Once you obtained log-probabilities for each token, concatenate them into one tensor. Hint: The concatenated tensor needs to be in shape: (length of tokens X number of PoS-tags)

Make sure you concatenate and apply the `log-softmax` along the correct dimensions.

(f) Execute 4 trainings runs (with a context window of 1 and 2, with and without pretrained embedding). Other hyperparameters should be identical to task d). You have a command line argument to directly use your context implementation:

1. Context Size: [1/2] (command line argument `--context_size X`)
2. Pretrained Embeddings: [Yes/No] (command line argument `--pretrained_embedding`)

Example execution:

```
python run_task_2.py --pretrained_embedding --context_size 2
```

Compare these results with the ones not using context. Create a final result file named `result.csv` in csv-format that in the following format:

```
context_size,pretrained_embedding,accuracy
<either 0,1 or 2>,<either yes or no>,<accuracy from that run>
1,yes,0.5
...
```

The resulting csv-file should contain results from **6 runs** (2 runs in task d) and 4 runs in task f)).

Further, each run will log a checkpoint and plots into the `results` folder. Include this folder in your submission as we use the final checkpoint to evaluate your implementation. You can perform more runs if desired but **do not commit more than 20 runs**. If you have done more, please delete run directories but make sure you include the configurations required for this task.

Task 3 (Extra Fun: Try out your implementation) **0 points**

You can load a trained model using `model_interaction.py` using:

```
python model_interaction.py --results_path results/0001
```

It will ask you to enter some example sentence and it will give you the annotated sentence back!