HW2: Command-line design

In this assignment, you'll incrementally build up a simple Python command-line tool. Please note that for this homework assignment, you will be graded on the final tool and your associated write-up; you only need to turn in intermediate steps if you are unable to complete step #3.

Goal

While there are many formats used for tagging data, one of the most common is multi-column tabular format with a blank line between each sentence. For instance, here are two sentences as they appear in the CoNLL-2000 shared task on chunking (Tjong Kim Sang & Buccholz 2000):

```
The DT B-NP
August NNP I-NP
deficit NN I-NP
and CC O
the DT B-NP
# # I-NP
2.2 CD I-NP
billion CD I-NP
gap NN I-NP
registered VBN B-VP
in IN B-PP
July NNP B-NP
are VBP B-VP
topped VBN I-VP
only RB B-ADVP
by IN B-PP
the DT B-NP
# # I-NP
2.3 CD I-NP
billion CD I-NP
deficit NN I-NP
of IN B-PP
October NNP B-NP
1988 CD I-NP
. . O

Sanjay NNP B-NP
Joshi NNP I-NP
, , O
European JJ B-NP
economist NN I-NP
at IN B-PP
```

```
Baring NNP B-NP
Brothers NNPS I-NP
& CC I-NP
Co. NNP I-NP
, , O
said VBD B-VP
there EX B-NP
is VBZ B-VP
no DT B-NP
sign NN I-NP
that IN B-SBAR
Britain NNP B-NP
's POS B-NP
manufacturing NN I-NP
industry NN I-NP
is VBZ B-VP
transforming VBG I-VP
itself PRP B-NP
to TO B-VP
boost VB I-VP
exports NNS B-NP
. . O
```

Here the first column is the token, the second is a hypothesized POS tag, and the third is the chunk tag. In Python we might represent

- each row as a list or tuple (here, we choose the former),
- each sentence as a list of such lists, and
- the entire corpus as a list of sentences.

In this assignment, you will incrementally develop a Python command-line tool which automatically reads in tagging data in the above format and randomly splits it into training, development, and test data, writing the resulting "splits" to new text files.

Step 1: parsing

For data in the above format, we can read the data one sentence at a time using the following generator function.

```python
from typing import Iterator, List


def read_tags(path: str) -> Iterator[List[List[str]]]:
    with open(path, "r") as source:
        lines = []
        for line in source:
            line = line.rstrip()
            if line:  # Line is contentful.
                lines.append(line.split())
            else:  # Line is blank.
                yield lines.copy()
                lines.clear()
    # Just in case someone forgets to put a blank line at the end...
    if lines:
        yield lines
```

Then, to read the entire corpus at once, then, we can do the following, for instance:

```
corpus = list(read_tags("conll2000.txt"))
```

What to do

Using the `argparse` module, build a Python command-line tool which takes four arguments: `input`, `train`, `dev`, and `test`. The program should:

1. Read all of the `input` data in using the above snippet.
2. Split the data into an 80% training set, 10% development set, and 10% test set.
3. Write the training set to the `train` path.
4. Write the develoment set to the `dev` path.
5. Write the testing set to the `test` path.

The resulting training, development, and testing set files should be in the same columnar format as the input format.

Hints

- Read the `argparse` docs before you get started.
- The skeleton of a program is provided in `split.py`.
- Do not hard-code any of the paths.
- You may have to mark your code executable, which you can do by running `chmod +x split.py` from the command-line.
- To stay DRY, you may want to write a function `write_tags` which writes the data out.

Testing

1. Confirm that the following works as expected:

```
$ ./split.py conll2000.tag train.tag dev.tag test.tag
```

2. Manually check that the file lengths are correct using the `wc` command-line tool:

```
$ wc -l train.tag dev.tag test.tag
```

Note that your exact lengths may differ slightly (because the sentence lengths differ slightly) but that `dev.tag` and `test.tag` should be roughly the same lengths, and both should be roughly 8x shorter than `train.tag`.

Step 2: Randomization

Above, we did not specify how the data was split. Now, we will modify it so that the division is pseudo-random.

What to do

Modify `split.py` from the previous step so that the split into training, development, and testing data is pseudo-random. The easiest way to do this is to reuse the code from the previous step, but randomly shuffle the list of sentences using `random.shuffle`.

Hints

- Read the `random` docs before you get started.
- Note that `random.shuffle` **works in-place**.

Step 3: Seeding

When Python loads the `random` module, it automatically "seeds" the pseudo-random number generator (PRNG) with the current wall clock time. Thus every time one runs `./split.py`, a different split is obtained. This makes it extremely difficult to exactly replicate a workflow that involves random number generation. One solution involves having the user provide a seed (an integer, usually) to the PRNG before it is used. So long as the user keeps a record of the seed used (or uses a memorable number, like a birthdate or an address) anyone can replicate the "random" behavior in the future.

What to do

Add a `--seed` flag to your script, and use this seed to seed the PRNG before randomizing the data. Mark the flag mandatory in the `argparse` declaration so that a user must specify some seed.

Testing

1. Confirm that the following succeeds:

```
$ ./split.py --seed=272 conll2000.tag train.tag dev.tag test.tag
```

2. Confirm that your script prints out an informative error message if the user omits the `--seed` flag. E.g.:

```
$ ./split.py conll2000.tag train.tag dev.tag test.tag
usage: split.py [-h] --seed SEED input train dev test
split.py: error: the following arguments are required: --seed
```

3. Run your script over the input file producing `train.tag,` `dev.tag`, and `test.tag` using a fixed seed. Then, compute their SHA-256 checksum using a tool like the `shasum` command-line tool:

```
$ shasum -a256 train.tag dev.tag test.tag
```

Then, run the script over the input file *again*, using the same seed as the previous time, and confirm that the SHA-256 checksums have not changed.

Hints

- Read the `random` docs before you get started.
- There are in fact two ways to accomplish this, either fine:
  - seeding the global random number generator.
  - create a `random.Random` object, seed it, and use it for shuffling.

Stretch goals

(Recall that these are optional, not required.)

1. Print out the number of sentences and tokens in each of the output files in some visually-appealing format using f-strings.
2. Do #1, but instead of printing it out, use Python's built-in logger and log the information at `INFO` level. This information will not show up when you run the command unless you set the log level to `INFO` or lower; do so inside the `if __name__ == "__main__":` block, immediately before you parse the command-line arguments, or allow the user to enable a verbose mode via an optional, boolean `-v` flag.
3. Install the `black` code reformatter and reflow your code using `black -l79`.
4. Install the `flake8` linter and make your code pass its checks.
5. Add type signatures to your functions. Then install the `mypy` static type checking tool and make your code pass its checks.

What to turn in

1. Your final `./split.py` as per Step 3.
2. A one-page write-up, in PDF form, detailing any challenges you experienced and how you dealt with them.

References

Tjong Kim San, E. F. and Buchholz, S. 2000. Introduction to the CoNLL-2000 shared task: chunking. In *Fourth Conference on Computational Natural Language Learning and the Second Learning Language in Logic Workshop*, pages 127-132.