Programming for LT Applications

Artur Kulmizev

March 3, 2021

This assignment package will be rolled out in three parts, each corresponding to a lab section: **Web Scraping**, **Pre-processing and Visualization**, and **Modular Programming**. This document will be updated with each part accordingly, which will have separate exercises associated with it. The goal of this assignment is to get to you to write practical applications in Python, using the concepts we covered in the lectures. You should hand in a single tarball[1], named according to the following format: `assignment4.tar.gz`. The directory structure of the tarball should be the following:

```
A3_0_assignment_3
├── twitter_query
│   ├── README.txt
│   └── twitter_query.py
├── wiki_query
│   ├── README.txt
│   └── wiki_query.py
├── term_plotter
│   ├── README.txt
│   └── term_plotter.py
└── tagger
    ├── tagger
    │   ├── __init__.py
    │   ├── pre_process.py
    │   └── model.py
    ├── README.txt
    ├── run.py
    └── config.yaml
```

In other words, the examiner should easily be able to navigate to a particular exercise folder, and call the exercise script directly via e.g.

```
$ python script.py ARG1 ARG2
```

This code should run out of the box[2] and produce the desired results. You can approach the problems in any way that you want (using classes, functions, etc.), but the code has to be executable from the corresponding exercise script file and return the desired output. You should be also mindful in making your code clean and interpretable, commenting where necessary. The `PEP 8` style guide[3] can give you some tips in this regard, but you are by no means required to use it. Each exercise folder should also include a `README.txt` file that explains how to run your code, as well as how you went about writing it (e.g. did you make a class?).

[1] https://www.howtogeek.com/362203/what-is-a-tar.gz-file-and-how-do-i-open-it/

[2] We shouldn't get errors when running a simple query.

[3] https://www.python.org/dev/peps/pep-0008/

Lab 1: Web Scraping

Finding good data is a crucial first step in many NLP applications. When ready-made corpora are not available, we often revert to scraping relevant texts from the internet. In the web scraping lecture, we learned a bit about web scraping: how websites are built upon HTML files, how we can parse these files to extract information, among other things. If you recall, we discussed two distinct scenarios that might arise when scraping the web. In the first scenario, you might have access to a website's API, which you can use to interact with the website directly. Otherwise, you might have to do the hard work yourself: understand the structure of the website data and parse it accordingly. We'll explore these situations in this lab, where you'll build two tools: one for extracting Tweets for a specified user on Twitter, and another for extracting the descriptions of concepts from Wikipedia.

Extracting Tweets

Your task: Write a program that takes a Twitter user's handle as an argument and returns the user's most recent Tweet(s), as well as the time of their posting. Your script should also accept an optional integer N, which, if specified, returns the user's N-most recent Tweets. If not specified, it should return only the single most recent Tweet. For example:

```
$ python3 twitter_query.py jack 3
11:28 PM, Feb 21, 2020: I've discovered the way to make slack grind
to a crawl and crash repeatedly
3:29 AM,  Feb 21, 2020: Talked #bitcoin Africa with #NiiQuaynor,
@PindarWong, and @moneyball! Earth globe europe-africa
11:25 PM, Feb 20, 2020: I just talked with @leslieberland's mom
on the phone AMA
```

You should work with the `Tweepy`[4] library for navigating the Twitter API. In order to do so, you will need a set of credentials: a consumer key and secret, and an access token and access token secret. These can be accessed via a JSON file found at `/local/kurs/advprog/module3/credentials.json`. You can decide how to handle replies, retweets, images, emoji, etc., but make sure to document your decision in doing so. Your script should be called `twitter_query.py`.

Mining Wikipedia

Your task: Write a program that takes a search term as an argument and then prints the content of the Wikipedia page for that term. For example:

```
$ python3 wiki_query.py ankylosaurus
Ankylosaurus is a genus of armored dinosaur. Its fossils have been
found in geological formations dating to the very end of the Cretaceous
Period, about 68-66 million years ago, in western North America,
making it among the last of the non-avian dinosaurs. It was named by
Barnum Brown in 1908, ...}
```

Make sure that your program uses at least one `try-except` clause (you can choose which exceptions to try to catch). Also ensure that you can search for terms with non-`ASCII` characters. For example, check that your program works with the search term

[4] http://docs.tweepy.org/en/latest/index.html

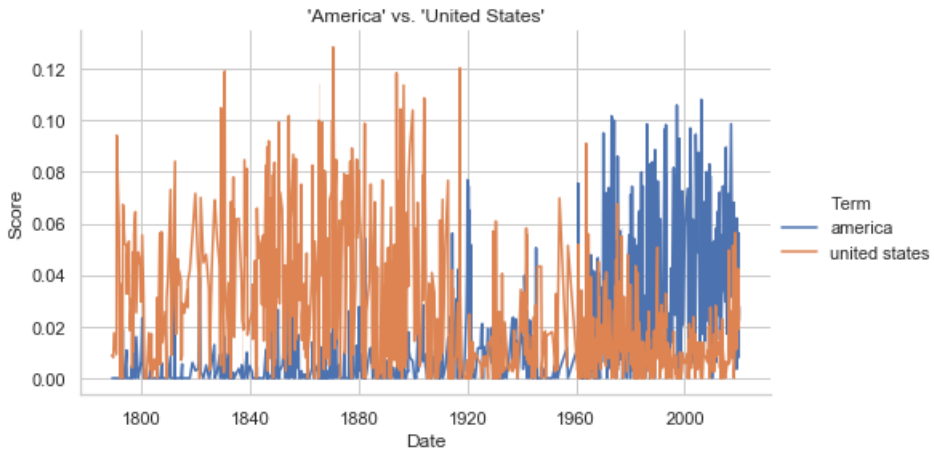
"Miloš_Zeman". You might want to use the libraries `urllib` and `beautifulsoup` (see the lecture notebook and online documentation), but feel free to approach the problem however you want. Your script should be called `wiki_query.py`.

Lab 2: Pre-processing and Visualization

After collecting raw text data, it is important to know how to process it. This can involve anything from tokenization to dependency parsing. We learned how to do this in the lecture, using a power library called SpaCy. Once your data has been processed, you can begin to gather some insights about how it's structured and what it represents. Visualization is a key component of this process, and we covered how to generate basic scatter, line, and bar plots in Python. In this lab, you will be asked to generate a line plot that takes all documents in our presidential speech corpus as input and plots the tf-idf scores of specified terms, sorted by year. This should give a sense of how particular terms/concepts came to prominence throughout the history of the US (as told by its presidents).

Diachronic Term Relevance

Your task: Write a program that calculates the tf-idf scores of a set of specified terms as they occur in each of our scraped presidential speeches and plots each term's score as a function of time (i.e. the speech's date) Your program should output an image that looks something like this:



You should use `argparse` for this assignment, where you set the arguments as follows:

- `terms:` a **list** of up to five terms, in quotations, separated by white space. Each "term" can consist of three tokens (e.g a trigram), with English stopwords assumed to be filtered out.
- `path:` an optional **string** argument that specifies where the speech directory is. If not provided, the program should assume to look at the folder in the current directory, e.g.: `./us_presidential_speeches`.
- `title:` an optional **string** argument that adds a title to the plot. If not provided, the program outputs the plot without a title.
- `output:` an optional **string** argument that saves the plot under the specified name. If not provided, saves the plot as the concatenation of all terms, separated by an underscore token. e.g.: `america_united_states.png`.

Ultimately, in order to produce the output shown here, the examiner should be able to call the script as follows:

```
$ python3 term_plotter.py --terms "america" "united states" \
                          --title "'America' vs. 'United States'"
```

This will look in the current directory for a folder called `us_presidential_speeches/` and generate a file called `america_united_states.png`.

To complete this assignment, you will make use of several libraries, including `numpy`, `pandas`, `seaborn`, and `scikit-learn`. To obtain the tf-idf score of a given word, as it occurs in a speech, you will need to fit `scikit-learn`'s `TfidfVectorizer` on a list of speeches, using `fit_transform()`[5]. Using the fitted vectorizer, you will need to extract the tf-idf score of each specified term as it occurs in each speech. Ultimately, you should aim to populate a `pandas DataFrame` that looks like the following:

```
          Date     Score           Term
0   1841-09-09  0.000000        america
1   1945-08-06  0.000000        america
2   1991-02-27  0.026612        america
..         ...       ...            ...
992 1885-12-08  0.068363  united states
993 1963-11-27  0.012194  united states
994 1869-05-19  0.069120  united states

[1990 rows x 3 columns]
```

You can then pass this `DataFrame` to `seaborn` in order to generate the desired plot.

When working on the assignment, be mindful of the possible exceptions you can catch. For example, what happens if the directory passed to the `path` argument does not exist? Also, what if one or more of the terms passed to the `terms` argument are not in the speech vocabulary? What if none of them are? Instead of returning errors, you should instead aim to deal with these exceptions accordingly, letting the user know what the (potential) problem is and how it can be fixed. In your final submission, these cases need to be addressed, but it cannot hurt to include other types of exceptions as well.

Lab 3: Modular Programming and scikit-learn

By this point, you should have a broad idea about what sorts of libraries and toolkits are at your disposal as a language technologist. Taking libraries like `BeautifulSoup`, `spacy`, and `scikit-learn` into account, you should be well equipped to start writing your own NLP pipelines: from pre-processing to training a model to classifying new data. In the lecture, we walked through the process of doing this for a Part-of-Speech tagger. However, we left some parts of our code incomplete. Your task for this lab will be to fill in these gaps with added functionality. You can find the tagger code at `/local/kurs/advprog/module3/tagger`.

Document Tagging

Your task: Include functionality that allows for tagging an entire `.txt` file, instead of a single sentence like `model.POSTagger.tag_sentence()` does now. You should still be able to call the tagging mode via the following command:

[5] note that, when fitting the vectorizer, you'll need to 1) account for unigrams, bigrams, and trigrams occurring in your vocabulary, and 2) filter out English stopwords.

```
$ python3 run.py --mode tag --text my_file.txt --config config.yaml
```

The difference here is that your code should recognize that the option passed to `-text` is a file. If it is[6], then you should read the file, tokenize it, and tag it using the functions/methods you have available to you. The file should be assumed to a standard raw text file, like the one in `/local/kurs/advprog/module3/tagger/files/paradise.txt` (feel free to use this if you want). After tagging, you should output the tokens and predicted tags in a separate text file in the ***same directory*** as the input file, with the suffix `.tag` (e.g. `my_file.txt` → `my_file.txt.tag`). The output should be identical to the tab-separated `.txt` format we saw in the lecture. If the option passed to `-text` is a sentence string — not a file — you should proceed to tag it using `model.POSTagger.tag_sentence()` as before.

> **NOTE**: when tagging unseen text, your `run.py` script will look for a pretrained model `.pickle` file, which is specified in `config.yaml`. If it cannot find this file, it will throw an error. This means you have to train a model first, by calling `-mode train`. The script will then look for the training file in `config.yaml` and save a model trained on that data.

Evaluation

So far, we only have an indication of how well our model performs via cross-validation on the training set. If we'd like to compare our model to an objective accuracy metric, we'll need to see how it fares when tagging a held-out test set.

Your task: Write a new mode called `eval` that accepts a conditional argument called `-gold`, which points to a gold-standard test (or dev) file in either `.txt` or `.conllu` format. You should load this file, tag it using a saved model (specified in the config), and compare your results with the gold tags. Every `scikit-learn` model (including the `LinearSVC`) has methods that do this, so you should refer to the API for your chosen model to figure out exactly how. When finished tagging, you should print the results you obtained, using `sklearn.metrics.classification_report`. This will return the precision, recall, and f1-score for every tag in the tagset. The examiner should be able to call the script and obtain approximately the following output:

```
$ python3 run.py --mode eval --gold my_gold_file.txt --config config.yaml


Evaluating model on ./en_ewt-ud-test.conllu.
Loading model from ./svm_tagger.pickle.
              precision    recall  f1-score   support

         ADJ       0.90      0.89      0.90      1692
         ADP       0.93      0.96      0.95      2019
         ADV       0.92      0.88      0.90      1226
         AUX       0.96      0.97      0.97      1503
       CCONJ       0.99      0.99      0.99       739
         DET       0.99      0.99      0.99      1896
        INTJ       0.97      0.82      0.89       120
        NOUN       0.89      0.91      0.90      4133
         NUM       0.92      0.93      0.93       536
```

[6] You can check this using methods defined in `os.path`.

```
         PART       0.94      0.98      0.96       630
         PRON       0.98      0.98      0.98      2158
        PROPN       0.87      0.83      0.85      2075
        PUNCT       0.99      1.00      1.00      3106
        SCONJ       0.82      0.80      0.81       386
          SYM       0.99      0.79      0.88        92
         VERB       0.93      0.94      0.94      2647
            X       0.87      0.59      0.70       139

    micro avg       0.94      0.94      0.94     25097
    macro avg       0.93      0.90      0.91     25097
weighted avg       0.93      0.94      0.93     25097
```

> **NOTE**: You won't need to worry about tokenization using `SpaCy` here, since the gold file is already assumed to be tokenized-per-line. Simply load the file as you normally would and process it using the functions in the `pre_process` module.

Feature engineering

Your task: Try to obtain a higher accuracy on the `en_ewt` test set. Experiment with other features than the ones we've included `pre_process.token_to_features()`. Try out different models that are implemented in `scikit-learn`. You can find a list of potential models/methods here: `https://scikit-learn.org/stable/supervised_learning.html`. If you're feeling especially inspired, you can play around with `sklearn-crfsuite`, an extension of `scikit-learn` that features several implementations of Conditional Random Fields: a powerful class of sequence models. Whatever you settle on, make sure your changes are reflected in the codebase that you end up submitting. Make sure to also document everything in the `README.txt` file, including how to run your code, what sort of features you settled on, and what kind of model you employ. Don't write the `README.txt` with me (Artur) as the audience, but rather someone that is interested in using your code for research or otherwise.