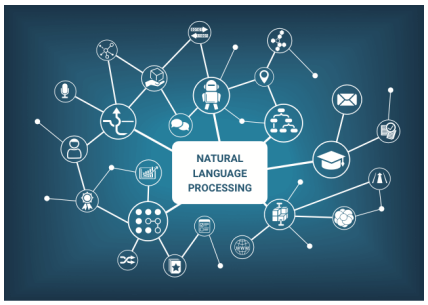

A reusable extensible framework for Natural Language Processing

DS 3500 / Prof. Rachlin

Overview

As we progress into advanced programming and begin to embrace attitudes of professional software developers and data scientists, our focus turns more to reusability, code maintainability, and extensibility. Reflecting on DS2500 homework assignment #2 *(Presidents and Poets*), those of you who took that class would remember processing six or so text files and producing a variety of custom visualizations. Some of you explored the topic further by analyzing other kinds of text files for your class project. But probably you weren't thinking about re-using your code for a future class, or what re-usable code even means! Imagine if you had a library would have enabled you to carry out the assignment in just a handful of lines of code – wouldn't that have been nice! In this class we'll build just such a library, and you will demonstrate its reusability by comparing a set of documents of your own choosing.

Library Requirements

1. The library should be implemented as the object instance of a class. You are free to import other libraries to facilitate the implementation of your library methods.

2. You should be able to *load* or *register* an arbitrary number of related text files. You may choose the text files, but they should be related in some way (see below for some suggestions.)

3. Each text file that you load should be pre-processed so that the pre-processor cleans the data, removing unnecessary whitespace, punctuation, and capitalization. You may want the pre-processor to gather some statistics such as average sentence or word length, readability scores, sentiment, and so on, but at minimum you should include a word-count as this will support one of the required visualizations described below.

4. A generic parser / pre-processor may be sufficient, but what if your file format is not very generic? When registering the file, you can pass a custom parsing function that will carry out the parsing and pre-processing of your unique files! You should implement support for this even if you might not need it for your documents. We'll discuss this more in class.

5. The library will support three visualizations, one of which is very specific, the other two are left at your discretion.

    a. (Specific) Text-to-Word Sankey diagram. Given the loaded texts and either a set of user-defined words OR the set of works drawn from the *k* most common words of each text file, generate a Sankey diagram from text name to word, where the thickness of the connection represents the wordcount of that word in the specified text.

    b. (Flexible) Any type of visualization containing sub-plots, one *sub-plot* for each text file.

    c. (Flexible) Any type of comparative visualization that overlays information from each text file onto a *single* visualization.

Library Architecture

I recommend that your library be given a useful name. (Imagine trying to share your library on GitHub but you called it "hw2" – who's going to want to download that?)

At minimum, you should implement the following methods:

- **load_text(self, filename, label="", parser=None)**
  # Register a text file with the library. The label is an optional label you'll use in your
  # visualizations to identify the text

- **load_stop_words(stopfile)**
  # A list of common or stop words. These get filtered from each file automatically

- **wordcount_sankey(self, word_list=None, k=5)**
  # Map each text to words using a Sankey diagram, where the thickness of the line
  # is the number of times that word occurs in the text. Users can specify a particular
  # set of words, or the words can be the union of the *k* most common words across
  # each text file (excluding stop words).

- ***your_second_visualization(self, misc_parameters)***
  # What you do here is up to you, but it should produce a single visualization with one
  # subplot for each text file. Rendering subplots is a good, advanced skill to know!

- ***your_third_visualization(self, misc_parameters)***
  # This is also left up to you, but it should be a single visualization that overlays data from
  # each of the text files. Make sure your visualization distinguishes the data from each
  text file using labels or a legend.

As you pre-process each text file, you'll want to store intermediate results: word counts, statistics, maybe the clean version of text, etc. A standard way to do this might be to have a state variable: **self.data** that is a dictionary. Presented below is a generic solution using dictionaries within dictionaries. The layout and content are ultimately up to you but try to come up with an approach that makes it easy to add custom parsers that store other kinds of information and custom visualizations that can access this information in standard ways.

```
{
        word_count: {text1: wordcount1, text2: wordcount2, …. }
        word_length: {text1, wordlength1, text2:wordlength2, … }
        sentiment: {text1, sentiment1, text2:sentiment2, … }
        .
        .
        .
}
```

Data Sources

Choose at least three related text files. The type of file you choose is entirely up to you, but I recommend something different than presidential inaugural speeches! Some possible ideas or data sources:

- Project Gutenberg (www.gutenberg.org)
- Political speeches
- National constitutions
- Tweet compilations
- Corporate filings
- Letters/Journals/Diaries, for example:
  https://content.lib.washington.edu/civilwarweb/index.html
  (It would be interesting to compare letters from soldiers fighting for the North vs South!)

- News articles or blog posts

- Religious texts
- Philosophical tracts
- Song lyrics

Assignment Guidelines

Your submission should include a 1 to 3-page report in **PDF** format summarizing your data sources, insights, and conclusions.

You may work in groups of 1 to 4. For group submissions, one submission per group is sufficient. Please document the contributions of each team member to the code base in your final report.

Code for this mini-project should be managed on the NEU Enterprise Git Server. Grant Professor Rachlin and the three TA's access or make the project public. Even though the code will live in a repo, please also submit your code files and project report to Gradescope so that the TAs can comment on specific lines of code.

We'll have in-class mini-presentations to share our work. Each group should prepare a ONE SLIDE POSTER (PDF version to be submitted to GradeScope).

What to Submit

- Code (.py)
- 1 to 3-page report, including author contributions, visualizations, commentary and a link to your code repository (.pdf)
- 1 slide poster for the in-class mini-presentation (.pdf)