**“Word Clouds in Python”**

Daniel Klein

Computer-Based Honors Program

The University of Alabama

Fall 2013

**Project Description:**

This project is in under the direction of Dr. Joseph Smith, Associate Professor of Political Science at the University of Alabama. We aim to build a piece of software, written in Python, that will perform automated content analysis on a collection of legal documents and create a statistical word cloud illustrating the terms that characterize the collection.

We are approaching the research objective with a highly object-oriented design that will be built incrementally though primarily test-driven development. Final delivery of the software is scheduled for early December 2013. The GitHub repository for the project is viewable at <https://github.com/dmarklein/WordCloud>.

We presented our work at the 2014 Southern Political Science Association Conference in New Orleans, Louisiana, and we will also be presenting our work at the 2014 Undergraduate Research & Creative Activity Conference at the University of Alabama in Tuscaloosa, Alabama.

The basic idea for the flow of the project is this:

1. Each document has a file to itself. The DocumentConverter parses each file and creates a Document object from each one.
2. Given a group of these Document objects, the DocumentSorter creates subsets of them (by sorting on a given metadata field).
3. Given a collection of one or more subsets of Documents, the AnalysisEngine performs the actual statistical analysis on term frequency and whatnot and creates a list of (term, weight) tuples representing the most important terms in each subset, which it passes to the WordCloudGenerator.
4. The WordCloudGenerator has the easy part: it takes the list of terms and weights and creates a visualization word cloud.

**SupremeCourtOpinionParser Class**

**Description:** This script, given a directory containing text files of Supreme Court cases from Lexis, will parse each file in the directory to extract the cases, then parse each case to extract the opinions therein. The script will then write each opinion to file, along with key information about the corresponding case. This class is a rewrite of Dr. Smith’s script for parsing opinions out of Lexis files – the underlying ideas and algorithms are largely the same. I merely translated the script into a more modularized, object-oriented form.

**Init params:**

* output\_dir

**Data members:**

* output\_dir
* majority\_dir
* dissent\_dir
* concur\_dir
* concur\_dissent\_dir

**Methods:**

* parse\_file(source\_file\_path)
* parse\_case(case)
* parse\_alt\_opinions(alt\_opinions, case\_header)
* get\_delimiter()
* remove\_bracket\_nums()
* split\_into\_cases(delimiter)
* categorize\_opinons(split\_alt\_opinions)
* get\_author(opinion)
* get\_info(case\_header)
* various “get\_foo” helper methods
* write\_opinion(opinion\_with\_type, opinion\_author, case\_header)

**Document Class**

**Description:** This class represents a document object (such as a single Supreme Court opinion). A typical object will consist of the document’s text and a metadata object consisting of various fields relevant to the document. We will subclass this class for each type of document we will use.

**Init params:**

* doc\_metadata
* doc\_text
* output\_filename

**Data members:**

* doc\_text
* doc\_metadata (Metadata object)
* output\_filename
* word\_count (how many words are in the text of the document)

**Methods:**

* count\_words(text) (performs basic wordcount on text)
* write\_to\_file() (basically, pickle; write the Document, properly formatted, to file)
* print\_doc() (display the Document with its metadata)
* print\_metadata() (display the Document’s metadata)
* \_\_str\_\_() (overloaded method)

**Test Cases:**

* Verify serialization/pickle (write\_to\_file())
* output file not writable
* Verify \_\_str\_\_
* Verify count\_words
* Verify print methods

**DocumentStorage Class**

**Description:** This class inherits from the Document class; an object of this class is essentially a Document object along with storage for a couple statistical metrics for the terms contained in the text of the document.

**Init params:**

* doc\_metadata
* doc\_text
* output\_filename

**Data members:**

* doc\_text (this is processed from its original form – punctuation and stop words are removed, numbers and single-letter words are removed, and words are stemmed)
* doc\_metadata (Metadata object)
* output\_filename
* word\_count (how many words are in the text of the document)
* split\_text (the text of the document in a list)
* stemmed\_text (text of doc in a list, with each term stemmed)
* term\_list (a list of all the terms used in the document, relative term frequency for each term)

**Methods:**

* all methods inherited from the Document class
* create\_split\_text(text)
* build\_term\_list(split\_text)
* populate\_term\_freqs(term\_list)
* filter\_text(text, should\_drop\_prop\_nouns=False)
* remove\_proper\_nouns(text)
* remove\_punctuation(text)
* remove\_nums(text)
* remove\_short\_words(text)
* remove\_footnotes(text) – this is not currently used
* remove\_stop\_words(word\_list)
* stem\_text(word\_list)
* calculate\_term\_frequency(term)
* calc\_tfidf(term, doc\_freq)

**Test Cases:**

**Metadata**

**Description:** A metadata object will be a collection of fields (pieces of metadata) that accompany of piece of data (in our case, the text of a document). We will subclass this class for each type of document we use.

**Init params:**

**Data members:**

* various fields – will be different for different Metadata subclasses

**Methods:**

* print\_fields() (print the fields within the Metadata class)
* print\_metadata() (print fields and their values)
* \_\_str\_\_() (overloaded method)

**Test Cases:**

* Verify \_\_str\_\_
* Verify print\_fields

**SupremeCourtOpinionMetadata**

**Description:** This is a specialization of the Metadata class, containing data attributes specific to a Supreme Court opinion.

**Init params:**

**Data members:**

* field\_names
* case\_title
* case\_num
* case\_us\_cite
* case\_supreme\_court\_cite
* case\_lawyers\_ed\_cite
* case\_lexis\_cite
* case\_full\_cite
* case\_dates (list of (date, action) tuples)
* case\_disposition
* opinion\_author
* opinion\_type

**Methods:**

All methods for SupremeCourtOpinionMetadata are inherited from the generic Metadata class.

**SuperDoc Class**

**Description:** A SuperDoc object will consist of a piece of text that represents the concatenation of the text fields from 1+ documents, along with a list of metadata objects – each item in this list will be the metadata from one of the documents that make of the SuperDoc. This class is no longer used in the system.

**Init params:**

**Data members:**

* component\_metadata (list of Metadata objects) [this is a bad var name]
* superdoc\_text
* output\_filename
* word\_count

**Methods:**

* count\_words(text) (performs basic wordcount on text)
* write\_to\_file() (basically, pickle; write the SuperDoc, properly formatted, to a file)
* print\_superdoc() (display the SuperDoc, formatted)
* print\_component\_metadata() (display the component metadata, formatted)
* \_\_str\_\_() (overloaded method)

**Test Cases:**

* verify serialization/pickle (write\_to\_file())
* output file not writable
* verify \_\_str\_\_
* verify count\_words

**SuperDocGenerator**

**Description:** Given a list of Document objects, this class will combine the text of the documents into one text and the respective Metadata objects into a list of Metadata objects. This class is no longer used in the system.

**Init params:**

* doc\_list

**Data members:**

* doc\_list (list of docs to combine into SuperDoc)
* output\_superdoc
* output\_superdoc\_filename

**Methods:**

* create\_superdoc() (creates a SuperDoc from the given Document objects)
* add\_doc(doc\_to\_add) (adds a Document to doc\_list)

**Test Cases:**

* create\_superdoc() with valid input doc\_list
* create\_superdoc() with empty input doc\_list
* create\_superdoc() with input doc\_list containing a non-Document object
* create\_superdoc() with input doc\_list containing a single doc
* output file is not writable

**DocumentSorter**

**Description:** Given a collection of document objects, a sort key (which is a field of the appropriate metadata object), and optionally a list of “allowed values” for the sort key, this sorter can perform two basic functions (1) to break a set of documents into subsets, sorting on a given metadata field (this yields subsets accounting for all of the documents in the initial set) and (2) to create a single subset from a set of documents by selecting only those documents with certain values for a given metadata field (this does not necessarily yield a subset containing all of the documents from the original set).

**Init params:**

* doc\_list

**Data members:**

* doc\_list (input list of documents to sort)

**Methods:**

* sort\_docs(sort\_field) (performs sorting – this is the most important method in the class)
* create\_subset(sort\_field, allowed\_values) (creates a subset of the current set of documents by only selecting documents whose sort\_field is equal to some value in allowed\_values).
* add\_doc(doc\_to\_add) (add a Document to the list of docs to sort)

**Test Cases:**

* sort\_docs with valid input doc\_list
* sort\_docs with empty input doc\_list
* sort\_docs with input doc\_list containing a non-Document object
* sort\_docs with sort field that that isn’t valid
* create\_subset with valid list of allowed\_values
* create\_subset with empty list of allowed\_values
* create\_subset with list of allowed\_values that don’t match any Documents
* add\_doc with valid Document object
* add\_doc with non-Document object

**DocumentConverter**

**Description:** Given a file containing one and only one document (along with fields/labels/metadata), this class will parse the file and create a Document object from the file.

**Init params:**

* file to parse
* pickle\_path (the path to output the converted Document)

**Data members:**

* input\_path
* output\_path
* converted\_doc (the Document object output by the converter)
* format file? (a file defining the format of the file to parse)

**Methods:**

* convert\_file() (the primary method of the class – converts a file to a Document)
* save\_converted\_doc()
* get\_author(file\_path)
* get\_titled\_item(line, item\_regex)
* split\_dates(date\_string)

**Test Cases:**

* ideal case – properly formatted file, properly created Document object
* file is improper format
* no metadata present in file
* no text in file (this may not actually be an error?)
* output file is not writable
* input file does not exist
* input file is empty

**AnalysisEngine**

**Description:** Given a list of subsets of Document objects (and perhaps analysis parameters TBD), this class will perform natural language text processing on the collection of documents and return a weighted list of the “most important” terms in each subset in the collection. This utilizes term frequency-inverse document frequency analysis as its basis for weighting terms (<http://en.wikipedia.org/wiki/Tf%E2%80%93idf>).

**Init params:**

* set\_of\_docs
* Analysis parameters TBD

**Data members:**

* subsets (list of subsets of documents)
* num\_docs (how many documents we’re analyzing)
* term\_list (list of all unique terms in full set of documents)
* other analysis parameters TBD?

**Methods:**

* set\_subsets(set\_of\_docs)
* count\_docs()
* convert\_docs(subsets)
* build\_full\_term\_list()
* get\_most\_freq\_terms(subsets, num\_terms)
* determine\_relevate\_terms(subsets, term\_list)
* analyze\_docs() (the primary method of the class – performs the analysis)
* collect\_term\_info(subset, relevant\_terms, num\_terms=5000)
* build\_weighted\_pairs(raw\_term\_info)
* calc\_doc\_frequency(term)
* calc\_tfidf\_for\_subset(term, subset)
* save\_term\_info(raw\_info, output\_path)
* build\_output(info\_set)
* destem(stemmed\_term, subsets)

**Test Cases:**

* analyze\_docs() with a single input subset
* analyze\_docs() with multiple input subsets
* analyze\_docs() with zero input subsets
* the previous three cases with regular Document objects?
* invalid analysis parameters
* analyze\_docs() with invalid type in input doc list (not Document or SuperDocument)

**WordCloudGenerator**

**Description:** This class and its functionality are relatively simple – given a list of (term, weight) tuples (and probably a parameter specifying the number of terms to illustrate), this will return a word cloud that visualizes the most important terms and how important they are relative to each other.

Much inspiration for this module was drawn from: <https://github.com/amueller/word_cloud> and <http://peekaboo-vision.blogspot.com/2012/11/a-wordcloud-in-python.html> – this class is essentially just a slightly modified version of his word cloud generator, so I am eternally grateful for Mr. Mueller’s work.

**Init params:**

* weighted\_terms
* output\_filename
* image\_height (default 400)
* image\_width (default 800)
* other visual parameters TBD

**Data members:**

* weighted\_terms (list (term,weight) pairs)
* output\_filename
* image\_height
* image\_width
* other visual parameters?

**Methods:**

* query\_integral\_image(integral\_image, size\_x, size\_y)
* generate\_word\_cloud(num\_terms\_to\_visualize=50, margin=5) (the primary method of the class – creates a word cloud from the weighted term list)

**Test Cases:**

* ideal case – generate\_word\_cloud() with valid weighted list
* empty input weighted list
* input weighted list with a single term
* invalid parameters (such as num\_terms\_to\_visualize > len(weighted\_terms))
* input is not in correct format

**WordCloudMain**

**Description:** This class represents the “main” program of the system – it kicks off everything else. Parts of this class are modified for the user’s specific analysis goal.

**Global variables:**

* OPINION\_PATH – the path where the raw opinion files live
* PICKLE\_PATH – the path to which the system should serialize the opinion Document objects
* ACCEPTED\_CITES – optional; a list of citations for use in sorting

**Init params:** none

**Data members:** none

**Methods:**

* main(should\_pack\_opinions=False) – the main method; if should\_pack\_opinions=True, this serializes all opinion files in OPINION\_PATH into pickled Document files in PICKLE\_PATH; otherwise, performs the entire analysis process; this method is NOT modified by the user
* pack\_opinions() – serializes opinion files; this method is NOT modified by the user
* unpack\_opinions() – unpacks pickled Document files from PICKLE\_PATH; this method is NOT modified by the user
* sort\_opinions(opinon\_list) – sorts the opinions into one or more subsets to analyze; one section of this method is modified by user to create the subsets s/he wishes to create
* run\_analysis(subsets, num\_terms) – fires up the AnalysisEngine to perform the statistical analysis and generate weighted list(s); this method is NOT modified by the user
* generate\_clouds(subset\_lists) – creates a wordcloud for each given subset; this method is modified by the user to create the word clouds s/he wishes to create

**Test Cases:**