**Design for “Word Clouds in Python” Project**

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>.

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) and passes each subset to the SuperDocGenerator.
3. Given a subset of Document objects, the SuperDocGenerator creates a SuperDocument object.
4. Given a set of SuperDocumentObjects, 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 the set of SuperDocuments, which it passes to the WordCloudGenerator.
5. 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:**

**Data members:**

**Methods:**

**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:**

**Data members:**

* doc\_text
* doc\_metadata (Metadata object)
* 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

**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

**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.

**Init params:**

**Data members:**

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

**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)
* print\_superdoc() (display the created SuperDoc)
* 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 and a sort key (which is a field of the appropriate metadata object), this will yield subsets of document objects, each subset having the same value for the sort field (subsets will be returned as a dictionary?).

NOTE: We will likely be using this sorter in two different ways (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:**

* list of documents to sort
* field\_to\_group
* allowed\_values (optional)

**Data members:**

* doc\_list (input list of documents to sort)
* sort\_field (field of Metadata on which to group)
* allowed\_values (list of values
* output\_subsets (dict containing key:value “subset name”:[list of documents] pairs)

**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).
* print\_subsets() (prints out the subsets generated by sorting)
* add\_doc(doc\_to\_add) (add a Document to the list of docs to sort)
* remove\_doc()???

**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
* format template?

**Data members:**

* input\_filename
* output\_filename
* output\_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)
* print\_doc(output\_doc) (displays the Document yields by the conversion)

**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 1+ SuperDoc 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 the collection. This will likely utilize term frequency-inverse document frequency analysis as its basis for weighting terms (<http://en.wikipedia.org/wiki/Tf%E2%80%93idf>).

**Init params:**

* 1+ SuperDoc objects
* Analysis parameters TBD

**Data members:**

* weighted\_terms (this is the output list of (term, weight) tuples that feeds into WordCloudGenerator)
* num\_docs (how many SuperDocs we’re analyzing)
* other analysis parameters TBD?

**Methods:**

* analyze\_docs() (the primary method of the class – performs the analysis)
* print\_docs() (displays the identifiers of the SuperDocs to be analyzed – this is a bad method name)

**Test Cases:**

* analyze\_docs() with a single input SuperDoc
* analyze\_docs() with multiple input SuperDocs
* analyze\_docs() with zero input SuperDocs
* 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 really quite 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.

**Init params:**

* weighted\_terms
* num\_terms\_to\_visualize
* other visual parameters TBD

**Data members:**

* weighted\_terms (list (term,weight) pairs)
* num\_terms\_to\_visualize [this is a long name]
* other visual parameters?

**Methods:**

* generate\_word\_cloud() (the primary method of the class – creates a word cloud from the weighted term list)
* print\_term\_list() (displays the list of terms with their weights)

**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