diviText: Visualizing Text Segmentation for Text Mining

BY

Amos Chapman Jones

A Study

Presented to the Faculty

of

Wheaton College

in Partial Fulfillment of the Requirements

for

Graduation with Departmental Honors

in Computer Science

Norton, Massachusetts

May 2011

Contents

1 Introduction **4**
1.1 Guide 5

2 Literature Review **6**
2.1 Data Mining 6
2.1.1 Text Mining 7
2.1.2 Techniques for Clustering and Classifying Texts 12
2.2 Common Text Mining Tools 15
2.2.1 Meandre 1.4.8 15
2.2.2 WordSmith 5.0 17
2.2.3 MALLET 18
2.2.4 NLTK 21
2.2.5 GATE 23
2.2.6 R 26

3 Methods and System Engineering **30**
3.1 Detailed Functional Specification 30
3.1.1 Goals 30
3.1.2 User Interface 31
3.1.3 Data: Spaces Between Words 37
3.2 Black Box: The Public API 38
3.2.1 CutterPanel Module 38
3.2.2 PHP 40
3.3 White Box: Internal Components 41
3.3.1 JavaScript and ExtJS 3.3.1 41
3.3.2 Hypertext Preprocessor (PHP) 43
3.3.3 Error Handling 47
3.4 Code Walk-Through 49
3.4.1 Page Load 49
3.4.2 Center: CutterPanel 54

3.4.3 Clicking a Word . . . 58
3.4.4 Automatic Cutters . . . 60
3.4.5 Text Upload . . . 61
3.4.6 Text Download . . . 64
3.4.7 Loading the Text Manager . . . 65
3.4.8 Clicking in Text Manager . . . 68
3.4.9 Save Chunkset . . . 69

4 Use Cases **74**
4.1 Toy Example . . . 74
4.1.1 Using Visual Cutter . . . 76
4.1.2 Using Simple Cutter . . . 79
4.1.3 Using Advanced Cutter . . . 79
4.2 Real-World Examples . . . 81
4.2.1 Beowulf in Fitts . . . 81
4.2.2 Other Examples . . . 82

5 Conclusions **83**
5.1 Future Work . . . 83
5.1.1 Functionality . . . 83
5.1.2 Usability . . . 84
5.1.3 Stability . . . 84

A Clustering Using Other Tools **86**
A.1 Meandre . . . 87
A.2 Python: NLTK and SciPy . . . 89
A.3 R . . . 89
A.3.1 R using RPy . . . 91

B Licensing and Source Code **94**
B.1 Licensing . . . 94
B.2 Source Code . . . 95

Bibliography **96**

Chapter 1

Introduction

Quantitative experiments in text mining often require the segmentation of texts into smaller units. Text segmentation is the process of dividing text into smaller pieces. It can be done by hand in a text editor, but this is time consuming and error prone especially when working with large corpora. diviText is a tool to help scholars segment text in an efficient and accurate way.

In research at Wheaton College on the Anglo-Saxon corpus, automated tools have performed the task of text segmentation. The original incarnation of this tool, written in the summer of 2008 was a Perl script that was highly specific to the Anglo-Saxon corpus. This script runs on the command-line where segmentation parameters are specified in the script's argument list. Texts to be segmented had to be placed in a specific folder. The arguments segmented the text based on a specified segment length. This served its purpose but the process was slow, the algorithm inefficient, it required a facility with command-line arguments, and targeted only Anglo-Saxon texts.

A second, and more general, version of the script was written for use over the web in the summer of 2010. This provided text boxes for user input and a file upload field. This worked well and was far more efficient than the original script. It introduced a different method to segment texts based on a set number of segments. But this prototype provided nothing more than a simple user interface.

The need for an efficient and easy to use tool for text segmentation led to the creation of diviText. diviText is the next incarnation of text segmentation tools. It provides a graphical segmentation technique where the user can visualize their segments before finalizing the final placement of breakpoints between segments, or chunks. The user can upload many texts and segment each differently, multiple times.

Because text segmentation occurs within a larger context of the computational analysis of texts, this thesis covers research regarding the area of text mining and some tools commonly employed when analyzing texts and corpora. Later sections

explain diviText, its functionality, and systems design.

1.1 Guide

This section serves as a guide to the rest of this thesis.

Chapter 2, Literature Review, provides a look at the topic of "Data Mining," specifically "Text Mining" (Section 2.1). Section 2.2 covers tools commonly employed when text mining with a focus on their ability to load texts and perform clustering and/or classification methods.[1]

Chapter 3, Methods and System Engineering, provides an in-depth look at diviText. Section 3.1 outlines diviText's functionality. Section 3.2 briefly explains a few key software methods in diviText. Section 3.3 looks at the different technologies used in diviText and provides some vocabulary used with each technology to make the following sections easier to understand. This section sets up Section 3.4 which walks the reader through much of the diviText code with associated code snippets.

Chapter 4, Use Cases, provides examples of what diviText can do through a few use-cases. Section 4.1 shows off diviText functionalities through a toy example, and Section 4.2 looks at a few real-life use-cases.

Chapter 5, Conclusions, reviews how diviText can be used and could be expanded upon in the future.

Appendix A, Clustering Using Other Tools, provides helpful examples of how to use some of the tools referenced in Section 2.2 to perform cluster analyses on a toy dataset.

Appendix B, Licensing and Source Code, provides a few details regarding licensing (Section B.1) and how to obtain the diviText source code (Section B.2).

[1]Unsupervised (cluster analysis) and supervised (classification) machine learning methods are beyond the scope of this thesis, but are imporatnt to cover here to set the context for diviText.

Chapter 2

Literature Review

Information and knowledge describe patterns seen in the world. Patterns are of interest in a multitude of academic fields and business opportunities. Sociologists look for patterns, and changes in patterns, to describe society. Physicists use mathematical models to describe patterns in the universe. Statisticians look for patterns, and anomalies in expected patterns, to describe trends. Google's business model depends on finding patterns in text, images, and e-mail to serve the best and most relevant advertisements to web searchers. Facebook collects and finds patterns in people's social graphs to sell to advertisers like Google. Amazon finds patterns in consumer buying habits to present shoppers with relevant products in the hopes that shoppers will buy more. Market analysts look for patterns in the stock market to try and maximize return on investment for investors.

Each day, more information is created and collected. This presents opportunities for scholarship and business. But looking for patterns in information is not trivial and more information presents additional difficulties. Computers are a great boon to apply in this area, especially computer software that implements algorithms to aid in pattern detection.

2.1 Data Mining

Data mining is the process of searching for patterns using computing tools. Computers require a specially coded type of information. Data are coded as strings of zeros and ones, or binary data. Binary data is the operational language of all digital computers organized into patterns interpretable in ways that make the computer work. Quantitative and qualitative information can be stored in computers as data in some pattern of zeros and ones. This data can now be "mined".

Data mining is quite like the process of mining for gold in the earth. First prospecting occurs to find where gold resides in the ground. Gold ore is found in hard rock

or loose gold found in rivers. Depending on where gold is found, specific instruments must be designed to extract the gold. Drills are used to extract gold ore. Pans, dredgers and sluice boxes are used to expose loose gold. Gold ore is then smelted into gold ingots which can be sold or locked away in places like Fort Knox. Loose gold can be sold directly to buyers.

Data is first prospected for its applicability to the problem. Is it the data equivalent of gold or calcium? Depending on the type of data, specific instruments and methods are developed to extract the useful data. Streaming video requires vastly different analysis than thousand year-old texts. In the case of business, the findings in the data can be sold or kept proprietary. In academia, findings will likely be published in journals.

No matter the field of interest or purpose when data mining, similar approaches are employed. Kononenko and Kukar (2007) show one "interactive and iterative" standard process, CRISP-DM (CRoss Industry Standard Process for Data Mining), that can be applied to the creation and life-cycle of a data mining project for business. The idea is that the problem of data mining is broken into smaller steps: problem understanding, data understanding, data preparation, data modeling, evaluation, and deployment. Problem and data understanding give researchers a place to start. How is the problem defined? What data is mostly likely to produce quality results? How does changing the data affect the problem? These feed into the steps of data preparation and modeling. Data must be prepared and stored. Models are created to organize, classify, and analyze the data. Models are evaluated for accuracy and quality of results. Models are then then deployed and fed back into the problem and data understanding processes. Good models and data can be refined iteratively by researchers. The details of each step vary depending on the problem and a variety of techniques and analysis algorithms are employed, some of which are described in Section 2.1.2.

2.1.1 Text Mining

Text mining is a subset of data mining focusing primarily on textual and natural language data. Text mining presents unique challenges due to how human language has developed. Human languages tend to be ambiguous and largely unstructured. Meaning is placed onto words and phrases by language structure, syntax, and vocal stress not obvious to computers and non-native speakers.

Jurafsky and Martin (2009) cite many different types of ambiguity with the one sentence, *I made her duck*. Word-sense ambiguity arises from words with multiple meanings or interpretations given a context. For instance, the word *made* could refer to the act of cooking the duck or the act of creating the duck, maybe out of plastic. Syntactic ambiguity is when two words could belong to the same sentence structure or be separate. *Her duck* could refer to a waterfowl belonging to a girl

(separate sentence entities) or refer to the act of making the girl lower herself in a rapid motion. Other ambiguities include the ability to differentiate between statement or question, determining the use of the word in a sentence, or determining the actual pronunciation of a word.

2.1.1.1 Corpora

Texts are a static form of data that appears in many ways, from hand written manuscripts of the Middle Ages and personal letters, to published books, magazine articles and Internet message board rants. Works are categorized by some method, be it logically by author, language or topic or subjectively by somebody's opinion of what is a good story.

A large collection of these categorized works is called a corpus. Some large corpora frequently cited include the University of Toronto's Dictionary of Old English (DOE) used by Anglo-Saxon scholars containing over 2 million words (Drout *et al.*, 2011; Healey *et al.*, 2004), the British National Corpus (BNC) that includes many texts, news articles and spoken word conversations with over 100 million words (BNC, 2007); and Mark Davies' corpora, the Corpus of Contemporary American English (COCA) and Corpus of Historical American English (COHA) with over 400 million words each (Davies, 2008b; 2010).

Many digitized corpora have online tools to search through texts, retrieve word frequencies, and more. For example, subscribing customers to the Dictionary of Old English can search for word and phrases as well as variants through the entire Old English corpus. The tools also provide full bibliographic information for all their texts (Healey *et al.*, 2009).

Using the same starting corpus, a collection of tools provided by the Wheaton College Lexomics Group allows users to download word counts for all poetry and prose texts, manuscripts and genres (poetry and prose) in the Anglo-Saxon corpus. Users can build their own collection of word counts for texts beyond traditional manuscript boundaries using a *Virtual Manuscript* tool. DOE subscribers can use a suite of Perl scripts to count words and reproduce the data found on the Lexomics website (LeBlanc *et al.*, 2010).

For the BNC, COCA, COHA, and other corpora, Davies provides an extensive database lookup tool to quickly search through the millions of words and phrases. The frequencies and context of every instance of searched words and phrases in the corpus is available very quickly using a modified relational database (Davies, 2008a; 2009).

Encoding Each of these corpora must be encoded in some format to store on a computer. The simplest encoding is raw text, usually using ASCII (American Standard

Code for Information Interchange) or Unicode, a text standard for storing many non-Roman characters. Texts stored as raw text, are just the words of the text without any special formatting or stylizing information or editorial notes. Project Gutenberg cites the advantages of this form, that, simple, raw text and ASCII in particular, can be read on nearly every computer built in the last thirty years, and will be readable by all computers for the foreseeable future (Hart, 2011).

The Anglo-Saxon Corpus is stored using Standard Generalized Markup Language (SGML) and includes metadata (Healey *et al.*, 2004). Using SGML, it is possible with proper tagging to keep track of line numbers in a poem or store the text's author without interfering with the author's words.

More advanced methods shown by Davies (2009) store all 7 word strings (7-grams) appearing in texts with their part-of-speech for extremely fast lookup of phrases using database queries. Information pertaining to individual texts are stored in other tables in the database.

Metadata An increasing number of corpora are digitized with associated information pertaining to the nature or source of the texts. This metadata is not necessarily part of the body of the text, but may contain useful information regarding publication date, author, genre, etc. Information about individual words may be stored such as the word's part-of-speech or definition.

There are many ways to store metadata, similar to storing texts themselves, ranging from the simplistic to well defined standards. Simply, metadata can be stored in an accompanying file, in the same file as the text, or in a more sophisticated and scalable approach, such as stored externally in a database that references the text.

Due to standards like the Text Encoding Initiative (TEI), texts and metadata are increasingly likely to be stored together in the same file (TEI Consortium). Using Extensible Markup Language (XML), texts are said to be "marked up" or "tagged" to include this data in the text itself using delimiters in a tree structure.

This structure has advantages because each word can be tagged and later isolated with its own part-of-speech tag, lemma, definition or more. Tagging with metadata also facilitates a separation of sections of a text such as the table of contents and index separated from the main body of text.

2.1.1.2 Word Counts

For computers, even the most neatly organized corpora represent text mining challenges; words, after all, are still ordered in human order. One way to structure the data into an algorithmically comprehensible form is to simply count the frequency of each word. In doing so, a word-list is created that stores word frequencies (Archer, 2009), as shown in Table 2.1.

Token	Count	Token	Count	Token	Count
⋮		caught	3	chain	1
case	5	cauldron	2	chains	1
cat	37	cause	3	chair	1
catch	4	caused	2	chance	4
catching	2	cautiously	3	chanced	1
caterpillar	28	ceiling	1	change	14
cats	13	centre	1	changed	8
cattle	1	certain	3	changes	2
caucus	3	certainly	14	⋮	
				total	27,333

Table 2.1: A sample of unique tokens and their raw counts in Lewis Carroll's *Alice's Adventures in Wonderland* (Carroll, 1865). There are a total of 27,333 types after removing punctuation and capitalization.

To count words, texts are tokenized. In this process, word tokens are created to represent each individual word string by some tokenization algorithm. Just as word tokens represent each individual word string in a text, word types represent each unique token in a text. Types can be thought of as the vocabulary of a text (Jurafsky and Martin, 2009).

Many methods of tokenization create tokens by splitting texts at each space character but this poses some counting issues. No differentiation is made between words that are logically the same to a human reader, but look different to the computer. For instance, when naïvely tokenizing, the token for *read* is different from the token for *Read* which is different when the token is immediately followed by punctuation such as *read:*. A common solution is to remove all punctuation from the string and to force case either up or down for each letter, thus collapsing each form of *read* into the same token creating a single word type. Another method is to separate the punctuation from the word string and thus creating tokens for punctuation, which for some types of analysis is useful.

Relative Frequencies One problem that occurs when raw word counts are used is that texts become segregated by length rather than content. Longer texts will employ the same words more frequently than similar short texts. To correct for this, relative word frequency is used. To calculate relative word frequency, the count of each word is divided by the total number of words in the text. The result for each word is the proportion of all words in the text that is that word. The sum of proportions for each type in a text is 1 and is the reason that relative frequencies normalize texts against

Token	Proportion	Token	Proportion	Token	Proportion
⋮		caught	0.00010975	chain	0.00003658
case	0.00018292	cauldron	0.00007317	chains	0.00003658
cat	0.00135367	cause	0.00010975	chair	0.00003658
catch	0.00014634	caused	0.00007317	chance	0.00014634
catching	0.00007317	cautiously	0.00010975	chanced	0.00003658
caterpillar	0.00102440	ceiling	0.00003658	change	0.00051220
cats	0.00047561	centre	0.00003658	changed	0.00029268
cattle	0.00003658	certain	0.00010975	changes	0.00007317
caucus	0.00010975	certainly	0.00051220	⋮	
				total	1

Table 2.2: The same sample of tokens with their relative frequencies in Lewis Carroll's *Alice in Wonderland*. Even the most frequent token shown, *cat*, represents only 0.135% of all word types in the text.

length. See Table 2.2 for the relative frequencies of the words from Table 2.1.

N-Grams When tokenizing a text, instead of isolating single words, it is possible to group consecutive words. One word token are called unigrams, two word tokens are called bigrams, three word tokens are called trigrams, and N word tokens are called N-grams (Jurafsky and Martin, 2009).

In December 2010, Google released an N-gram viewer using N-grams collected from over 500 years of scanned books in American and British English, Chinese, French, German, Hebrew, Spanish, and Russian. When searching for N-grams, users can compare multiple N-grams and view their popularity over a specified range of years. In addition to viewing and searching for N-grams, Google provides downloads for all sets of N-grams (up to 5-grams) (Michel *et al.*, 2010).

Lemmatization Another way to alter the word counting algorithm is to extract the lemma, or stem, of a word. A lemma is the base of a set of words that have essentially the same meaning and part-of-speech (Jurafsky and Martin, 2009). By lemmatizing input text, it is possible to remove attributes like tense, plurality, etc. from words and create fewer word types. For instance, to any native English speaker, the words *step*, *steps*, and *stepped* are basically the same, only different in tense. No simple algorithm can be used to lemmatize, though. The words *seek* and *sought* have the same basic meaning, but are radically different in spelling (Weiss *et al.*, 2005). Another example given, is the word *bored* which could be an adjective meaning tired, or the past tense of the word *bore* meaning to drill a hole. To date, there are no

perfect computational methods for lemmatizing words, especially in English. So a typical method used to lemmatize, is for human researchers to compile a list of words and their associated lemma(s) (*c.f.* Kleinman, 2011).

2.1.2 Techniques for Clustering and Classifying Texts

Machine learning algorithms are often grouped as *unsupervised* or *supervised*. Unsupervised learning places texts into groups after analysis while, supervised learning requires texts to be partitioned into groups prior to analysis.

While used in, but not limited to text mining, clustering and classifying techniques use word frequencies of texts in corpora as attributes to describe texts. Each text or segment of a text is represented by vectors of words and that word's relative frequency. These vectors of word frequencies can form the basis for clustering and classification methods. These methods have roots in statistical and probability modeling methods as do many other analysis techniques.

2.1.2.1 Hierarchical Clustering

One method of unsupervised learning is hierarchical clustering which has two types, agglomerative (bottom-up) and divisive (top-down). The result of a hierarchical cluster analysis is a set of groups one of which each text in the analysis belongs.

Agglomerative Clustering In agglomerative clustering, initially, each text (vector of word counts) forms its own group. Using a metric, the "distances" between texts are calculated. This metric could be Euclidean distance, Manhattan distance, angular separation or one of many other distance metrics. Using the computed distances between vectors of word counts, the closest two texts are grouped, or clustered together into a clade. The distance from this new cluster to all other texts is then recalculated. Using another metric, this cluster, or any text(s) inside of the cluster, new distances are calculated. Texts and clusters are then compared for similarity, and clustering continues until all texts belong to a single clade (Kononenko and Kukar, 2007; Kraus, 2010).

Divisive Clustering Divisive clustering is basically the opposite of agglomerative clustering. All texts are placed into a clade at the start and that clade is subdivided into smaller clades. This method is less popular because it is more computationally intensive than agglomerative clustering. But one advantage with divisive clustering is that after only a few subdivisions, trends and large groups should be recognizable (Kononenko and Kukar, 2007).

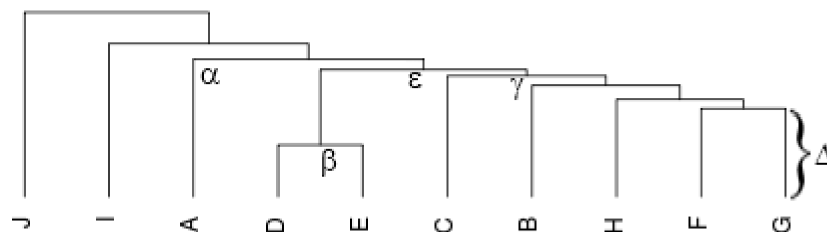


Figure 2.1: A labeled dendrogram with ten leaf nodes made using the R programming language. Four clades are labeled (α, β, γ, ϵ). Δ indicates the distance between texts F and G.

Dendrograms Hierarchical clustering results are often visualized using dendrograms because they are easy to read. Dendrograms provide a visual way to determine which texts are more alike.

More specifically, a dendrogram is a tree in which the leaf nodes represent individual texts and non-leaf nodes represent clusters, referred to as clades. The vertical distance between clades, in a vertically-oriented dendrogram, represents the relative mathematical distance between clusters.

The labeled dendrogram in Figure 2.1 shows the results of clustering ten texts. Labels A through J at each leaf represent a single text. The clade labeled β represents the linkage of texts D and E. γ shows a clade that links text C to the clade containing texts (B, H, F and G). ϵ is a clade that links two clades β and γ together. To analyze the results of the cluster analysis, it is important to look at the heights of the horizontal lines connecting texts and clades. The lower the horizontal line appears to the bottom of the dendrogram, the more similar a text or clade is to the text or clade to which the line links. For example, the link between texts D and E (β) is the lowest, thus those two texts are most similar. Text J is least similar from all other texts since its link to the rest of the dendrogram is the highest. Δ indicates the distance between texts F and G.

2.1.2.2 Classification

Supervised learning differs from clustering in that datasets are partitioned into groups prior to analysis. Texts are further separated into a training set and a test set. Each text in the training set is further categorized or labeled into subsets of known classes. The training set is used to 'train' the classifier, that is, teach the classifier how to identify instances of each class. Training sets are thus sets of texts whose class is already known, for instance sets of Shakespeare's or Dickens' works. The test set is the set of texts whose class is not known prior to analysis (or is known but ignored

during testing in order to verify the accuracy of the current model).

Bayesian Classifier One type of classifier is Bayesian classification. These classifiers use probabilities to classify a vector $v = w_1 w_2 \dots w_k$ of attributes (a list of k word counts) into a class C (Friedman and Kohavi, 2002; Weiss *et al.*, 2005). The objective is to estimate $P(C|v)$, that is, the probability of a text described by word count vector v falling into class C. Using Bayes' rule, this decomposes to $\alpha P(C)P(v|C)$, where α is a normalization constant not typically calculated, as it remains the same throughout the problem. The issue encountered is that no two texts have identical word distributions. Even when ignoring word counts and using a binary indicator to mark the presence of a word in a text, there are 2^k possible vectors; this number increases exorbitantly when using word counts.

To correct for this, the Naïve Bayes classifier is used. The classifier is said to be 'naïve' because it assumes that the appearance of each attribute, word, in v is independent. By assuming this, $P(v|C)$ decomposes into $P(w_1|C)P(w_2|C)\dots P(w_k|C)$ and the classification rule becomes

$$P(C|v) = \alpha P(C)P(w_1|C)P(w_2|C)\dots P(w_k|C).$$

The probability statistic is a product of probabilities that anything is of class C and the probability of a word occur given that we are looking at class C. And now, because any given w_i may not appear in a class, a very small number can be used in place. Friedman and Kohavi (2002) suggests $0.5/n$, where n is the number of instances of that word in the training set, otherwise known as Laplace smoothing.

The class decided by Bayesian classifiers is the class that yields the highest probability.

Maximum Entropy Maximum Entropy (MaxEnt) classifiers use statistical methods of multinomial logistic regression to correct for the unrealistic assumption of independence between attributes used by Bayesian classifiers (Weiss *et al.*, 2005). Word counts are used as predictors in a regression model to determine the probability of a text belonging to a class.

Decision Trees Decision trees use a classification algorithm that determine class based on simple sets of rules based upon training data used to traverse a tree. Like all trees, decision trees are made of nodes and leaves. Leaves represent the classes found in the training data. All other nodes consist of rules that determine the branching of the path the text being classified takes to a leaf node (Żytkow, 2002). Starting at the root node, a text being classified takes some path out of the node based upon the answer to a rule such as "has more than 5 occurrences of the word *child*" or "does

not contain *sneak* and has more than 1,000 unique words." The process continues until a leaf node is reached, at which point the text is classified. Many algorithms are based upon this idea such as the C4.5 Decision Tree algorithm and Classification and Regression Trees (CART). It is the processes employed by these algorithms to determine what rules are best and how many rules are enough to avoid problems of over fitting or useless branches (Kohavi and Quinlan, 2002).

2.2 Common Text Mining Tools

Many tools and packages have been developed for the purposes of data and text mining.

Detailed examples of many of the tools described in the following section, all using the same sample data set are, provided in Appendix A. In this section, the focus on each tool is directed at usability by novices, in particular for their ability to perform text mining using cluster analysis.

2.2.1 Meandre 1.4.8

Developed by the Software Environment for the Advancement of Scholarly Research (SEASR), Meandre is a data-intensive flow engine developed in Java to enable data processing via a platform-independent web user-interface. Meandre strives to be a fast and flexible environment for both users and developers (SEASR, 2010; Ács *et al.*, 2010).

Meandre is simple to set up as all it requires is a computer running Java and a modern web browser. Upon connection, the user is presented with the Meandre "Workbench," Figure 2.2, a clean interface that allows for the creation of a work-flow, or flow, as shown in Figure 2.3. Complex tasks are broken down into smaller tasks through the use of independent components that automate processing of data upon input.

Meandre comes packaged with many working sample flows ranging in complexity from simple tag-cloud builders and comma-separated value (CSV) viewers to the more complex C4.5 decision tree builders and clusterers. One intent is to allow users to change these and build their own flows. When building, adding onto or changing existing flows, the user selects pre-built components and adds them by dragging them from the list onto a working pallet. The user can then connect these components by dragging outputs from one component to the same typed input of another component. Many components have functional parameters, like `file_url` and `delimiter` that allow for more powerful control and flexibility of input data from the user's machine or the web in a variety of standardized and supported formats.

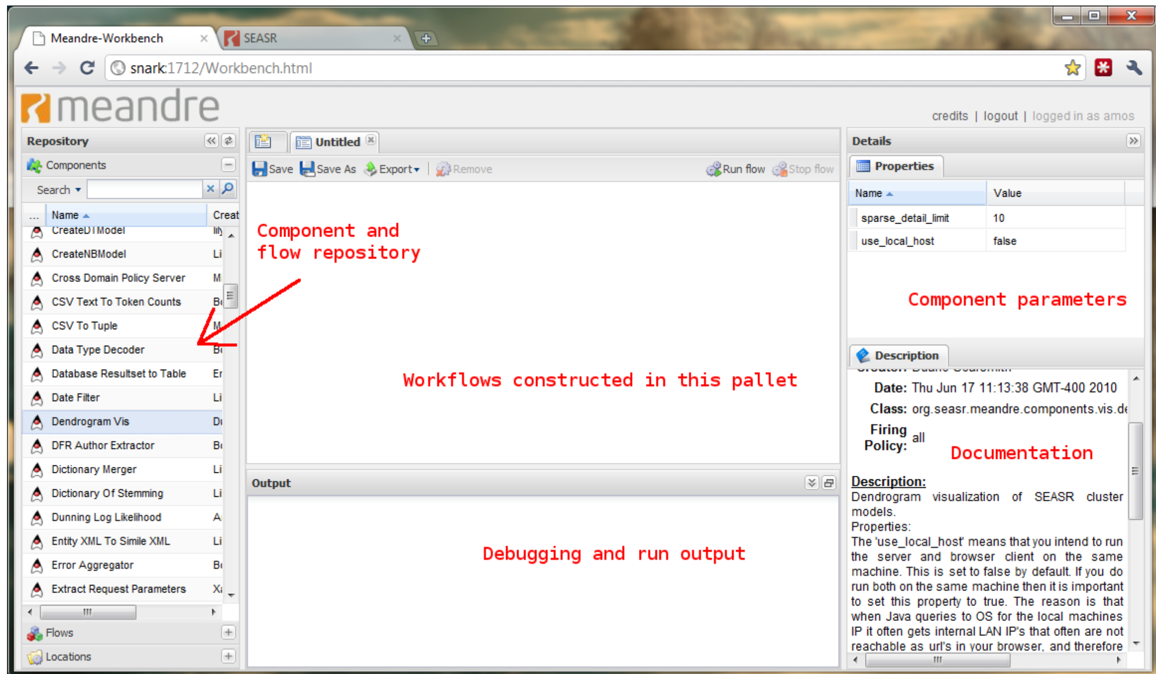


Figure 2.2: Initial Meandre Workbench. The repository of built-in flows, workflow pallet, and console are highlighted.

Developers can write and compile their own components in nearly any language. Components are designed to be "reusable, combinable and have predictable uniform behavioral attributes when they are executed" (Ács *et al.*, 2010). In other words, components are designed to automate one singular task on input data. By adhering to Meandre's functional framework, new components can make use of existing classes and structures and a new component can interact with existing components. This can be seen in the extensive list of components provided in the default working environment.

At the time of this writing, the current version of Meandre (v1.4.8) is a tool that, while powerful, needs more refinement. There are some issues with the required level of sophistication for both developers and users. Linguists and/or digital humanists with little to no experience in programming may find that making flows can be difficult as it is not readily obvious what components can and cannot connect. While parameters do allow some customization of input data, the ability to simply start with raw text files and run an analysis does not exist in a simple way. Another concern with Meandre, is the issue of usable output. For example, the dendrograms created by Meandre's clustering tools do not display information in an easily consumable format (see Figure A.3).

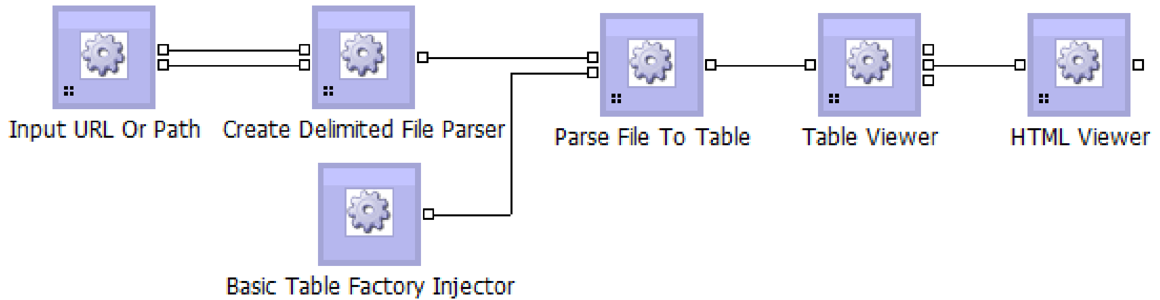


Figure 2.3: Simple Meandre flow that opens, parses and displays a CSV file in HTML.

2.2.2 WordSmith 5.0

WordSmith 5.0 is a Windows application created by Mike Scott at the University of Aston in the UK. This application is designed to facilitate the detection of patterns in large amounts of text (Scott, 2010). To do this, the program employs a number of subprograms: WordList, KeyWords and Concord.

WordList is a relatively simple program that counts words in texts. This is not the limit of WordList's power as it can determine frequencies of words both in individual texts and in the set combined. Other, purely word-based descriptive statistics are computed, including the mean and standard deviation of word length. WordList also takes sentences into consideration and finds the means and standard deviations of both sentences and paragraphs. For the user, WordList can generate output as simple Excel spreadsheets or a custom word list file that can be used in subsequent programs.

KeyWords is more complex program that calculates keywords in a text by comparing a text to a larger set of texts known as a Reference Corpus (RC). To select a keyword, the program defines the *keyness* of the word by identifying the relative frequency of a word in the text, comparing that to the frequency of the word in the RC, and performing a statistical calculation (Scott, 2010). If the word occurs above some user (or machine) defined minimum, the word is added to the list of keywords. If the word occurs more than expected in the text compared to the RC, its keyness is positive. Keyness is ordered and a list of keywords is produced.

The subprogram Concord produces concordances of texts, that is, it searches for patterns in texts given a phrase to search for. This phrase can be as simple as a single word or set of words or a more complex set of words including partial words, i.e. the phrase `the*` will match *the* and *there* among many other words while `the` will only match *the* (see Figures 2.4 and 2.5). This input phrase is similar to, but not nearly as powerful as regular expressions. The Concord subprogram then provides a list of every such occurrence and a few words on either side as context. Lists of most

N	Concordance	Set	Tag	Word #	Sent. #	Sent. Pos.	#	File	%
3,737	is some advantage in this; because these twin-tubs being so small they fit			106,672	5,146	11%	0	moby.txt	50%
3,738	sheep leap over a vacuum, because their leader originally leaped there when			117,512	5,617	71%	0	moby.txt	55%
3,739	bolster up a lame-duck visitor, because these days it's every girl for herself. Ive			39,003	2,944	71%	0	flap.txt	62%
3,740	zone, whose centre had now become the old man's head. Meantime, from the			203,090	9,780	88%	0	moby.txt	94%
3,741	Clarence Ahearn Company' becomes 'The Ahearn, Piper Company'--and 'Piper			32,177	2,481	65%	0	flap.txt	51%
3,742	his great padded surtout becomes the property of his executioner; how, in			160,501	7,464	44%	0	moby.txt	74%
3,743	the idea of "Piper Brothers" becoming "The Ahearn, Piper Company" startled			32,417	2,500	78%	0	flap.txt	52%
3,744	at least none but a supper and a bed. The landlord of the Spouter-Inn had			27,448	1,358	4%	0	moby.txt	13%
3,745	bed; Sal and me slept in that ere bed the night we were spliced. There's plenty			10,680	625	86%	0	moby.txt	5%
3,746	for a resurrection. Sixteen hours in bed! the small of my back ached to think of it.			13,540	759	18%	0	moby.txt	6%
3,747	set to planing away at my bed, the while grinning like an ape. The			9,556	577	84%	0	moby.txt	5%
3,748	but in the wife, the heart, the bed, the table, the saddle, the fireside, the			157,156	7,306	70%	0	moby.txt	73%
3,749	repugnance to his smoking in the bed the night before, yet see how elastic our			23,891	1,210	54%	0	moby.txt	11%
3,750	if, like Queequeg and me in the bed, the tip of your nose or the crown of your			23,611	1,202	29%	0	moby.txt	11%
3,751	day. And the women of New Bedford, they bloom like their own red roses. But			16,306	876	57%	0	moby.txt	8%
3,752	whaling stop at this same New Bedford, thence to embark on their voyage, it may			6,071	409	48%	0	moby.txt	3%
3,753	The Chapel. In this same New Bedford there stands a Whaleman's Chapel, and			16,386	881	20%	0	moby.txt	8%
3,754	tossing like slumberers in their beds; the ever-rolling waves but made so by			180,461	8,410	88%	0	moby.txt	84%
3,755	of day; when, looking over the bedside, there squatted Queequeg, as if he had			35,582	1,782	65%	0	moby.txt	16%
3,756	Horace Tarbox had been Mr. Beef the butcher or Mr. Hat the haberdasher,			21,581	1,553	50%	0	flap.txt	35%
3,757	unpeopled prairies had always been their choice in life, and I felt that they			22,096	1,189	46%	0	gold.txt	38%
3,758	of Starbuck, who had thus far been the foremost to assist him. His ivory leg			206,529	9,943	90%	0	moby.txt	96%

Figure 2.4: Sample concordance output for the pattern `the*`.

N	Concordance	Set	Tag	Word #	Sent. #	Sent. Pos.	File	%
125	she said "I'm awfully sorry about the Deyo dance. I'll give you my word of			44,611	3,435	82%	flap.txt	71%
126	Meadow, who sang a song about the Blundering Blimp in the first act and			21,713	1,558	61%	flap.txt	35%
127	old Italian publisher somewhere about the 15th century, during the Revival of			100,763	4,937	29%	moby.txt	47%
128	himself, and mutters something about the doors of convicts' cells being never			19,736	1,035	61%	moby.txt	9%
129	when he betrayed this solicitude about the safety of the ship, some of the			93,801	4,604	28%	moby.txt	44%
130	Captains of this earth. He skulks about the wharves of Joppa, and seeks a ship			18,906	983	31%	moby.txt	9%
131	the line, as it silently serpentines about the oarsmen before being brought into			107,570	5,164	74%	moby.txt	50%
132	the swimming crew are scattered about the whale in contrasting expressions of			102,485	5,005	60%	moby.txt	48%
133	were, as a dry old codger said about the Indian as a scalp-lifter, naturally fitted			727	52	87%	gold.txt	2%
134	down out of that! Mind what I said about the marchant service--don't aggravate			30,349	1,508	50%	moby.txt	14%
135	what the landlord said about the harpooneer's not coming home at all			11,308	651	33%	moby.txt	5%
136	maid, knew she bad been right about the bowl; it was still half full. She			33,011	2,544	85%	flap.txt	53%
137	Fairboalt's lingering reservations about the Freddy Gedney story quite vanished.			29,768	2,312	55%	flap.txt	48%
138	the paper, waited for a remark about the hold-up of the day before. But either			54,108	4,123	85%	flap.txt	86%
139	sneer, made a loud remark about the right of the lower animals to ride with			58,471	4,421	62%	flap.txt	93%
140	shouldn't have made that remark about the infantry being particularly dangerous.			34,961	2,718	69%	flap.txt	56%
141	which unheeded reel about the streets, you will see other sights still			15,735	842	66%	moby.txt	7%
142	It's the sort of island you read about." The rowboat was lowered into the golden			7,652	488	14%	flap.txt	13%
143	Medes. I peered and pryed about the Devil-dam; from her, hopped over to			29,024	1,432	18%	moby.txt	13%
144	rough and ready cook pottering about the fire, boiling buffalo-meat and mixing			3,317	163	71%	gold.txt	6%
145	she told me, while we pottered about the fire, how she waited her chance			52,627	2,971	52%	gold.txt	90%

Figure 2.5: Sample concordance output for the pattern `the`.

common phrases and word distances to the phrase aid in the search for patterns.

WordSmith is a robust program currently in its fifth incarnation, and very easy to install and use. Two drawbacks are that WordSmith is a Windows-only application (the recommended Mac solution is to buy a copy of Windows and dual-boot on an Intel Macintosh) and has an $80 (US) price point.

2.2.3 MALLET

MALLET, the MAchine Learning for LanguagE Toolkit, developed by Andrew McCallum and graduate students at UMASS Amherst, is a Java-based, platform-independent

toolkit for natural language processing through statistics. A few capabilities include text clustering, document classification, and topic modeling (McCallum, 2002).

MALLET operates only on a command-line interface invoking the `mallet` tool. This tool can perform a few main functions from the command-line, namely importing files and training classifiers.

To import files, the user isolates like texts, e.g. texts by the same author, into unique directories. The directory path is then part of the associated label to each text in the directory upon import. A special file is output containing each text, its word counts, and associated metadata. This output can be used by many of MALLET's other functions and tools.

Another of MALLET's functions is to perform classification of texts. To do this, the user must use an output file to train a classifier using one of a few trainers made available by MALLET including Naïve Bayes, Maximum Entropy and Decision Trees. The training methods provide straightforward ways to randomly separate training and test sets and to run repeated trials for cross validation. The `text2classify` tool can then use the trainer to classify a directory of text(s).

MALLET includes other tools including sequence taggers and topic modeling. One of the most powerful features for the Java programmer is the Application Programmer Interface (API). It provides a rich set of classes for creating new trainers and other tools.

The major drawback of MALLET for linguists and digital humanists is the command-line-only interface. This is not immediately intuitive and requires prior command-line experiences and may remain to be a significant hurdle when teaching MALLET to non-programmers.

2.2.3.1 Classifying Cynewulf Texts using MALLET

An example of classifying Cynewulf texts is provided here. Texts were split into three categories, *Definitely Cynewulf* (Cy), *Not Cynewulf* (NotCy) and *Possibly Cynewulf* (MaybeCy) on recommendation of Anglo-Saxon scholar Professor Michael Drout, (English, Wheaton College). The *Definitely Cynewulf* (Cy) set of texts contain the runic signature of Cynewulf. The preliminary organization of texts in directories is shown in Table 2.3.

A separate directory containing raw texts of the MaybeCy is read in and tested against each of the models. Table 2.4 reports the probability that Cynewulf was the author of each Maybe-group texts for each of the classifiers.

The Naïve Bayes classifier indicates that no text in MaybeCy is categorized by Cynewulf. The C4.5 Decision Tree indicates that both Andreas and Christ A are categorized as Cynewulf works and is undecided (*0.5*) on Guthlac B. Maximum Entropy only categorizes Christ A as a Cynewulf text. Christ A is classified by two of the

Set	Category	Text
Training Set	Definitely Cynewulf (Cy)	The Fates of the Apostles Juliana Elene Christ B
	Not Cynewulf (NotCy)	All other Anglo-Saxon poetry
Test Set	Possibly Cynewulf (MaybeCy)	Andreas Christ A Christ C Guthlac B

Table 2.3: Division of texts in experiment to determine likelihood of Cynewulf authoring four texts in the Test Set.

	Naïve Bayes	C4.5 Decision Tree	Maximum Entropy
Andreas	0.0	1.0	1.98×10^{-31}
Christ A	0.0	1.0	0.997454
Christ C	0.0	0.0	2.17×10^{-32}
Guthlac B	0.0	0.5	8.47×10^{-6}

Table 2.4: Results indicating the probability of each text in MaybeCy Test Set being authored by Cynewulf using MALLET's Naïve Bayes, C4.5 Decision Tree, and Maximum Entropy classifiers. Very small values in the Maximum Entropy column are so small they should be considered as zero in likelihood.

Listing 2.1: Shell script used to call the MALLET tools to train classifiers and classify Cynewulf texts.

```
1 # read contents of subdirectories separated into groups
2 bin/mallet import-dir --input cytest/* --output cynewulf.mallet
3
4 # train Naive Bayes, C4.5 Decision Tree, and Maximum Entropy trainers
5 bin/mallet train-classifier --input cynewulf.mallet --output-classifier
      cy_nb.classifier
6 bin/mallet train-classifier --input cynewulf.mallet --output-classifier
      cy_c45.classifier --trainer C45
7 bin/mallet train-classifier --input cynewulf.mallet --output-classifier
      cy_me.classifier --trainer MaxEnt
8
9 # classify new data using the three trained classifiers
10 bin/text2classify --input tests --output - --classifier cy_nb.classifier
11 bin/text2classify --input tests --output - --classifier cy_c45.classifier
12 bin/text2classify --input tests --output - --classifier cy_me.classifier
```

three classifiers as belonging to the Cy group which lends credence to the thought

that the first part of Christ (Christ A), as well as the known second part (Christ B), were written by Cynewulf, but probably not the last part (Christ C).

The set of command-line invocations used to perform this experiment is shown as a shell script in Listing 2.1. Each command was run individually so that the output could be parsed by hand for relevant output. Using `mallet`'s `import-dir` function, the Cy and NotCy groups' texts (located in the `cytest/cy` and `cytest/notcy` directories, respectively) are read in and grouped according to subdirectory and output into the `cynewulf.mallet` file. Using this file, three classifiers are made using the entirety of the texts as training data for each of the Naïve Bayes (default), C4.5 Decision Tree and Maximum Entropy training methods through MALLET's `train-classifier` function. Classification is done on a set of raw texts stored in the `tests` directory using each of the three different classifiers using the `text2classify` program.

2.2.4 NLTK

The Natural Language ToolKit, NLTK, is an open-source, platform-independent module for the Python programming language (Bird *et al.*, 2009). NLTK makes full use of Python's abilities including lambda functions (nameless functions that operate just as any other function), list comprehensions (defining lists using a mathematical set-building notation), and a comprehensive set of its own classes and functions to read, count, tag, and more, working on both raw and highly structured texts. In addition to a rich tool set, many full corpora are available to download through the `nltk.corpus` module, and the NLTK contains many languages, lists of special words and whole sets of tagged and untagged texts.

The functionality of NLTK is immense and well documented. Simple tasks often involve but a single function, for example splitting an entire corpus into list of sentences or lists of words. Python's lists make it simple to count words enabling NLTK to create frequency distributions. A few functions exist in the frequency distribution (`FreqDist`) class to easily retrieve counts of total and unique words, a range of words sorted by count and much more. An example is shown in Section 2.2.4.1. Once imported, a set of texts can quickly be turned into training sets for part-of-speech taggers or applied in a cluster analysis as shown in Section 2.2.4.2.

Since NLTK is released as a collection of Python scripts, developers can write new modules and edit existing modules. The code documentation is well written and available to the user through Python's extensive help library. Because NLTK is just a module in Python, input and output can be used in conjunction with other modules like Numpy and SciPy, numerical and scientific modules, respectively.

The biggest drawback of NLTK is Python itself. It is not that Python is a particularly difficult language to learn, in fact it is currently considered to be one of the most

suitable for teaching programming to novices (Leping *et al.*, 2009). However, as with any programming language, there is a steep learning curve for non-programmers. To accomplish even simple tasks with NLTK, one must be prepared to grasp non-trivial concepts of programming and data structures.

2.2.4.1 Counting Words in Inaugural Speeches

Listing 2.2 shows a simple use of NLTK's `FreqDist`, frequency distribution, class to count the words used in all 56 US Presidential Inaugural speeches.

Listing 2.2: Only five lines of code can produce a sorted list of of the top 20 words used in Presidential Inaugural Speeches.

```
1 import nltk
2 from nltk.corpus import inaugural
3
4 words = inaugural.words()
5 fd = nltk.FreqDist( [ w.lower() for w in words if w.isalpha() ] )
6 print fd.items()[:20]
```

Both `nltk` and the `nltk.corpus.inaugural` modules must be imported. The `words` function retrieves every word in the corpus as a list. A frequency distribution is instantiated with a list of all words using a list comprehension. A word is included if it is alphanumeric (a word or number) and then lowercased. In the instantiation, only words are counted. The `items` function of a `FreqDist` returns a list of words and their counts, ordered by counts. The top 20 are shown in Table 2.5.

Word	Count	Word	Count	Word	Count
the	9906	that	1726	which	1002
of	6986	we	1625	have	997
and	5139	be	1460	with	937
to	4432	is	1416	as	931
in	2749	it	1367	not	924
a	2193	for	1154	will	851
our	2058	by	1066		

Table 2.5: Top 20 words used by US Presidents in Inaugural speeches.

2.2.4.2 Clustering Words in Inaugural Speeches

Listing 2.3 takes advantage of standard Python functionality, NLTK and the SciPy science package to cluster all 56 Presidential Inaugural speeches, from Washington's

first inaugural speech in 1789 to Obama's 2009 inaugural speech, and produce the dendrogram in Figure 2.6.

Listing 2.3: Python script using NLTK, NumPy, and SciPy to hierarchically cluster and render a dendrogram of Presidential inaugural speeches. Resulting dendrogram is shown in Figure 2.6.

```
1  import nltk
2  from nltk.corpus import inaugural
3  from numpy import array
4  from scipy.cluster import hierarchy as hi
5  import pylab
6
7  # create list of counts of words for each speech
8  fd = [ nltk.FreqDist( [ w.lower() for w in inaugural.words(f) if w.isalpha() ] )for f
       in inaugural.fileids() ]
9  # make a list of every word in the speeches
10 words = [ w for i in range(0,len(fd)) for w in fd[i].keys() ]
11 # remove duplicates
12 uniques = set(words)
13 # make one array for each speech containing identically ordered word counts
14 table = [ array( [ fd[t][u] for u in uniques] ) for t in range(0,len(fd)) ]
15 # sum each array of counts, determines number of total words in speech
16 sums = [ sum(table[i]) for i in range(0,len(fd)) ]
17 # convert table to proprtions instead of raw frequencies
18 proptable = [ array( [ float(c)/sums[i] for c in table[i] ] ) for i in
       range(0,len(table)) ]
19
20 # cluster and build dendrogram
21 Z = hi.linkage( proptable, method='centroid' )
22 hi.dendrogram( Z, labels=inaugural.fileids() )
23 pylab.show()
```

Using the same `inaugural` data set, a list for frequency distributions, one for each speech, is created using the same list comprehension in line 8 that are used in line 5 of the previous example. A list is then created for each word appearing in the corpus, duplicates are kept (line 10). The next line (12) uses Python's `set` method to turn a list into a list without duplicates. A list of `arrays` is created (one list of words for each text) where each `array` contains the counts, in order of word appearing in the unique set of words, for each speech (line 14). The counts are then turned into proportions to normalize for speech length. Using SciPy methods, a hierarchical agglomerative cluster analysis is performed resulting in the dendrogram in Figure 2.6. More detail about clustering methods using SciPy is shown in Appendix A.2.

2.2.5 GATE

GATE, the General Architecture for Text Engineering, developed at the University of Sheffield (UK) is a Java-based, open-source, platform-independent front-end "capable

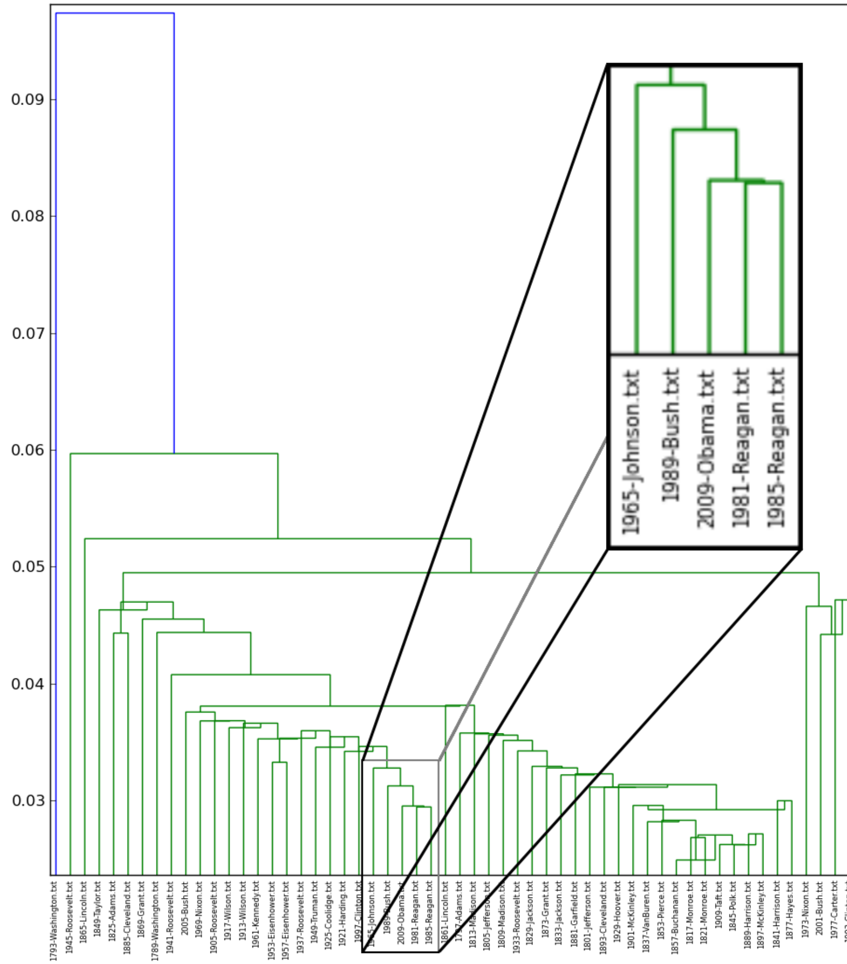


Figure 2.6: Dendrogram of a hierarchical cluster of all 56 Presidential inaugural speeches (years 1789 to 2009). The inset shows how closely both of Reagan's speeches clustered to Obama's speech as well as two other similar speeches, Bush's only inaugural in 1989 and Johnson's only.

of solving almost any text processing problem" (Cunningham *et al.*, 2010). GATE provides a User Interface (UI) for users (initial startup screen seen in Figure 2.7) and an API for developers.

CREOLE (a Collection of REusable Objects for Language Engineering) is the backbone and design philosophy of GATE. Each object and class is designed to be an independent entity with well-defined interfaces to interact with other objects.

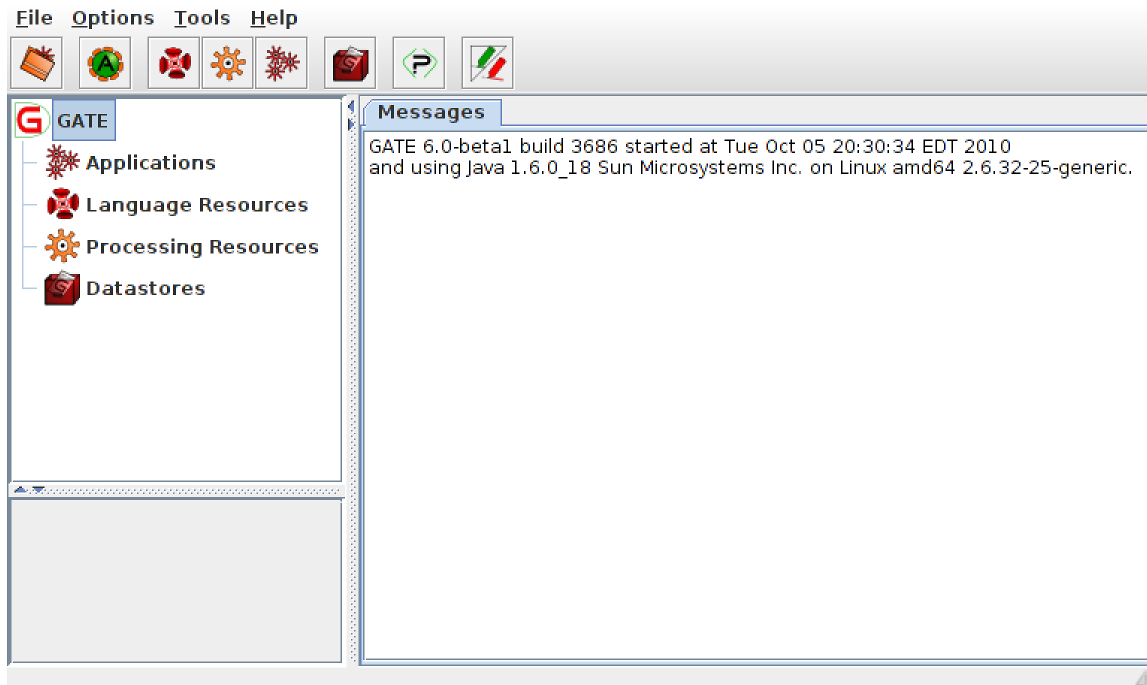


Figure 2.7: GATE user interface.

Three types of GATE components exist, LanguageResouces (LRs), ProcessingResources (PRs), and VisualResources (VRs). LRs exist as input texts and corpora. PRs exist to process texts, e.g. parsing. VRs exist to interact with the user in a GUI.

ANNIE (A Nearly-New Information Extraction) System serves primarily as a pipeline of PR tools. Some functions are designed to tokenize, split, and tag, input texts. These texts can take on many forms including, but not limited to HTML, XML, SGML and raw text (Cunningham *et al.*, 2010).

GATE is itself mostly an infrastructure designed to facilitate new functionality. But, because of the modular nature of the GATE design philosophy, much of the functionality comes through user-developed plugins as well as core plugins designed and shipped with GATE. Plugins are written in Java and use the functionality provided by the GATE infrastructure. Many plugins exist to read different languages, tag, parse, and, even search and translate web pages through Google and Yahoo.

For users, GATE provides a simple UI with easy access to imported texts and corpora and processing tools. With a text selected, the user is presented with a text editor as well as various markup tools that can highlight HTML or XML tagged regions or sentences and paragraphs in raw text. A selected corpora presents the user with an editor to add and remove texts. Applications are pipelines of various processing resources.

Despite an active developer-base, it may be difficult to find plugins that accomplish specific tasks. A search of the GATE website (November 2010) yields no information on cluster analysis and little on classification. This means, if a user wants to cluster, they would need to write their own clustering plugin.

2.2.6 R

R is an open-source programming language geared towards statistical computing and graphics across a wide variety of operating systems (R Development Core Team, 2010). The R community provides support and builds a wide array of packages to complement the many existing features of R.

R operates like any scripting language through an interpreter on the command line or through pre-built scripts. A variety of mathematically useful data structures are provided from scalars to vectors and matrices. Many functions and operators are designed to interact with each data type.

One of the most powerful features of R is its ability to produce publication-quality graphical reports. The Comprehensive R Archive Network, or CRAN, maintains current and archive versions of R as well as a repository of user-built packages.

Like other scripting languages, the biggest drawback to R is that it is a programming language. R is the language of choice for statisticians who want to perform statistical analyses and get quality graphical output. We view R as having a very difficult learning curve for linguists and digital humanists with no programming experience.

2.2.6.1 Cynewulf Clustering in R

This example intends to show R's powerful clustering features using the Cynewulf data from the MALLET example in Section 2.2.3.1.

First, text files from the previous Cynewulf experiment are read in, words counted, relative frequencies calculated, and stored in a comma separated value (CSV) file. The last twelve lines of the CSV file are shown in Table 2.6. Each text is stored in one row, with the first column indicating the text. Subsequent columns indicate the relative frequency of a word in the text. From this, riddles were removed from the list of texts leaving a total or 109 texts in the cluster.

The code to cluster in R from this table is straightforward and shown in Listing 2.4. In line 1, the `cynewulf.csv` file is read in. The `row.names=1` parameter indicates that the first column is where the names of the texts are stored. The following two lines (3 and 4) calculates the distance matrix and clusters. The last three lines (6-8) print the output of the plot function (the dendrogram) to the file

not/A03_020_Deor_T00300.txt	0	0	0	0	0	0	0	0	...
not/A03_018_Part_T00280.txt	0	0	0	0	0	0	0	0	...
not/A08_WaldA_T01420.txt	0	0	0	0	0	0	0	0.01	...
not/A07_Finn_T01410.txt	0	0	0	0	0	0	0	0	...
maybe/A02_001_And_T00050.txt	0	0	0	0	0	0	0	0	...
maybe/GuthB.txt	0	0	0	0	0	0	0	0	...
maybe/ChristC.txt	0	0	0	0	0	0	0	0	...
maybe/ChristA.txt	0	0	0	0	0	0	0	0	...
cy/A02_002_Fates_T00060.txt	0	0	0	0	0	0	0	0	...
cy/A03_005_Jul_T00150.txt	0	0	0	0	0	0	0	0	...
cy/A02_006_El_T00100.txt	0	0	0	0	0	0	0	0	...
cy/ChristB.txt	0	0	0	0	0	0	0	0	...

Table 2.6: Vectors of relative word frequencies for the last 12 (of 109) texts in the CSV file used to generate the cluster analysis. Each text was prepended with a word to identify the group of the text to the dendrogram reader. Numerical columns represent unique words and their relative frequencies.

Listing 2.4: R script to perform hierarchical cluster on the Cynewulf CSV dataset.

```
1 vectors <- read.csv( "cynewulf.csv", header=T, comment.char="", row.names=1 )
2
3 vdist <- dist( vectors )
4 result <- hclust( vdist )
5
6 png( 'cynewulf_cluster_r.png' )
7 plot( result, hang=-1 )
8 dev.off()
```

cynewulf_cluster_r.png. In the plot function, the parameter hang=-1, indicates that the leaves of the dendrogram should hang down to a fixed line.

The resulting dendrogram has 109 leaves and is thus very crowded with texts, most of which are of minimal concern to the experiment. However, all of the texts in the MaybeCy and Cy groups are clustered very closely. This makes it possible to snip out just the relevant portion of the dendrogram, as shown in Figure 2.8.

2.2.6.2 RPy: R in Python

One effort to minimize the necessity of the complexities of R without reducing the abilities of R is to use Python, a novice-friendly language. The RPy package for Python does just this. Much of the necessary data input and manipulation can occur within Python while statistical reports and graphs can be generated in R using the RPy interface.

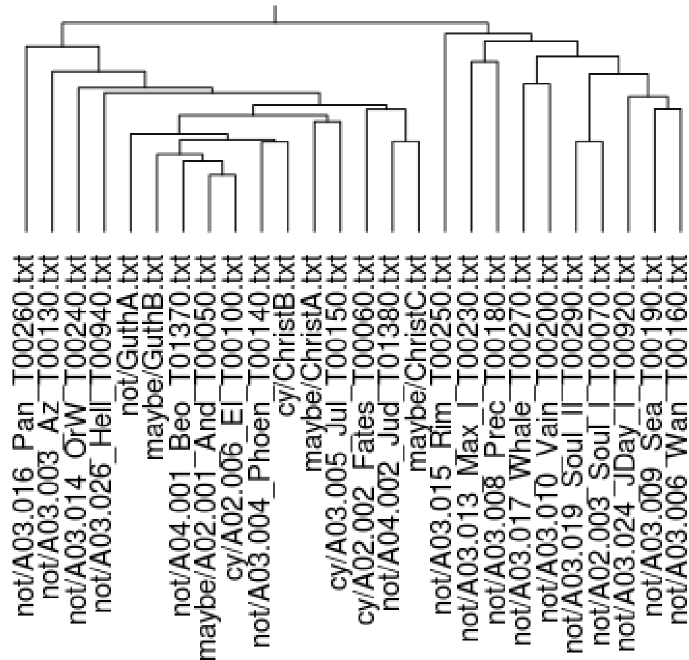


Figure 2.8: 26 of the 109 texts in the cluster analysis consisting of almost all of the first texts to be grouped. The first two texts grouped were Elene and Andreas, a Cynewulf text and Maybe Cynewulf text, respectively.

RPy instantiates an R environment inside of the running Python environment. Variables can be passed from Python to R using the `r.assign()` method. R to Python data transfer occurs when an R method returns data from a method which is then returned through the Python method call into the Python environment. Such data can be any standard R datatype crafted in certain ways in Python so RPy can interpret the structure into something R can handle. R functions and data can be accessed directly from Python using the `r` class.

Cynewulf Clustering in R through Python using RPy Using RPy, it is possible to reproduce the results from the R cluster of Cyenwulf texts in Section 2.2.6.1. In this case, Python is used to read in the data and passes the data off to the R environment to let R do the clustering.

Listing 2.5 produces the same dendrogram as Listing 2.4 with, however, a little more work. Lines 1 through 3 import the needed modules, `rpy` for R interaction, `csv` to read in the data file, and `numpy.array` to coerce Python's data types into data R can handle. Line 5 opens and reads the file `cynewulf.csv` into a list of rows, where each row is a list of values as delimited by commas. The `for` loop on

Listing 2.5: Python script to perform hierarchical cluster on the Cynewulf CSV dataset. This script is equivalent to the R script in Listing 2.4. To make variables clear, a `p` is prepended to Python variables and `r` is prepended to R variables.

```
1 from rpy import r
2 import csv
3 from numpy import array
4
5 preader = csv.reader( open( 'cynewulf.csv', 'r' ), delimiter=',' )
6
7 ptexts = []
8 pvectors = []
9
10 for row in preader:
11     ptexts.append( row[0] )
12     pvectors.append( [ float(n) for n in row[1:] ] )
13
14 r.assign( 'rtexts', ptexts )
15 r.assign( 'rvectors', array( pvectors ) )
16
17 r( 'rvdist <- dist( rvectors )' )
18 r( 'rresult <- hclust( rvdist )' )
19
20 r.png( 'cynewulf_cluster_py.png' )
21 r( 'plot( rresult, labels=rtexts, hang=-1 )' )
22 r( 'dev.off()' )
```

Line 10 iterates through rows read in from the file. The list `ptexts`, instantiated on Line 7, gets the text's name of the row, stored in the initial cell. `pvectors` appends the list of all values in the row that are not the first value. Because `csv` reads the file in a string, the values must be coerced into `floats`.

The `r.assign` function places the Python variable, argument 2, into a variable in R specified as argument 1. When adding `pvectors` to R, it must be wrapped in a `numpy.array`, as that is how R accepts matrices.

Lines 17 and 18 use R commands placed inside single quotes to calculate the distance matrix and cluster based on the matrix. Lines 20 through 22, create the PNG output file, `cynewulf_cluster_py.png`, and plots the dendrogram into that file. One change from the R script is that the labels must be specified since they were not included in the `rvectors` matrix.

The resulting dendrogram is also large and cluttered, but otherwise, the same as before.

Chapter 3

Methods and System Engineering

This chapter explains how diviText works at both high and low level system views. Section 3.1 provides a look at the system's functionality by presenting how the user interacts with the application. Section 3.2 gives an overview of some functions and scripts and the results of their invocation. Section 3.3 details how parts of the JavaScript and PHP systems work in an effort to make later sections more understandable. Section 3.4 discusses how and when functions and scripts are invoked.

3.1 Detailed Functional Specification

diviText is a web-based application designed to facilitate the process of text segmentation for quantitative analysis in text mining. To do this, diviText presents a clean user interface (UI) that allows the user to upload texts to a server, graphically or automatically segment their texts while simultaneously viewing a current snapshot of their choices, and download word frequencies for their segmented texts.

diviText is rendered in a modern web browser as an HTML page built with the ExtJS 3.3.1 JavaScript framework. This framework facilitates data-driven page rendering and communication with the server. Communication between the browser and server uses Ext's Asynchronous JavaScript and XML (AJAX) module. This creates a request for a page which is returned from the server and parsed by the JavaScript. A custom "CutterPanel" module is implemented to perform text segmentation, both visually and automatically.

3.1.1 Goals

Scholars who "mine" texts often need to segment these texts in order to provide a fine-grained analysis that cannot be achieved by simply working with whole texts within corpora. diviText is designed to provide an easy way to graphically view

text segmentation and return data that can be used in subsequent quantitative text analysis.

3.1.2 User Interface

Upon loading diviText into a web browser for the first time in a session, the user is presented with a fresh interface with some text instructions on how to use the application. This text is placed in the main, Visual Cutter Panel, so users can practice using the tool prior to uploading their own texts. Figure 3.1 shows the main diviText interface.

Many sections in the UI can be resized by the user.

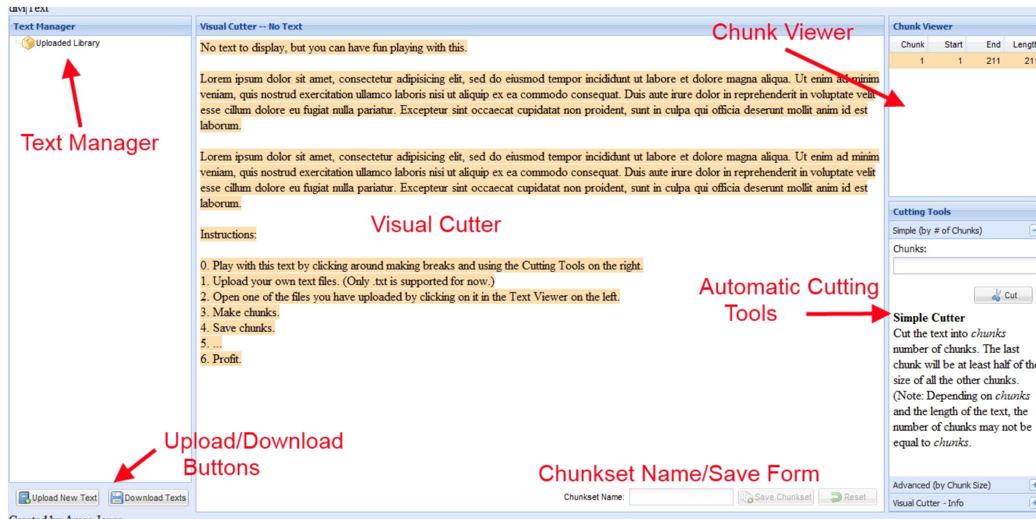


Figure 3.1: Initial interface presented to user upon the start of a session. Annotations were added to indicate some common vocabulary and label some areas of the screen.

3.1.2.1 Text Manager and File Uploader

The Text Manager is the first thing that the user must access (upper left in Figure 3.1). This tree object manages uploaded texts and chunksets built with the Cutting tools. The "Upload New Text" button (lower left) displays a dialog box in the browser with a form that allows the user to choose a text file from their hard drive, name it, and upload it to the server (see Figure 3.2).

The first field in the form is the "File" field. Clicking the "Browse..." button displays a window produced by the user's operating system. Using this, the user can choose a file from their computer.

Figure 3.2: Dialog box opened when "Upload New Text" button is clicked.

After selecting the desired file, the "Text Name" field is automatically populated with the file name parsed from the file path in the previous field. This can be edited by the user to whatever they wish to name the text.

Clicking the "Upload" button, tells the browser to send the file and associated name to the server. If the file is uploaded successfully, the dialog box is removed and the Text Manager is reloaded, adding the new text to the tree of uploaded texts (see Figure 3.3).

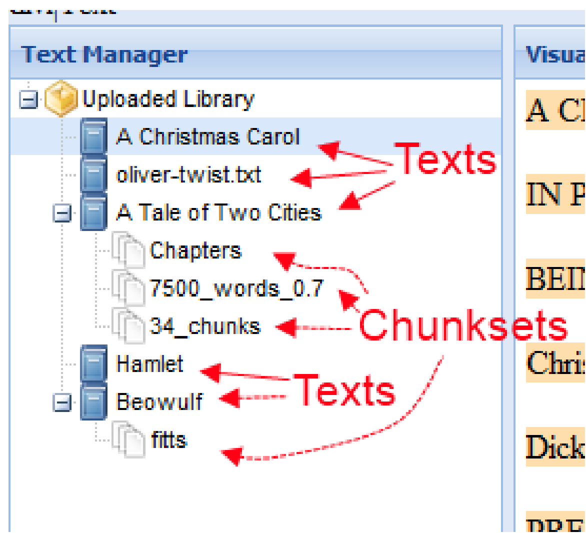


Figure 3.3: Text Manager after five texts have been uploaded and four different chunksets have been created, one for "Beowulf" and three for "A Tale of Two Cities."

The Text Manager's tree represents the hierarchy of the user's files that have been uploaded. Hanging off of the "Uploaded Library" root node are all of the user's texts uploaded to the server. If the user has created chunksets for a text, a leaf node hangs off of the text node (see Figure 3.3).

Using the contextual menu, right-clicking or double-clicking a node in the tree,

the user can operate on texts and chunksets. A right click on a node representing a text displays a menu with two options, as shown in Figure 3.4. The user has the option of showing the text (also accomplished by clicking the node), or removing the text and all associated chunksets (see Section 3.1.2.2 for creating chunksets).

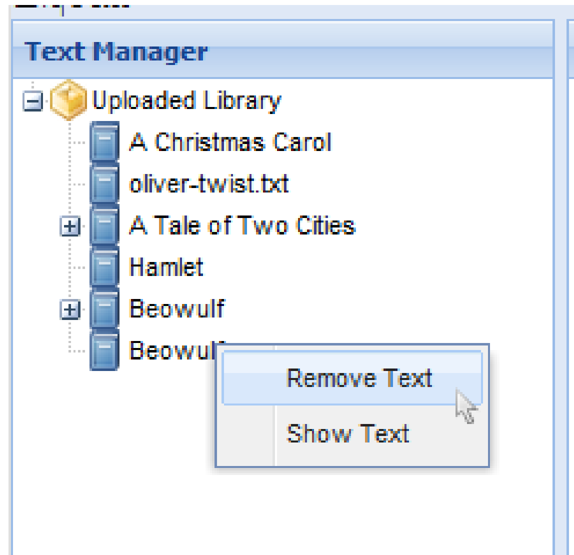


Figure 3.4: Context menu shown when right-clicking or double-clicking a text node in the tree. In this example, the user has accidentally uploaded the same text twice and wishes to remove one copy of the text.

Similarly, when the user right-clicks a chunkset, a menu is displayed with two similar options: "Remove Chunkset" deletes the chunkset and "Show Chunkset" loads the associated text into the Visual Cutter window and displays the text and associated chunk breaks in this chunkset.

3.1.2.2 Visual Cutter Panel

The Visual Cutter panel, located in the center of the page is the focus of the application (see Figure 3.1).

Upon selecting a text from the Text Manager, the body of the text is rendered into the Visual Cutter region. If the user has previously segmented the text, it is colored so the user can distinguish how the text was previously segmented.

To change how the text is segmented (that is, to add a new chunk break), the user simply clicks the word that starts a new chunk. The words on both sides of the new chunk-break are then recolored to make the change obvious. When hovering over a word with the mouse, but prior to clicking a word, the word is highlighted and a black bar placed on the top and left sides of the word indicating that this word will

be the start of a new chunk if selected. Hovering over a word also reveals the number of the word in the text. Words are indexed starting at 1.

The images in Figure 3.5 show some of the progression (top left to bottom right) of a user segmenting the initial sample text into segments on sentence boundaries. This is accomplished by clicking the first work of each sentence, except for the first sentence (because the very first word in the text implicitly starts a new segment).

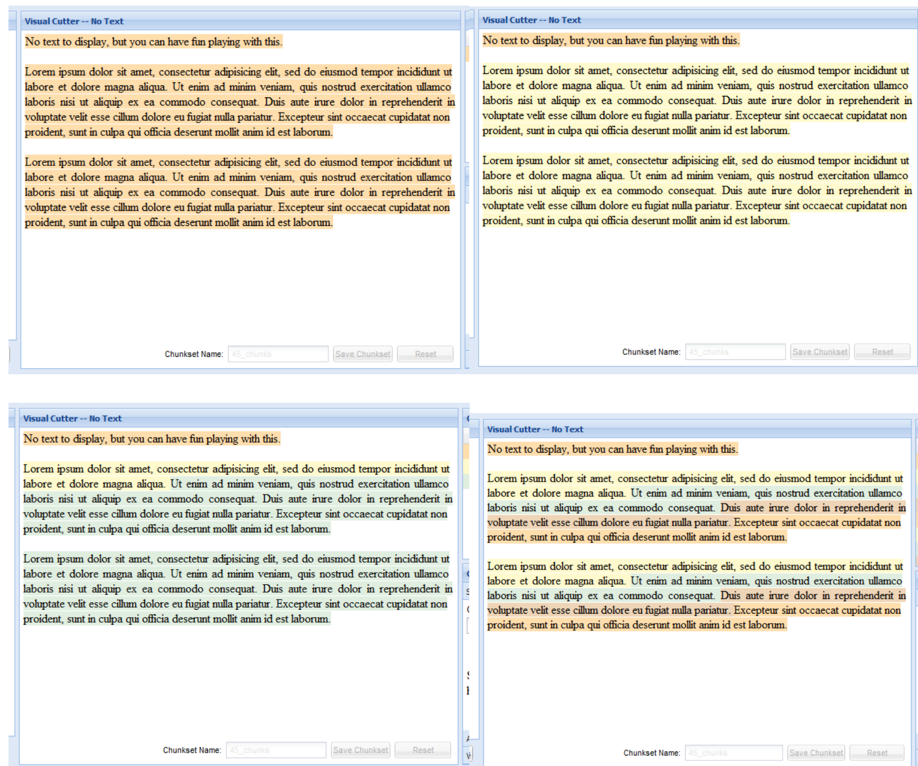


Figure 3.5: The progression of clicks (from the top left to the bottom right) to segment the sample text into segments of sentences. The bottom right image indicates the final segmentation on sentence boundaries.

Located at the bottom of the middle Visual Cutter Panel (see Figure 3.1) are “Save Chunkset” and “Reset” buttons and a text field where the user can specify the name of a chunkset. This input bar is shown in Figure 3.6. The “Save Chunkset” button sends a message to the server with instructions on how to segment the text and to save this chunkset with the name in the field. The “Reset” button removes all segments in the text. This bar is disabled when the sample text is active or when switching to a different text.

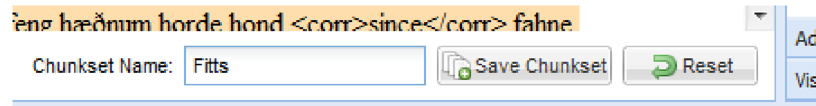


Figure 3.6: The bar at the bottom of the Visual Cutter Panel. Here, the "Chunkset Name" field is populated indicating that the user has named the chunkset "Fitts" prior to saving.

3.1.2.3 Chunk Viewer

As the user sets chunk breaks in the Visual Cutter Panel, a table in the top right of the page indicates how the text is currently segmented. The table displays the chunks in order, from top to bottom of the text, indicating the word numbers that define the first and last words in the chunk as well as the number of words in each chunk. Each row in the table is colored with the same color of the chunk of text in the Visual Cutter Panel. A sample is shown in Figure 3.7 after a few chunks have been created.

Clicking on a row in this table jumps the Visual Cutter panel to the first word in that chunk (indicated by the "start" element of the row). The first and last words in that chunk are then briefly highlighted in red in the text.

Chunk Viewer

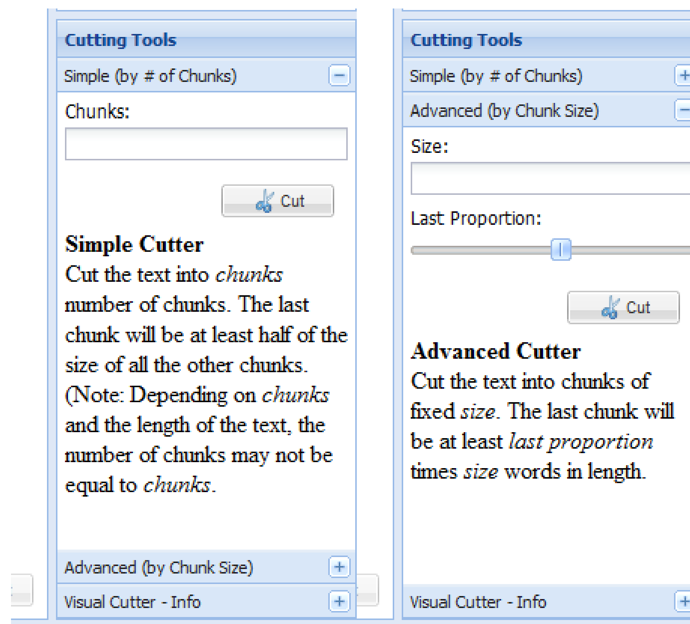
Chunk	Start	End	Length
1	1	12	12
2	13	31	19
3	32	48	17
4	49	64	16
5	65	81	17
6	82	100	19
7	101	117	17
8	118	133	16
9	134	150	17

Figure 3.7: The Chunk Viewer showing a table of results for a text that has been segmented into nine pieces. The chunks correspond to the chunks from the final image in Figure 3.5.

3.1.2.4 Automatic Cutting Tools

In addition to manually setting chunk breaks as previously discussed, the interface also provides options to automatically segment texts based on input parameters. Automated cutting options are located in the lower right of the diviText interface (see Figure 3.1). An accordion menu presents each option: "Simple Cutter," "Advanced Cutter," and an info-box for the Visual Cutter.

The Simple and Advanced Cutting options have a "Cut" button that sends the specified parameters to the Visual Cutter panel to visualize the cut. This step does not send any data to the server, but only shows the result of the cut. The user can then modify these automated chunk breaks through visual means. It should be noted that the Simple and Advanced Cutting options apply to the entire text and erase any cuts previously made.



(a) Simple Cutter. (b) Advanced Cutter.

Figure 3.8: Automatic cutting tools in the lower right of the interface. Simple Cutter (a) cuts the text into the specified number of chunks. Advanced Cutter (b) cuts the text into user-specified sized chunks.

The Simple Cutter option, shown in Figure 3.8a, allows the user to specify the number of chunks in which to cut the text. Chunk breaks will be placed every $round(\frac{N}{C})$ words where N is the number of words in the text and C is the specified number of chunks, that is, the size of the chunks will be the best integer value to fit the text to the desired number of chunks. This number is specified in the "Chunks" field as an integer greater than 0.

The Advanced Cutter option, shown in Figure 3.8b, uses a "Size" parameter to cut the text into fixed sized chunks. This field accepts only positive integers. A slider with the label "Last Proportion" affects only the final chunk. If the final chunk ends up being less than the specified proportion of all the other chunks, it will be merged with the final chunk creating a chunk larger than the specified chunk size. This slider

defaults to a value of .50 and is limited to .01 intervals between .01 and 1. When sliding, the current value is shown in a tool tip. For example, if a text consists of 100 words, and the user entered a chunk size of 7, and a last proportion of .5, the text would be split into 13 chunks of size 7, and a 14th chunk of size 9. If the last proportion were to change to something less than .28, the text would be split into 14 chunks of size 7 and a 15th chunk of size 2.

3.1.3 Data: Spaces Between Words

The underlying data structure for storing chunk breakpoints is a simple array. When a user clicks a word or its following space, an array is updated either adding/removing a number to/from the array. This number is the index of the space between the clicked word and the previous word, an index to a space character.

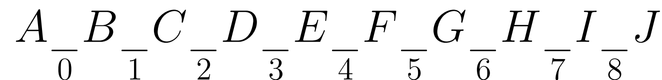


Figure 3.9: A visual explanation of space breaks used to define chunksets. Here words are represented as capital letters A through J and spaces are underscores indexed 0 through 8.

Figure 3.9 aids in explaining how spaces are defined. Words are represented by letters A through J and spaces are represented by underscores indexed by numbers 0 through 8. There are a total of 10 words and 9 spaces. A new segment is marked by the space to the left of the word that defines a new chunk (that is, the space after the last word in a chunk or the space before the first word in a chunk).

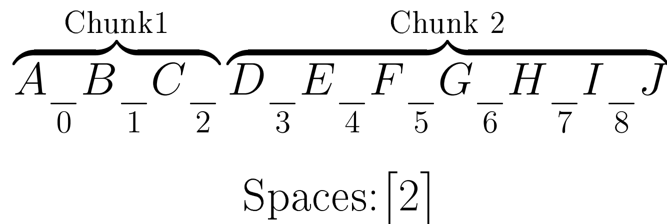


Figure 3.10: The resulting visual and space array data if the user clicked D or space 3.

For instance, if two chunks are defined as ABC and DEFGHIJ, then the array of spaces would only contain a 2 (shown in Figure 3.10). A break cannot be placed

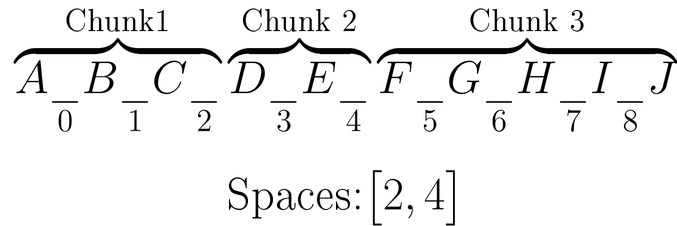


Figure 3.11: Assuming the two chunks shown in Figure 3.10, the resulting visual and space array data if the user then clicked F or space 5.

on the first word (i.e. the space that would be represented by space −1) because the start of the text is implicitly the start of the initial chunk.

If the user were then to define three chunks as ABC, DE, and FGHIJ (by clicking on word “F” or space 5), a 4 would be added to the array (shown in Figure 3.11).

This array of space data is used in the backend to produce chunksets. Figure 3.12 summarizes the major functional componented of the diviText tool.

3.2 Black Box: The Public API

This section outlines the Application Programmer's Interface (API) to a few PHP scripts and the JavaScript `CutterPanel` module.

3.2.1 CutterPanel Module

Built in JavaScript using the ExtJS framework, a custom `CutterPanel` component was developed as the focus of diviText. This is simply an extension of the existing `Ext.Panel` module provided by ExtJS. This panel handles both the rendering of texts to the page and the cutting of texts.

A few functions have been developed for the `CutterPanel` component to perform text segmentation and change the current text. Here, the functions are explained only at a high level. See Section 3.3 for the technical implementation details.

3.2.1.1 newText

The first of the `CutterPanel` functions is `CutterPanel.newText(text, id)`. A call to this function updates the Visual Cutter Panel with a string of text, specified with the `text` parameter. The text is parsed and replaces the text currently in the

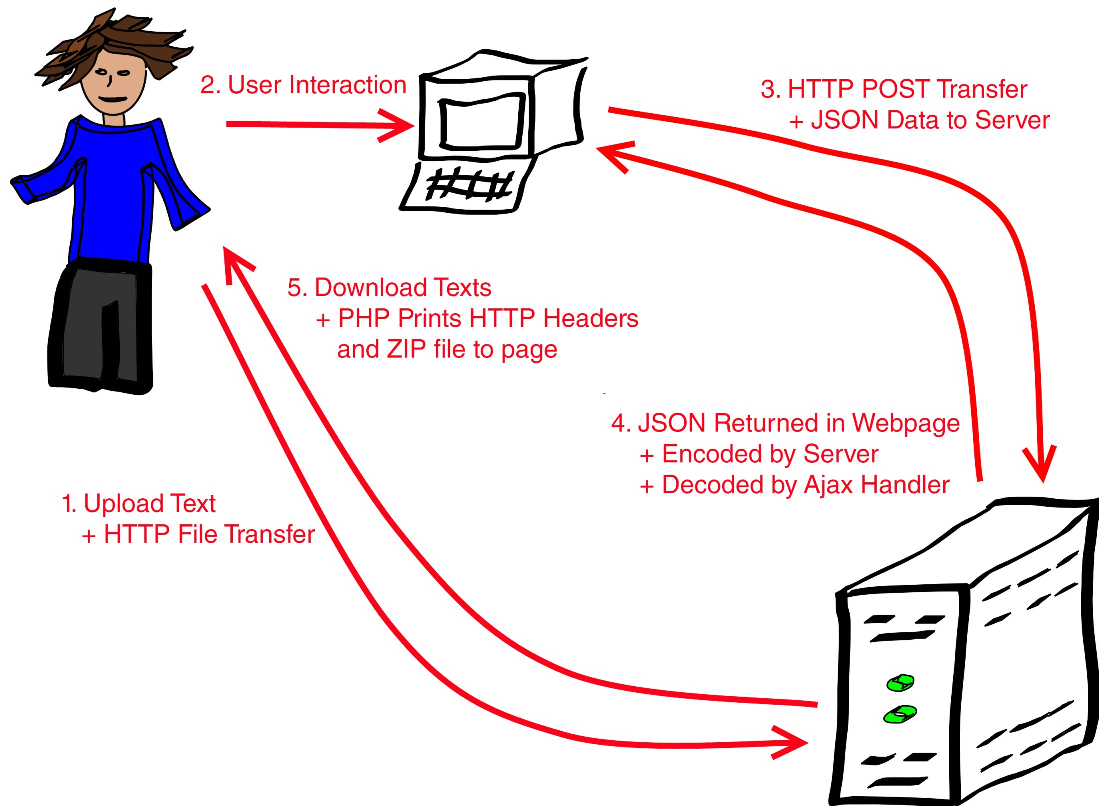


Figure 3.12: An overview of the diviText system. Numbers roughly correspond to the order in which actual processes take place.

Visual Cutter Panel. It also updates the id of the text currently contained in the Visual Cutter Panel with the `id` parameter to the function.

3.2.1.2 reset

The `CutterPanel.reset()` method removes all current segmentation parameters in the text displayed in the `CutterPanel`.

3.2.1.3 getSpaces

The `CutterPanel.getSpaces()` method returns a sorted array of all space breaks in the current text that define the space between each segment.

3.2.1.4 threeParamSpacer

`CutterPanel.threeParamSpacer(size, shift, last)` is one of the automatic cutter functions. This uses the `size` parameter to define fixed-size chunks in the current text. This size is an integer greater than or equal to 1. The `last` parameter specifies the minimum proportion of the final chunk between values 0 and 1. The last chunk can be no smaller than $size \times last$ words. The `shift` parameter is unused in this implementation.[1]

3.2.1.5 oneParamSpacer

`CutterPanel.oneParamSpacer(chunks)` is the second automatic tool that creates `chunks` number of chunks in the current text. This method sets up parameters for `CutterPanel.threeParamSpacer` calculating $size = round(\frac{N}{chunks})$. The `last` parameter is defaulted to a value of 0.5.

3.2.2 PHP

Server-side PHP scripts are invoked by the diviText JavaScript through the `Ext.Ajax` module. Data is sent from the browser to the server in the form of HTTP POST messages. The server responds with a page of text formatted as JSON (JavaScript Object Notation) data so the browser can parse a JSON object from the text. Specific scripts are invoked to perform specific functions. This section outlines the PHP scripts that are invoked by Ajax calls and what the scripts expect as POST data.

3.2.2.1 gettexts.php

The `gettexts.php` script returns a list of texts and chunksets belonging to the user as nodes in a tree. Texts have attributes like: `text`, the display name of the text; `tid`, the internal id of the text; `size` the size of the text in bytes; and a `children` node. The `children` node holds information regarding all chunksets belonging to the text. Chunksets have attributes similar to texts.

The data returned is formatted properly to be immediately parsed and placed into the Text Manager tree in the user interface.

3.2.2.2 gettext.php

The `gettext.php` script returns the text of a specified file and the name of the text. A `textid` parameter is sent to the server as POST data corresponding to the

[1] The `shift` parameter cannot be used in this visual implementation. See Section 5.1 for mention of this and other functionality that can be added in the future.

internal id of the text.

3.2.2.3 uploadtext.php

To upload a text to the server, `uploadtext.php` is called. This script expects a `name` parameter to name the text and builds a unique text identifier. A file is expected to be sent from the browser to the server using HTTP POST file transfer. This file is then cached in the user's directory on the server for further access.

3.2.2.4 removetext.php

To remove a text from the server, `removetext.php` is called. The POST parameter `textid` is expected. This identifies the correct text to remove. Using this script permanently removes the text from the server.

3.2.2.5 chunk.php

The `chunk.php` script is used to cut a text into segments. This script expects three POST data items. The first is `textid`, the unique text identifier, as in most other scripts. The second item is `name`. This is the name of the new chunkset which will be displayed in the Text Manager. This is parsed to form a unique chunkset identifier. The final POST item expected is `spaces`. This is a JSON encoded array of spaces. This string is decoded by the PHP script on the server to get the chunk breaks.

3.3 White Box: Internal Components

This section is designed to provide an overview of how things in the diviText system are put together. Section 3.3.1 describes parts of the ExtJS framework that allow for easier discussion in later sections concerning the JavaScript components. Section 3.3.2 describes how PHP pages work in the diviText system and the main data structures employed. A section on error handling in both JavaScript and PHP is found in Section 3.3.3.

3.3.1 JavaScript and ExtJS 3.3.1

All of the client-side code is written in JavaScript and run through the web browser. Specifically, much of the layout of the page viewed by the user is built using the ExtJS 3.3.1 framework. This is a great advantage because ExtJS supports all modern web browsers including Internet Explorer 6+, Mozilla Firefox 1.5+, Opera 9+, and Chrome 3+. This compatibility means that the code must only be written once, and

ExtJS will take care of the many differences between browsers. Another advantage is data-driven page rendering. Given only a few instructions about how the data is formatted, the dataset itself and how to display the data, ExtJS can render high-quality tables, trees, grids, etc.

ExtJS has a hierarchy of basic and complex "components." A component is simply a JavaScript Object that inherits features from a line of ancestor classes. For instance, an `Ext.Panel` is an extension of the `Ext.Container` type which simply holds items to be rendered to the page. The `Ext.tree.TreePanel` class is a descendent of the `Ext.Panel` type. This tree can use XML or a JSON object retrieved from the server or hard-coded into the page to render a hierarchical tree. Nodes in the tree can be moved, added, deleted or changed. The TreePanel has a few associated, but not inheriting classes like `Ext.tree.AsyncTreeNode` and `Ext.tree.TreeSorter` to handle rendering and alteration of data in the tree. It should be noted that the name of the class is not indicative of class hierarchy but a logical arrangement of associated classes.

The main body of code is stored in `divitext.js`. This file contains all of the logic to build the page itself and communicate with the server. `CutterPanel.js` contains the logic to render the Visual Cutter Panel and all cutting options.

When instantiating any component, it is common to see some code like:

```
1 var myPanel = new Ext.Panel({
2     title: "Cool Panel",
3     id: "mypanel-id",
4     ... some more configuration options ...
5     items: [ ... Ext components ... ]
6 });
```

This code creates a new `Ext.Panel`, a simple container for more complex objects, and places it in the variable `myPanel`. The parameter to the constructor of `Ext.Panel`, and all Ext objects created with `new`, is an object (set of key, value pairs) that configures the new component. Most configuration options override any option defined in the component definition. Common configuration options are `title`, `id`, and `items`.

The `title` option specifies the title of the object. In most components, this is displayed on the page.

The `id` option is a unique name that identifies the object to Ext. Using the `Ext.get(id)` or `Ext.getCmp(id)`, where `id` is a string, Ext will return an `Ext.Element` or a component, respectively. These are useful to retrieve the component's HTML, or component itself once it has left scope.

The `items` option is an array of components. The array entities within `items` are subsequently added, or drawn within the component.

Custom components are built on top of existing Ext components by using the `Ext.extend` method. Typically this method takes two parameters: first, an existing

Ext component to augment; and second, an object that specifies the parameters to use when building the custom component, that is, what makes the custom component unique. A simple example of this follows.

```
1 CustomPanel = Ext.extend( Ext.Panel, {
2     title: "My Sweet Custom Panel"
3 });
```

This example builds a custom component called `CustomPanel` which augments the basic `Ext.Panel`. This custom component specifies a default title "My Sweet Custom Panel" which will be displayed when the object is rendered to the page. To use `CustomPanel` in code, simply instantiate it:

```
1 var myPanel = CustomPanel({
2     id: 'mypanel2-id',
3     items: [ ... ]
4 });
```

If the `title` configuration option was set in the instantiation, that would override "My Sweet Custom Panel" defined in the definition of the `CustomPanel`.

This method of creating custom objects has the advantage of creating components in separate files, instantiating them when needed, and providing the ability to easily reuse, recycle, and augment new components. One major disadvantage is that the build process for these components is different, and more complex than simple instantiation.[2] Constructors and initiation functions must be defined to render `items` without error.

3.3.2 Hypertext Preprocessor (PHP)

The server-side software is written in PHP, specifically intended to be run using PHP version 5.1.6 (an old but very stable version). One major disadvantage to this version of PHP is the inability to use UTF-8 text.[3] Many special, and language-specific characters can break the system to the point that not all of the user's text may be saved to the server (see Section 5.1).

This section addresses how the PHP scripts are designed to work and the data structures that handle Users, Texts, and Chunksets.

3.3.2.1 Scripts

Each PHP page starts out similarly:

[2] This complexity is left to the reader to pursue but is used heavily in the code.

[3] This prevents many non-English texts from being uploaded. To deal with special characters in SGML-formatted Anglo-Saxon texts, an ASCII encoding of characters is created. For instance, þ(thorn) is represented by `&t;`. PHP version 6 is slated to support UTF-8 natively.

```
1 $HOME = "../..";
2
3 require_once( "$HOME/includes/nav.php" );
4 require( $MODTEXTS );
5
6 session_start();
7 login();
```

The HOME variable defines a relative path from the script to the root of the diviText web site. This helps build links between pages from a fixed point of reference. Here this script is located in a directory two levels removed from the root.

The file /includes/nav.php is required. Using HOME, the script can find this required navigation script which is located in the includes directory which is a directory immediately off of the root. This script generates links to many scripts using the HOME variable to build correct paths.

Line 4 requires the file stored in MODTEXTS. This is a variable defined in the navigation page that contains the path to the file that holds the definition of the Text class described in Section 3.3.2.3. The path to the file is ../../modules/texts/text.php.

The call to session_start starts logging the user's session. The subsequent call to login logs the user in. This process is described in Section 3.3.2.2.

Then, the main part of the script occurs. These processes are described in Section 3.4. Error handling is described in Section 3.3.3.1.

If the page is part of an Ajax request (for example, to load a text), the page ends with:

```
1 echo json_encode( $message );
```

This prints a message that is read by the Ajax handler as described in Section 3.3.3.1.

3.3.2.2 Users

Data for a user is stored in the temporary PHP $_SESSION variable. This variable lasts as long as the user remains active at the web page. If the user closes their browser, they are no longer guaranteed to be able to recover their data from the current session.

This design is on purpose. It allows the system to hide the user's unique identifier. Since the $_SESSION variable is not explicitly visible to the user, all the information contained within is hidden from the user and any malicious users.[4] A cookie in the user's browser stores the unique token identifying a PHP session which is sent to the server to identify the session and allow access to the user's data.

[4] This is not entirely true. If a malicious user gains access to the browser cookies of a user (using a tool like FireSheep), the malicious user can act as an imposter.

At the start of each script, prior to HTTP headers being sent back to the browser,[5] a PHP server-side call is made to session_start(). This indicates that either, a PHP PHPSESSID cookie is present and this is a continuation of an active session, or the cookie does not exist indicating that this is a new session. Following this is a call to login() which ensures the array $_SESSION['user'] exists. If this array does not exist, it must be built in a call to new_session().

```
1 function new_session()
2 {
3     $_SESSION['user'] = Array();
4     $_SESSION['user']['id'] = uniqid( 'divi_' );
5     $_SESSION['user']['dir'] = DIVI_DIR . "/" . $_SESSION['user']['id'];
6     $_SESSION['user']['texts'] = Array();
7     return mkdir( $_SESSION['user']['dir'], 0700, true );
8 }
```

This $_SESSION variable is an associative array. The user's data is stored as a value of the $_SESSION['user'] key, which is itself made to be an associative array. This is populated with three data items.

The first is a unique id identifying the user stored in $_SESSION['user']['id']. The id is generated by PHP's uniqid function with the parameter 'divi_' to be prepended to each id returned. The id is based upon the current time. The final id may look like divi_4da361787f122. This serves to generate a working directory for the user.

The directory is then stored in $_SESSION['user']['dir']. The directory for each user is stored in /tmp/divitext, a directory created to be readable by only the web server software. The value of this key will contain /tmp/divitext/divi_4da361787f122 for the id above. A portion of this directory is shown from an active server in Figure 3.13.

The $_SESSION['user']['texts'] value is initialized to an empty array. The array will hold Text objects and will be indexed by the Text's id.

This 'user' key is assumed to be present in every script. Without it, there is no user, and thus nowhere to store data. Login functions at the top of each PHP page should guarantee this key to be present.

3.3.2.3 Texts

Texts uploaded to the server are handled by a Text class. This class stores information about where the text is located, its name, size, directory, and id.

The text also contains an array of Chunkset objects. This array is similar to the user's array of texts, but contains Chunkset objects indexed by the chunkset ids.

This class also provides a static method for generating these ids.

[5]HTTP headers define the page that is being transfered from the server to the browser. The meaning of this has no bearing on the reader's understanding of the rest of this text.

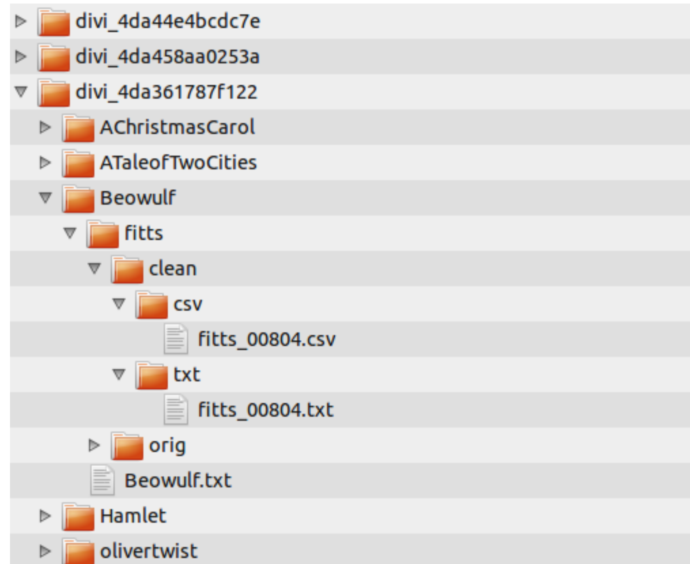


Figure 3.13: The contents of the `/tmp/divitext` directory from a live diviText server. User `divi_4da361787f122` has uploaded five texts. The *Beowulf* text directory has been expanded to show part of the *fitts* chunkset.

```
1 static public function id_from_name( $name )
2 {
3     $cname = "$name";
4     $cname = preg_replace( "/\.[a-zA-Z]{2,4}$/", "", $cname );
5     $cname = preg_replace( "/[^0-9A-Za-z_]/", "", $cname );
6     $id = "$cname";
7     return $id;
8 }
```

This method first parses out a possible file extension (Line 4) and removes all punctuation and whitespace except for underscores (_) using regular expression replaces (Line 5).

Methods are also provided to quickly read the original file into a string and to add and remove chunksets.

3.3.2.4 Chunkset

The `Chunkset` class handles chunksets created by the user using the diviText tool. This class is very similar to the `Text` class. Data members include the chunkset id, name, and directory. The array of spaces that created the chunkset is also stored (see Section 3.1.3). Methods include chunking methods, methods to write chunks to text files, and comma separated value (CSV) files.

3.3.3 Error Handling

This section discusses error handling that might occur in JavaScript and PHP through user's (mis)use of the diviText tool.

3.3.3.1 Errors In PHP

Error messages, for the most part, are handled manually. In many scripts, it is common to see:

```
1 $errors = null;
2 ...
3 if ( error conditions )
4     $errors[] = "Some error message.";
5 ...
6 if ( !$errors )
7     progress in the script
```

This creates a local variable `errors` and initializes it to `null`. Later, if some error condition has been met in Line 3, then Line 4 pushes the error message string "Some error message." into the array of errors. By not declaring `errors` as an array type, if no error messages have been pushed into `errors`, the `if` conditional in Line 6, will evaluate to *not null* (i.e. *true*), and the script can progress.

Later, in the script that prints output to the page, errors are printed if any errors occurred.

```
1 $message = null;
2 if ( $errors )
3 {
4     $message['success'] = false;
5     $message['errors']  = $errors;
6 }
7 else
8 {
9     $message['success'] = true;
10    $message['username'] = $username;
11 }
12 echo json_encode( $message );
```

This basic snippet of code prints a success status. If errors happened during execution, the `success` key of the message is set to `false` and the `errors` key is set to the array of errors. If no errors were caught in the `errors` array, then `success` is set to `true`. This script returns the username of a user if the script was successful. (See the next section for how this is handled by the browser.)

Error messages generated using this technique should be careful not to reveal any sensitive server information because these errors are displayed to the user, as described in the next section. Sensitive information may include: the user's unique id, database login information, server information, and so on.

Errors are also logged to the server's error logging file using the `trigger_error` method. In most instances, this method is passed text describing the error.

The error message logged by `trigger_error` can be more detailed because the error is not shown to the user, but logged to a secure log file. Information logged should be specific enough to aid in debugging. Details can include directories and variable contents among other sensitive information.

3.3.3.2 Handling PHP Errors In JavaScript

This section describes how errors returned from PHP scripts on the server are handled by the JavaScript at the browser. Many Ajax requests generated by the JavaScript include `success` and `failure` callback methods. These methods are executed after the Ajax request finishes.

A simple Ajax request to get a username may look like:

```
1  Ext.Ajax.request({
2      url: 'getusername.php',
3      success: function(r,o) {
4          var response = Ext.decode( r.responseText );
5          alert( response.username );
6      },
7      failure: function(r,o) {
8          var response = Ext.decode( r.responseText );
9          report_errors( response.errors, "Username Get Failure" );
10     }
11 });
```

A request is made to the `getusername.php` script. The script will return a JSON object encoded as a string. The string is then parsed for data which indicates one of two things: success or failure.

If the script is successful, the JSON object response may look like:

```
1 {
2     success: true,
3     username: 'jimbob'
4 }
```

This triggers the `success` method because of the key `success` and its value of true. The username is also stored in the object. Lines 3-6 in the Ajax request decode the response, stored in `r.responseText` and alert the username.

If the script failed, the JSON object response may look like:

```
1 {
2     success: false,
3     errors: [
4         "Database connection could not be established.",
5         "Backup database offline."
6     ]
7 }
```

This JSON object is a failure object because the key `success` was set to false. An array of errors is present. The `failure` callback of the Ajax request calls the function `report_errors` with the array of errors and the title “Username Get Failure”.

```
1 function report_errors( errors, title ) {
2     var out = "";
3     for ( var i = 0; i < errors.length; i++ )
4         out += "\n" + (i+1) + ". " + errors[i];
5     Ext.Msg.alert( title, out );
6 }
```

This method generates an `Ext.Msg.alert` dialog box with the data from the `errors` array in the body with the supplied title. This will alert the user that some error happened and depending on the error, what they may be able to do to resolve the issue. In this case, error messages report that no database could be contacted.[6]

The PHP script generates the JSON object that the request uses to call the appropriate callback method and stores the data used by the handlers. This was described in the previous section.

3.4 Code Walk-Through

This section walks through much of the code as it is executed. First, the process of building the page is described.

3.4.1 Page Load

diviText is loaded upon request of the diviText homepage `index.php`. This loads only the shell of the page. Parsing the HTML, the browser runs across `link` and `script` tags that specify required page inclusions for CSS (Cascading Style Sheet) and JavaScript, respectively. Once these files have been successfully loaded, the JavaScript can be executed.

The first executable line of code is

```
1 Ext.onload( function() { ... } );
```

This function builds the main display area of the page, an `Ext.Viewport` with the `border` layout engine. This layout engine specifies north, south, east, west, and center regions within the page and allows (in this configuration) resizable east, west and center elements. East, center, and west regions are populated with different Ext components. North and south regions are populated with static `header` and `footer` HTML `div` elements, respectively.

3.4.1.1 West

The west region lies on the left side of the page. This houses the Text Manager and buttons to upload and download texts.

[6] This database connection error sample is not part of any production code in diviText.

Text Manager The Text Manager User Interface (UI) element is a TextViewer component with the id text-viewer. This component is an extension of the Ext.tree.TreePanel component. Prior to rendering, data is loaded from the response text of an Ajax request to modules/texts/gettexts.php page (texts and chunksets that fill the tree). In addition to loading the data, listeners are set up to handle left and right-clicking nodes. These listeners use functions also (defined in TextViewer) to handle selecting, removing and displaying context menus (explained in more depth in Section 3.4.8).

In order to be added to the list of components to be rendered to the page, the following code is added to the definition of the west component.

```
1 items: [{
2     layout: 'fit',
3     border: false,
4     items: [ new TextViewer({id:'text-viewer'}) ]
5 }]
```

In this case, items is an array of Ext.Panel objects (these are not defined explicitly as panels, but Ext will assume any object defined without explicit reference to object type as a Ext.Panel). The subpanel is rendered to fill as much space as possible using the 'fit' layout without a border. The subpanel itself holds the TextViewer object (as seen in Line 4 with an explicit instantiation of the TextViewer component). This nesting is needed to ensure that the TextViewer is as large as the west region will allow.

Upload/Download Buttons The west region also contains buttons for uploading and downloading texts. The implementation of these functions are described in Sections 3.4.5 and 3.4.6.

To be rendered to the page, these buttons are defined in the definition of the west component as items of the footer's toolbar (of component type Ext.Toolbar), fbar.

```
1 fbar:[{
2     text: "Upload New Text",
3     handler: Uploader,
4     icon: 'icons/book_add.png'
5 }, '-', downloadButton ]
```

Again, this is an array of items, here assumed to be of the Ext.Button type if not explicitly defined. This means that the "Upload" button is lazily added to the footer (fbar). The text configuration option specifies the rendered button text. The handler option specifies a function call that will occur when the button is clicked (here a call to Uploader()). The handler's details are explained in Section 3.4.5. An icon that will be placed to the left of the button text (left being the default) is specified as the relative path to an image.

A string is the second object in `fbar`. This string, '-', when parsed in the context of an `Ext.Toolbar` indicates a vertical spacer is to be added.

The final object in `fbar` is the variable `downloadButton`. This holds an `Ext.Button` defined previously in the code (see Section 3.4.6 for more details). This variable was used simply to cut down on the levels of nested definitions.

3.4.1.2 East

The east region of the page, locally referred to as the main east region, is itself defined as a 'border' layout so that the Chunk Viewer and Cutting Tools sections can be resized vertically.

Chunk Viewer The Chunk Viewer portion of the UI is lazily defined as the first item in the main east region.

```
1 {
2     title: 'Chunk Viewer',
3     split: true,
4     region: 'center',
5     layout: 'vbox',
6     layoutConfig: {
7         align: 'stretch',
8         pack: 'start'
9     },
10    items: [
11       Ext.apply( cutter.updater, {flex:1} )
12    ]
13 }
```

The `split` option enables vertical resizing of the window. This region is defined as the center region of the main east region with a 'vbox' layout. The `layoutConfig` option is an object that contains more fine-tuning options. These specify to pack objects starting from the top, stretching them to fill the container horizontally.

The Chunk Viewer is itself the lone item in this center region defined previously in the code as the `updater` to the `cutter` object. The `cutter` object is the Visual Cutter Panel of type `CutterPanel` described in detail in Section 3.4.2. The `updater` object is a custom component `ChunkViewer`. In order to ensure that the Viewer (`updater`) fills the vertical region entirely, the `flex` parameter is applied with a value of 1, giving the updater priority vertical fill. The `ChunkViewer` is an extended `Ext.list.ListView`. Data in the list is defined as a JSON Store with three fields:

```
1 store: new Ext.data.JsonStore({
2     root: 'chunks',
3     fields: [
4         {
5             name: 'chunk',
6             type: 'int'
7         },
```

```
8         {
9             name: 'start',
10            type: 'int'
11        },
12        {
13            name: 'end',
14            type: 'int'
15        }
16    ]
17 })
```

Each field has two keys: a `name` that defines where that data member is located in the data, and a `type` for sorting purposes. The `root` configuration option specifies where the relevant array of data is in the loaded data object. An example of this object is:

```
1 {
2     chunks: [{
3         chunk: 1,
4         start: 1,
5         end: 42
6     },{
7         chunk: 2,
8         start: 43,
9         end: 101
10    }]
11 }
```

An array of `chunks` is parsed for objects containing the three `names` defined in the store.

The list is formatted be specifying columns to be displayed.

```
1 columns: [{
2     header: 'Chunk',
3     dataIndex: 'chunk',
4     align: 'right'
5 },{
6     header: 'Start',
7     dataIndex: 'start',
8     align: 'right'
9 },{
10    header: 'End',
11    dataIndex: 'end',
12    align: 'right'
13 },{
14    header: 'Length',
15    tpl: '{[values.end-values.start+1]}',
16    align: 'right'
17 }]
```

Each column is defined by the array `columns` in `ChunkViewer`. The first three columns are defined simply with a `header` specifying the displayed name of the column, a `dataIndex` referencing the items in the store, and an `align` specifying the alignment of the number in the cell. The final column is defined as a template that contains a compilation of values. Since `length` is defined as $end - start + 1$, there is no need to waste space storing the value in the store, so it is computed on

the fly in `tpl`. This is a string that is applied to an `Ext.XTemplate`. The value is contained in the curly braces, but since length is a mathematical operation, it must be placed in square brackets, the `XTemplate` notation for arbitrary code execution.

Cutting Tools The Cutting Tools panel is the south region of the main east region, a panel containing an `AutoCutter` component, an extension of `Ext.Container`.

```
1 {
2     title: 'Cutting Tools',
3     height: 400,
4     region: 'south',
5     split: true,
6     layout: 'fit',
7     items:[
8         new AutoCutter({
9             cutterPanel: cutter
10        })
11    ]
12 }
```

The height of the panel defaults to 400 pixels. The sole item of this panel is an `AutoCutter` with the main Visual Cutter Panel (the same `cutter` variable described previously) as the parameter `cutterPanel`.

The `AutoCutter` component defines three panels to be layed out in an 'accordion' layout. There is one panel for each type of cutter: simple, advanced, and visual.

The panel for the simple cutter in the items array of `AutoCutter` is defined lazily as a panel.

```
1 {
2     title: 'Simple (by # of Chunks)',
3     border: false,
4     items: [ new SimpleCutter({
5         cutterPanel: cp
6     }),{
7         contentEl: 'help-simple',
8         border: false
9     }]
10 }
```

This simple cutter panel contains two items, the second is a panel to which the HTML `div` with the id 'help-simple' is rendered. This `div` element is part of the `index.php` page.

The first is a `SimpleCutter` component that defines a form to allow simple cutter input. The sole configuration option to the new `SimpleCutter` is a reference to `cutter` above but stored locally in the variable `cp`.

The `SimpleCutter` component is an extension of a `CutterType` component which itself extends an `Ext.form.FormPanel` and defines a "Cut" button which responds to the "Enter" keypress. Because `SimpleCutter` is an extension of a form with a button, the only item `SimpleCutter` requires is a form field:

```
1 items: [ new EnterField({
2     xtype: 'numberfield',
3     fieldLabel: 'Chunks',
4     name: 'chunks',
5     allowNegative: false
6 }) ]
```

The EnterField component clicks the "Cut" button on pressing the "Enter" key. The xtype option specifies that the field is to contain only numbers using another form of component instantiation rarely used throughout the rest of the code that references a registered string name of a component. The string 'numberfield' is registered to correspond with the component Ext.form.NumberField. The name option specifies how to reference the field when submitting the form. The field also does not allow negative numbers.

The setup of the advanced cutter tool is similar to the setup of the simple cutter accordion panel, except that instead of SimpleCutter, AdvancedCutter is used. This too extends CutterType. The form fields that make up AdvancedCutter in the items array are:

```
1 new EnterField({
2     xtype: 'numberfield',
3     fieldLabel: 'Size',
4     name: 'size',
5     allowNegative: false
6 }),
7 new Ext.form.SliderField({
8     fieldLabel: 'Last Proportion',
9     name: 'last',
10    minValue: .01,
11    maxValue: 1,
12    increment: .01,
13    decimalPrecision: 2,
14    value: .5
15 })
```

The first field is similar to SimpleCutter's sole field. The second field is an Ext.form.SliderField. This creates a slider that ranges from .01 to 1 in increments of .01. The field defaults to .5.

3.4.2 Center: CutterPanel

The majority of page real estate is the center region of the screen. This section is the CutterPanel. This panel has two portions, the Visual Cutter area and a footer toolbar that contains the "Chunkset Name" field, and the "Save Chunkset" and "Reset" buttons.

3.4.2.1 DiviCutter

The DiviCutter component defined in divitext.js file is an extension of the CutterPanel component. This extension defines the footer toolbar and functions to load a default text and submit a chunkset to the server for saving since CutterPanel is only responsible for displaying, cutting, and storing chunks.

Upon loading the page and subsequently instantiating a DiviCutter component, the getDefaultText method is called. This creates an AJAX request for the file defaultText.txt which contains the default text seen upon loading the page.

```
1 getDefaultText: function() {
2     var cp = this;
3     Ext.Ajax.request({
4         method: 'POST',
5         url: 'defaultText.txt',
6         success: function(r,o) {
7             cp.newText( r.responseText, null );
8             var ftb = cp.getFooterToolbar();
9             if ( ftb )
10                ftb.disable();
11        }
12    });
13 }
```

If the text is successfully loaded, the success method is called with two parameters, r and o, response and request options, respectively. The text contained in defaultText.txt is stored in r.responseText. This is used as the first parameter in the call to the newText method of the CutterPanel. If the footer toolbar (ftb) exists, it is disabled, because the user should not be able to save chunksets of the default text.

The footer toolbar is built in DiviCutter's initComponent method. This method is called upon instantiation of the DiviCutter component (it is also called, if defined, in any component created thusly). The buttons and field are built just like previous buttons and fields.

```
1 {
2     text: "Reset",
3     icon: 'icons/arrow_undo.png',
4     handler: cp.reset,
5     scope: cp
6 }
```

One note is the use of the scope configuration option when creating buttons like this reset button. By defining this option, the handler operates under the specified scope. Here, this in the function cp.reset refers to the object pointed to by cp, the DiviCutter being created. Without explicitly defining scope, this in cp.reset would refer to the button and not, as the method assumes, the DiviCutter component.

3.4.2.2 CutterPanel

The CutterPanel component is the object that displays the text and keeps track of spaces. This component creates an Ext.Component type to add to its list of items to display the text. CutterPanel must be initialized with a textData configuration option. This contains the text to be rendered in the CutterPanel and is assumed to exist in all setup methods of the panel.

When a CutterPanel is first initialized, the makePanel method is called. This method creates a this.data object that contains data that aids in the rendering and functioning of the CutterPanel. This object contains: a string that acts as the base identifier to each word in the panel (this.data.cmp); an array of words in the text (this.data.textArray); a string containing tokens representing different space types (this.data.spaceString); the number of words and spaces and the sum of those; an HTML string that is used to render the text (this.data.html); and it also contains the array of selected spaces (this.data.selectedSpaces).

After the data is initialized, the cleanData method is called. This method simply replaces all instances of < and > with their renderable HTML counterparts < and > otherwise anything within a pair of those symbols would be treated as HTML tags, and not displayed.

Another method is called to split the text into an array of words.

```
textArrayify: function() {
    this.data.textArray = this.textData.trim()
        .replace( /[ \t\r\f\v\n]+/gi, " " )
        .split( /[ ]+/gi );
}
```

The textArray is generated by collapsing all runs of whitespace characters into a single space through a regular expression replace and then splits the text on these spaces creating an array of words.

The spaceStringer method is called to shrink all whitespace in the text into tokens representing the largest whitespace type in a run of whitespace characters. For instance, any run of whitespace with multiple consecutive newlines are collapsed into a single double-newline token d. Any run of whitespace with a single newline is tokenized as an n. Any run of whitespace with only spaces and tabs are represented as a t. And any run only containing spaces is reduced to a single s token. This is accomplished via regular expression replaces.

With textArray and spaceString, it is possible to build the HTML that renders the text, this.data.html. By iterating through the array of words HTML span tags are created that contain the word and the representation of the following associated space token. A word may be represented as:

```
<span title="103" id="cp-cmp-102"
    class="cp-space-highlight">flourished.<br/><br/></span>
```

This would represent word 103 (counting from 1) in the text. The span tag has a few attributes that aid in rendering and usability. The title attribute is the number of the word (again, counting from 1 for the sake of the user) and is rendered by the browser as title text, i.e. text that is displayed when hovered over for a long enough period. The id attribute is used by Ext to track the object on screen and to identify the word to the CutterPanel. The class attribute associates a CSS rule, in this case the rule to alter the appearance of the word when the user hovers over the word. The displayed text is "flourished." followed by two HTML breaks indicating that the word is at the end of a paragraph or stanza.

This is the case for all but the first and last words in the text. The first word does not employ the cp-space-highlight CSS rule because the first word always defines the start of a chunk. The last word has no trailing space.

After makePanel has finished and the CutterPanel component is initialized, it must be rendered. Prior to rendering, the beforerender event is fired. This event is picked up by its listener which simply calls this.paint.

```
1 paint: function() {
2     var comp = new Ext.Component({html:this.data.html});
3     this.add( comp );
4     this.data.html = "";
5 }
```

This function creates a new Ext.Component with the configuration option html set to the this.data.html created in makePanel. This component comp is added to the CutterPanel's items array via the standard add method. Nothing has been rendered yet. The this.data.html string is then blanked in an effort to save on memory.[7]

Similarly, the afterrender event is fired and picked up by its listener which simply calls the this.paint2 method. This method calls this.stylize and this.updateTable.

```
1 stylize: function() {
2     if ( !this.canFit() )
3         return;
4     for ( var i = 1; i < this.data.tN; i++ )
5     {
6         var e = Ext.get( this.data.cmp + i );
7         e.on( 'click', this.clickSpace );
8     }
9 }
```

this.stylize adds a click handler to each word in the text by iterating through all but the first word in the text, identified by the span's id attribute. Each word, which was given the HTML id attribute cp-cmp-# when created (#

[7]Blanking-out strings to save memory may or may not work. In theory, this would indicate to the Garbage Collector that the string is not in use and can be erased from memory. In practice, however, this may not be the case, at least immediately.

represents the index of the word in `this.data.textArray`), is retrieved using the `Ext.get` method which returns a `Ext.Element` object. The element can be modified, here by adding a `click` handler, to alter the way the element appears on the page.

The `updateTable` method is detailed in Section 3.4.3.1. This call sets up initial values in the Chunk Viewer table.

3.4.3 Clicking a Word

Each time a word in the Visual Cutter Panel is clicked, the method `clickSpace` in `CutterPanel` is fired.

```
1 clickSpace: function( e, t, o, tt ) {
2     var r = t;
3     if ( tt )
4         r = t.dom;
5     var p = Ext.getCmp( t.parentNode.id ).ownerCt;
6     var i = Number(t.id.match( /\d+/ )[0]);   // id # of space
7
8     // really want to select the previous space
9     if ( !tt )
10        i--;
11
12    // if space is unselected, select it
13    if ( p.data.selectedSpaces.indexOf(i) == -1 )
14        p.addSpaces( [ i ] );
15    else
16        p.removeSpaces( [ i ] );
17
18    p.updateTable(e,t,o);
19 }
```

This method retrieves the `id` of the clicked `span` tag on Line 6, passed to the function as the `target` parameter, and parses out the number. Since the actual data stored is the space before the word, the value of the id is decreased and stored in `i`. If `i` is in the `selectedSpaces` array, the break is to be removed using `removeSpaces([i])`. If `i` is not in `selectedSpaces`, then a new break is to be added using `addSpaces([i])`. The Chunk Viewer table is then updated. See Section 3.4.3.1 for an explanation of the `e`, `t`, and `o` function parameters.

3.4.3.1 Update Chunk Viewer Table

The `updateTable` method is often used to update the Chunk Viewer. This method has three parameters, `event`, `target`, and `object`. These are only useful in dertermining the scope of the function call. This could mean one of two things: (1) if `e`, `t`, and `o` are defined, the function is being called from the `clickSpace` method (since those are passed on click) and the scope is the scope of the Document Object Model (DOM) object clicked; or (2) if `e` is not defined, the call is from the `CutterPanel`

itself. Depending on the case, a few variables must be set accordingly to ensure that all the expected data can be accessed. Either way, a reference to the Visual Cutter Panel component is stored in the local variable `r`.

Once this has been ensured, rows can be created to fit in the store described in Section 3.4.1.2.

```
1 var list = r.updater;
2 var rows = [];
3 var spaces = r.getSpaces();
4 var end, start = 1;
5 var i = 0;
6 for ( i = 0; i < spaces.length; i++ )
7 {
8     rows.push({
9         chunk: i + 1,
10        start: start,
11        end: spaces[i] + 1
12    });
13    start = spaces[i] + 2;
14 }
15
16 rows.push({
17    chunk: i + 1,
18    start: start,
19    end: r.data.tN
20 });
21
22 list.getStore().loadData( { chunks: rows } );
```

The code segment above loops through all of the spaces that have been selected creating rows that correspond to the user's idea of word boundaries, starting at 1. The last line updates the store of the `ChunkViewer` with the new data and reloads.

Another loop iterates through each node in the viewer, updating the colors to correspond to the colors of the chunk in the Visual Cutter Panel:

```
1 var nodes = list.getNodes();
2
3 for ( i = 0; i < nodes.length; i++ )
4 {
5     var node = Ext.get( nodes[i] );
6     node.setStyle( 'background-color',
7             r.colors[ i % r.colors.length ] );
8 }
```

A call to `CutterPanel.recolor` updates the colors in the Visual Cutter Panel.

```
1 recolor: function( sp ) {
2     sp = 0;
3     var word = true;
4     var totalcolors = this.colors.length;
5     var color = 0;
6
7     var i;
8     for ( i = sp; i < this.data.tN; i++ ) {
9         word = Ext.get( this.data.cmp + i );
10
11        word.setStyle( 'background-color', this.colors[color] );
12
```

```
13          if ( this.data.selectedSpaces.indexOf( i ) == -1 )
14              ;
15          else
16              color = ( color + 1 ) % totalcolors;
17      }
18  }
```

The `recolor` method iterates through each `span` tag coloring the word with the appropriate color and updating the color if the index is in the array of selected spaces.

3.4.4 Automatic Cutters

When one of the automatic cutter tools is submitted, a call to the appropriate cutting function is called with values from the fields. When Simple Cutter is submitted, `CutterPanel.oneParamSpacer` is called with the number of chunks. When Advanced Cutter is submitted, a call to `CutterPanel.threeParamSpacer` is made with the size of the chunks, twice, and the last proportion size.

3.4.4.1 Simple Cutter

```
1 oneParamSpacer: function( chunks ) {
2     var size = this.data.textArray.length / chunks;
3     size = Math.round( size );
4     this.threeParamSpacer( size, size, .5 );
5 }
```

The `oneParamSpacer` method calculates the number of words needed to define `chunks` number of chunks and calls `threeParamSpacer` with the `size` arguments. The proportional size of the last chunk defaults to 0.5 meaning that the last chunk must be at least half of the size all of the other chunks.

3.4.4.2 Advanced Cutter

The `threeParamSpacer` method is used to perform both automatic cutting types.

```
1 threeParamSpacer: function( size, shift, last ) {
2     this.emptySpaces();
3
4     var spaces = [];
5     shift = size;
6     var curr = size - 1;
7     for ( ; curr < this.data.tN; curr += shift )
8         spaces.push( curr );
9
10    var cut = curr - shift;
11    var lastlen = this.data.tN - cut;
12    if ( ( lastlen / size ) > last && lastlen != 1)
13        ;
14    else
15        spaces.pop();
```

```
16
17     this.setSpaces( spaces );
18     this.updateTable();
19 }
```

The array of selected spaces is emptied so this operation overwrites all currently selected spaces. This simplifies some of the logic and standardizes the functionality of a call to this method.

The loop on lines 7 and 8 adds a break point each go-around `shift` number of spaces after the previous space. This creates a new array of spaces. After the loop, the length of the last chunk is calculated. If the length is too short, the last space placed into `spaces` is removed. The `selectedSpaces` array is updated through the call to `setSpaces`. The Chunk Viewer table is then updated.

3.4.5 Text Upload

As described in Section 3.4.1.1, when the user clicks the "Upload Text" button in the lower left of the interface, a dialog box is opened prompting the user to add a text.

The `Uploader` method handles the button click. This method creates a new `UploaderWindow` which contains an `UploaderForm`. It is this `UploaderForm` that handles all of the input and submission of the file to the server.

Two fields are added to the form in the `initComponent` method:

```
1 var namefield = new EnterField({
2     fieldLabel: 'Text Name',
3     name: 'name',
4     allowBlank: false
5 });
6
7 var filefield = {
8     xtype: 'fileuploadfield',
9     fieldLabel: 'File',
10    name: 'file',
11    listeners: {
12        fileselected: function(f,file) {
13            var name = file.match( /[^\\\/]+\..{3}/ );
14            if ( name )
15                namefield.setValue( name );
16            else
17                namefield.setValue( file );
18        }
19    }
20 };
```

The field created in the variable `filefield` is a special field that opens a browser/operating system dependant file browser that lets the user choose a file from their computer to upload. This field has a listener that changes the text of `namefield` when a file is selected. The `namefield` itself is a simple text field using the `EnterField` type discussed in Section 3.4.4 that submits the form when the "Enter" key is pressed.

Upon submitting the form, the `UploaderForm.upload` method is called. If the form is valid, such that the fields are filled, the `Ext.form.submit` method is used to send an Ajax request to the server with the name of the text and the text file itself.

```
1 this.getForm().submit({
2     url: 'modules/file/uploadtext.php',
3     method: 'POST',
4     waitMsg: 'Uploading text... May take a while.',
5     success: function(f,a) {
6         var tv = Ext.getCmp( 'text-viewer' );
7         tv._reload();
8         tv.getRootNode().expand( true );
9         Ext.getCmp( 'uploader-window' ).close();
10    },
11    failure: function(f,a) {
12        report_errors( a.result.errors, "Upload Error" );
13    }
14 });
```

This request sends the values contained in the field to the file on the server at `modules/file/uploadtext.php` via `POST`. A wait message is displayed while the file uploads until the server responds. If the server responds with success, the `success` callback method is called. This function informs the Text Manager to reload data from its source, as described in Section 3.4.7, and to expand the root node so the user sees their uploaded texts. The `UploaderWindow` is then closed. If the server responded with a failure message, the `failure` callback is called and an array of errors is reported as described in Section 3.3.3.

After the text has been transfered to the server, the `uploadtext.php` script is executed.

```
1 $newtext = new Text();
2 $id = $oid = Text::id_from_name( $_POST['name'] );
3 $i = 1;
4 while ( array_key_exists( $id, $_SESSION['user']['texts'] ) )
5 {
6     $id = $oid . "_$i";
7     $i++;
8 }
9
10 $errors = $newtext->set_data( $_POST, $_FILES['file'],
11     $_SESSION['user']['dir'], $id );
```

Immediately, an empty new `Text` object is created in `$newtext`. The `Text` class provides a static function that generates an id from the name the user gave the text. To ensure that no text has the same id, a loop checks that the id is not already the index to an existing text. If the id (or key) in `$_SESSION['user']['texts']` (an associative array) exists, an increasing number is appended to the id.

A call to `set_data` generates data in `$newtext`. The function uses PHP's `$_POST` and `$_FILES` global variables, the user's directory, and the text's id as parameters to place the text in the right location.

```
1 if ( !preg_match(  "/^text/", $file['type'] ) )
2 {
3     trigger_error( "Invalid filetype {$file['type']}." );
4     $errors[] = "Invalid filetype {$file['type']}.";
5     return $errors;
6 }
```

In the call to set_data, the file is checked for type using the segment of code above. If the text is not a plain text MIME (Multipurpose Internet Mail Extensions) type, the file is rejected and an error is logged and returned. This limits uploads to TXT files and a few other types like TEX (LaTeX) files.

```
1 $this->name   = $post['name'];
2 $this->size   = $file['size'];
3 $this->type   = $file['type'];
4 $this->id = $id;
5 $this->folder = $dir . "/" . $this->id;
6 $this->orig   = $this->folder . "/" .  $this->id . ".txt";
```

The local members of the Text class are set provided that the type is valid. The folder member is an absolute path to the directory containing the uploaded text, once moved, and orig is the original text file itself. Using these members, the text directory can be created and the file moved into the directory:

```
1 if ( !$errors && !mkdir( $this->folder, 0700 ) )
2 {
3     trigger_error( "Could not create text directory '{$this->folder}'." );
4     $errors[] = "Text dir failure.";
5 }
6
7 if ( !$errors && !move_uploaded_file( $file['tmp_name'], $this->orig ) )
8 {
9     trigger_error( "Could not move uploaded file to desination." );
10    $errors[] = "Could not move file.";
11 }
```

If either creating the text directory or moving the text fails, an error is triggered and logged.

```
1 if ( $errors )
2     rrmdir( $this->folder );
3 else
4     $this->clean_text();
5 return $errors;
```

If any error is logged, the script attempts to remove the directory and/or text, and the error is returned. If creating the directory and moving the text was successful, the text is cleaned:

```
1 public function clean_text()
2 {
3     $text = "";
4     $FH = fopen( $this->orig, 'r' );
5     while( !feof( $FH ) )
6         $text .= trim( ( fgets( $FH ) ) ) . "\n";
7     fclose( $FH );
8
```

```
 9     $text = preg_replace( "/\s*\n\n\s*/", "\n\n", $text );
10     $text = preg_replace( "/\s*\t\s*/", "\t", $text );
11     $text = preg_replace( "/[ ]+/", " ", $text );
12
13     $FH = fopen( $this->orig, "w+" );
14     $len = fwrite( $FH, $text );
15     fclose( $FH );
16
17     $this->size = $len;
18 }
```

Cleaning a text entails removing all unnecessary whitespace. A loop reads in the text one line at a time, removing whitespace from both ends of the line, and appending a single newline that was removed. Whitespaces are then collapsed into their largest form, similar to the way described in Section 3.4.2.2. The original file is then opened, truncated, and written back as the original text. The number of bytes written is the new size of the text.

If, in the end, the text was successfully saved and cleaned, `null` is returned from `set_data` and the text is saved to the user's list of texts:

```
1 $_SESSION['user']['texts'][$id] = $newtext;
```

The text's id is used as an index into an associative array of texts. A success message is written to the page and sent to the browser. Otherwise, a failure message will be written and returned to the browser.

3.4.6 Text Download

Clicking the "Download Texts" button triggers the handler saving a ZIP file of to all of the user's texts to their computer.[8]

```
 1 handler: function() {
 2     var body = Ext.getBody();
 3     var frame = body.createChild({
 4         tag:'iframe'
 5         ,cls:'x-hidden'
 6         ,id:'iframe'
 7         ,name:'iframe'
 8     });
 9
10     var form = body.createChild({
11         tag:'form'
12         ,cls:'x-hidden'
13         ,id:'form'
14         ,action:'download.php'
15         ,target:'iframe'
16     });
17
18     form.dom.submit();
19 }
```

[8] I did not write much of this code. The JavaScript is from *Saki's Extensions, Plugins and Know-How* at `http://www.extjs.us`. The PHP function `downloadFile` is from the PHP manual at `http://php.net/manual/en/function.header.php`.

This function creates a hidden frame and form on the page to call the script at `download.php`:

```
1 $FILE = "texts.zip";
2 $NEWDIR = "user.texts";
3
4 `mkdir $NEWDIR`;
5 `mv * $NEWDIR`;
6 `zip -r $FILE $NEWDIR`;
7 `mv $NEWDIR/* .`;
8 downloadFile( "$FILE" );
9 `rm -r $NEWDIR $FILE`;
```

This script uses Bash Shell commands to invoke programs on the server to move, zip, and remove files and directories. Commands wrapped in backticks (‘) invoke shell commands. If desired, the output of these invocations can be dumped into PHP variables.

In order to provide the user with a nice ZIP file that doesn't extract all the texts onto the desktop unexpectedly (a ZIP-bomb of sorts), a directory is created called `user.texts` (this folder cannot be in use by a user's text because all periods are removed from text ids that generate directories). All the user's text directories are moved into this new directory. This directory is then zipped up as `texts.zip`. The text directories are moved back to their original location, and the `user.texts` directory removed.

The file `texts.zip` is the file that will be downloaded via the `downloadFile` function. This function sends appropriate headers to the browser indicating that the browser is to download a ZIP file and prints the contents of the file. After the file is printed to the page, readying it for download, the ZIP file is removed.

3.4.7 Loading the Text Manager

Loading the Text Manager requires an Ajax call in the JavaScript by the `TextViewer` component (rendering was briefly described in Section 3.4.1.1). This is done implicitly both on load and reload.

```
1 root: {
2     id: 'texts',
3     text: 'Uploaded Library',
4     icon: 'icons/package.png',
5     expanded: true
6 },
7
8 dataUrl: 'modules/texts/gettexts.php'
```

The `root` configuration option in the `TextViewer` component specifies that the first node in the tree is always the "Uploaded Library" node and that it is always expanded. It also forces the tree to have a root, otherwise the tree would not be rendered properly.

The `dataUrl` configuration option implicitly specifies that the tree's loader is an `Ext.tree.TreeLoader`. When the tree is loaded for the first time, the loader is created and an Ajax request is sent to the `modules/texts.gettexts.php` script.

This invokes a script on the server that will return a page of encoded JSON data containing a valid JSON tree structure.

```
1 $texts = Array();
2
3 $i = 0;
4 // iterate through each Text object
5 foreach ( $utexts as $tid => $text )
6 {
7     // if the key in $tid points to nothing, no text object, continue
8     if ( !$text )
9         continue;
10    $texts[$i]['text'] = $text->GET_name();
11    $texts[$i]['tid']  = $tid;
12    $texts[$i]['size'] = $text->GET_size();
13    $texts[$i]['id']   = "texts/$tid";
14    $texts[$i]['type'] = "text";
15    $texts[$i]['icon'] = "icons/book.png";
16    $tcs = $text->GET_chunksets();
17
18    // if there are chunksets
19    if ( $tcs )
20    {
21        $j = 0;
22        // iterate through each Chunkset in the current Text
23        foreach ( $tcs as $csid => $cs )
24        {
25            // if the key in $csid points to nothing, continue
26            if ( !$cs )
27                continue;
28            $texts[$i]['children'][$j]['text']   = $cs->GET_name();
29            $texts[$i]['children'][$j]['tid']    = $cs->GET_id();
30            $texts[$i]['children'][$j]['spaces'] = $cs->GET_spaces();
31            $texts[$i]['children'][$j]['leaf']   = true;
32            $texts[$i]['children'][$j]['icon']   = "icons/page_white_stack.png";
33            $texts[$i]['children'][$j]['type']   = "chunkset";
34
35            $j++;
36        }
37    }
38    else
39    {
40        // the text has no chunksets and is a leaf
41        $texts[$i]['leaf'] = true;
42    }
43
44    $i++;
45 }
46
47 echo json_encode( $texts );
```

This loop iterates through each of the user's `Text` objects (Line 5) and builds an array of texts (Lines 10-15, 41), indexed from $i = 0$. If the text has one or more chunksets (Line 19), child nodes are built similarly to represent the chunkset(s) (Lines 28-33).

Each node in the tree must contain a few keys in order to be rendered properly by the `TextViewer`. The `id` key of the nodes representing texts identifies that the text hangs off of the `texts` node in the tree, i.e. the root node. The `text` key of each node is the text of the node displayed in the tree, i.e. the name of the text as entered by the user. Each node must contain the key `leaf`, which must be set to true if there are no child nodes (texts without any chunksets and all chunksets) or an array of `children` index by $j = 0$.

Each node must also store keys to add the functionality of the tree in the context of the diviText tool. The `tid` key references the unique text/chunkset identifier that will be sent to the server when loading texts and chunksets (see Section 3.4.8 for more about this process). The `size` key indicates the size of the text in bytes. This could be used to prevent texts of too large a size from being loaded, but it is currently unused. The `type` key specifies the type of node, whether the node is a text or chunkset. In chunksets, the `spaces` key is the array of spaces used to define the chunkset and allows chunksets to be reloaded into the Visual Cutter. The `icon` key points to an image on the server that the tree uses as the node's icon.[9]

The call to `echo json_encode( $texts )` properly formats the array of texts as a JSON object that may look something like the following after hand formatting:

```
1  [{
2      "text":"Fates of the Apostles",
3      "tid":"FatesoftheApostles",
4      "size":4874,
5      "id":"texts\/FatesoftheApostles",
6      "type":"text",
7      "icon":"icons\/book.png",
8      "leaf":true
9  },{
10     "text":"Christ",
11     "tid":"Christ",
12     "size":17367,
13     "id":"texts\/Christ",
14     "type":"text",
15     "icon":"icons\/book.png",
16     "children":[{
17         "text":"ChristABC",
18         "tid":"ChristABC",
19         "spaces":[1668,2252],
20         "leaf":true,
21         "icon":"icons\/page_white_stack.png",
22         "type":"chunkset"
23     }]
24 },{
25     "text":"Beowulf -- Formatted",
26     "tid":"BeowulfFormatted",
27     "size":147204,
28     "id":"texts\/BeowulfFormatted",
29     "type":"text",
```

[9]Icons are from FamFamFam's Silk Icon set. See Section B.1 for usage rights.

```
30      "icon":"icons\/book.png",
31      "children":[{
32          "text":"Beowulf, 42 Fitts",
33          "tid":"Beowulf42Fitts",
34          "spaces":[113,218,845, ... ,17151,17242,17265],
35          "leaf":true,
36          "icon":"icons\/page_white_stack.png",
37          "type":"chunkset"
38      }]
39  }]
```

The `TextViewer`'s loader knows how to parse this object and can properly display it.

The loader is also invoked when the `TextViewer._reload` method is called.

```
1 _reload: function() {
2     this.getLoader().load( this.root );
3 }
```

This method simply tells the loader to load the tree again from the root node. Since the `dataUrl` option was sent, the loader knows to reload the `gettexts.php` page and use the output as the new JSON object. This method is called whenever a text is uploaded and when a new chunkset is saved.

3.4.8 Clicking in Text Manager

When a text or chunkset in the Text Manager is clicked, the `click` event is fired and the `click` handler of the `TextViewer` component is called.

```
1 click: function( n, e ) {
2     if ( n != this.getRootNode() && n.attributes.type == 'text' )
3         this.selectText( n.attributes.tid );
4     else if ( n.attributes.type == 'chunkset' )
5         this.selectText( n.parentNode.attributes.tid,
6             n.attributes.spaces );
7 }
```

This listener get passed the clicked node (argument `n`) and the click event (argument `e`). If the node is not the root node, and is of type "text," then a node representing a text was clicked and the text must be loaded using the `selectText` method of the `TextViewer` component. If the node is of type "chunkset," a chunkset was clicked. This still results in a call to the `selectText` method, but to get the id of the text to load, the clicked node's parent node must be accessed. The array of spaces contained within the chunkset node is also sent to the `selectText` method so the chunkset can be reloaded properly.

The `selectText` method uses the following lines to generate the proper Ajax request:

```
1 Ext.Ajax.request({
2     url: 'modules/texts/gettext.php',
3     method: 'POST',
4     params: {
```

```
5           textid: id
6       },
```

The `selectText` method creates an Ajax request with POST parameters to the PHP script at `modules/texts/gettext.php`. The `params` object sent to the `Ext.Ajax.request` method defines a POST parameter `textid`. This parameter is assigned the unique text identifier that is to be retrieved, the variable `id` is the first parameter sent to the `selectText` method.

The `gettext.php` script simply checks that a `textid` parameter was sent in POST, and if the `textid` exists for the user. If both of these are true, the text of the text identified by `textid` is returned in a JSON object along with the text's name.

If the script returns successfully, the request's `success` method is called. This method decodes the JSON object returned by `gettext.php` and uses the text to populate the Visual Cutter Panel with a call to `CutterPanel.newText`.

If `selectText` received a second argument, an array of spaces because the clicked tree node was a chunkset, the spaces are given to the Visual Cutter Panel with a call to `CutterPanel.setSpaces`.

3.4.9 Save Chunkset

Saving chunksets is the most complicated of all actions performed by the server. In summary, an Ajax request is made to the server with POST data that is needed to cut the text, the text is cut, and the words in these cut files are counted and saved to the server.

```
1 finalize: function( name ) {
2         var textid = this.tid;
3         var spacestr = Ext.encode( this.getSpaces() );
4
5         if ( !textid || !name )
6             return;
7
8         Ext.Ajax.request({
9             url: 'modules/texts/chunk.php',
10            method: 'POST',
11            params: {
12                name: name,
13                textid: textid,
14                spaces: spacestr
15            }
```

When the user clicks the "Save Chunkset" button at the bottom of the Visual Cutter Panel, the `DiviCutter.finalize` method is called and passed the name of the chunkset to be created. The id of the current text is retrieved from the Cutter Panel, and the current array of spaces is encoded as a string in JSON notation.

An Ajax request is then made to the script `modules/texts/chunk.php`. Three parameters are sent, the name of the chunkset, the text's unique id, and the string-encoded array of spaces.

The chunk.php script makes sure that all the required data is present and that the text identified by textid is accessible. The string-encoded array of spaces is decoded back into an array. Using this information, the chunk method of the Text object indexed by textid is called with the array of spaces and the name of the chunkset.

This method generates a unique identifier for the chunkset similar to the way the text's unique identifier was set up in Section 3.4.5.

```
1 $cs = new Chunkset();
2 $cs->SET_name( $csname );
3 $cs->SET_id( $csid );
4 $cs->SET_folder( $this->folder . "/" . $csid );
5 $cs->chunk( $textstr, $spaces );
```

A new Chunkset object is created and some attributes are set similar to a Text's attributes. The folder attribute is created by creating a directory off of the text's directory with the chunkset's id as the name. The Chunkset::chunk method cuts a string ($textstr) on the requested spaces ($spaces).

```
1 $this->spaces = $spaces;
2 $cwd = getcwd();
3 mkdir( $this->folder, 0700 );
4 chdir( $this->folder );
5
6 $folders = array( "clean/txt", "clean/csv", "orig/txt", "orig/csv" );
7
8 foreach ( $folders as $f )
9 {
10     if ( !mkdir( "./$f", 0700, true ) )
11         $errors[] = "Could not make folder '$f'.";
12 }
13
14 $out = $this->chunker( $text, $spaces, "orig" );
15 $out2 = $this->chunker( $text, $spaces, "clean" );
16
17 chdir( $cwd );
```

The Chunkset::chunker method creates two directories off of the chunkset's folder, one for a clean version of the text (i.e. chunks without any punctuation or capitalization) and an original folder (i.e. chunks that retain original punctuation and capitalization, but not original whitespace). Two directories hang off of each of these, a directory containing word counts in Comma Separated Value (CSV) files and a directory containing the text of the chunks.

The Chunkset::chunker method is called twice, once to chunk without removing punctuation and once to chunk a clean text. The first few lines of chunker split the string containing the text into an array of individual words.

```
1 $text = collapse_spaces( $text );
2 $textarr = split_string( $text );
3 $chunksarr = split_on_spaces( $textarr, $spaces );
```

The call to collapse_spaces uses a regular expression replace to change all runs of whitespace with a single space character. The subsequent call to the method

split_string uses the fact that all words are separated by one and only one space to create an array containing only words.

The next call to split_on_spaces uses the array of spaces to subdivide the array of words into an array of of chunks, where each chunk is represented by an array of words in the chunk.

```
1 function split_on_spaces( $textarray, $spaces )
2 {
3     $chunkarray = null;
4     $spaces[] = count( $textarray ) - 1;
5
6     $start = 0;
7     foreach ( $spaces as $s => $sp )
8     {
9         $end = $sp;
10        $length = $s ? $end + 1 - $start : $end + 1;
11        $chunkarray[$end] = array_slice( $textarray, $start, $length, true );
12        $start = $end + 1;
13    }
14
15    return $chunkarray;
16 }
```

The split_on_spaces method adds the index to the last word in the text to the list of spaces, since the end of the final chunk was not added by the JavaScript CutterPanel component. The method then iterates through the array of spaces using each number in the array to determine the last word in the chunk, the length of the chunk and the first word in the next chunk. The $chunkarray variable is the array of chunks indexed by the index of the last word in the chunk. This saves the value for future use.

```
1 if ( $style == "clean" )
2     $chunksarr = remove_junk( $chunksarr );
```

Back in the chunker method, if the style was specified as “clean”, punctuation and capitalization is removed from each word in each chunk array using the remove_junk method.

```
1 $chunkhashes = null;
2 foreach( $chunksarr as $end => $chunkarr )
3     $chunkhashes[$end] = count_words( $chunkarr );
```

The array of words in each chunk is now ready to be transformed into an associative array (or hash table, referred to as a “hash”) of word counts indexed by word. Each array of words representing a chunk in $chunksarr is iterated through to create an array of hashes. This array of hashes is too indexed by the index of the last word in the index of the last word in the chunk.

```
1 function count_words( $textarray )
2 {
3     $wordcount = array();
4     foreach ( $textarray as $word )
5     {
6         if ( $word == "" )
```

```
7             continue;
8         $wordcount[ "$word" ] = isset( $wordcount[ "$word" ] ) ?
9                 $wordcount[ "$word" ] + 1 : 1;
10     }
11
12     return $wordcount;
13 }
```

The count_words method takes a single array of words representing a chunk and builds a hash of word counts by increasing the count of a word each time it appears in the array. If the word is the empty string, because it was all punctuation and thus the string emptied in remove_junk, it is skipped over. The hash of the counts indexed by word is returned.

Returning to chunker, the array of hashes is then iterated through and printed to files.

```
1 foreach ( $chunksarr as $end => $chunkarr )
2 {
3     $out  = $this->write_txt( $chunkarr, $endpad, $style );
4     $out2 = $this->write_csv( $chunkhashes[$end], $endpad, $style );
5 }
```

Each chunk array and chunk hash is then printed to a file with the chunkset's id followed by the index of the last word in the chunk. Note that the index is padded with zeros at the front so the files are properly sorted by name by the user's operating system (not shown in code above).

To write the text file, the array of words is simply imploded (words are concatenated) with a space separating each word in write_txt. This is the reason that original whitespace is not preserved.

Writing the CSV file is slightly more complicated. The write_csv method is shown below:

```
1 $total = get_total( $hash );
2 $unique = get_unique( $hash );
3 $hapax = get_hapax( $hash );
4
5 hash_sort( $hash, 'c' );
6
7 $csv = $this->id . "txt,$total,$unique,$hapax\n";
8 $csv .= "RANK,WORD,COUNT,RELATIVE FREQUENCY\n";
9 $rank = 0;
10 $allrank = 1;
11 $prevcount = null;
12
13 foreach ( $hash as $word => $count ) {
14     $prop = round( ($count / $total), 10 );
15
16     if ( $count != $prevcount )
17         $rank = $allrank;
18     $prevcount = $count;
19     $allrank++;
20
21     $csv .= "$rank,$word,$count,$prop\n";
22 }
```

A few statistics must be calculated: the number of total, unique, and hapax (words that appear only once) words are calculated. The hash is then sorted by word count and subsorted so that the words with equal counts are in (English) alphabetic order. Some header information is then stored to the `$csv` variable that will be printed to the file.

The list of words, sorted by count, is then iterated through. The relative frequency is calculated for each word in the chunk. If the count of the current word is not the same as the count of the previous word, the rank of the word is updated. This allows all words with the same count to be assigned the same rank. The current word's line in the CSV file is then appended to `$csv`.

If everything succeeded without error, the new `Chunkset` object is added to the array of chunksets in the user's `Text` object indexed by the chunkset's unique id. Success is returned to the page.

The `TextViewer` component is then told to reload its data and thus the new chunkset is added to the tree.

Chapter 4

Use Cases

This chapter is a guide to the user showing some basic examples using the diviText tool. Section 4.1 shows a toy example using all three cutting methods: Visual, Simple, and Advanced. Section 4.2 shows a real-world use by showing how to cut the Anglo-Saxon text *Beowulf* into fitts and what the user gets after downloading the results. Other potential use cases are mentioned. Note that high contrast colors have been used in these examples to make the chunks stand out more in print.

4.1 Toy Example

In the following sections, examples of the Visual, Simple, and Advanced cutting tools are illustrated through the use of a short piece of the *lorem ipsum* text commonly found as placeholder text in graphic design.

Before using any cutting tools, the user must upload their text to the server. Once the diviText interface is loaded, the user would select the “Upload Text” button in the lower left of the interface (refer back to Figure 3.1). This opens the upload dialog box shown in Figure 4.1.

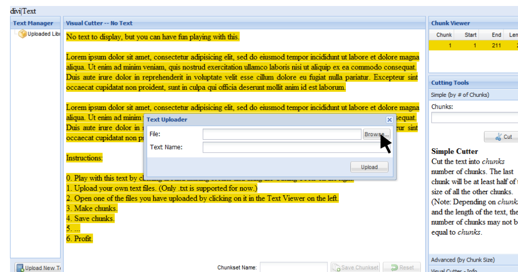


Figure 4.1: The Upload Text dialog box. The mouse indicates the “Browse...” button that opens a window to select a file from the user’s computer.

Clicking the "Browse..." button opens a browser/operating system specific "Open" window (a Chrome and Windows 7 "Open" window is shown in Figure 4.2). The `loremshort.txt` file has been selected and the "Open" button is about to be clicked. This returns control back to the Upload dialog box.

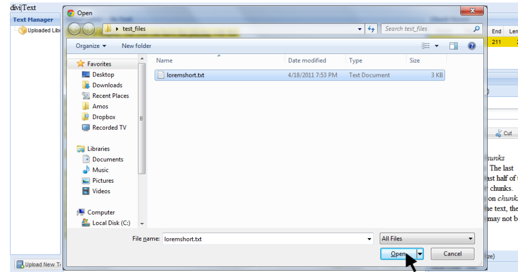
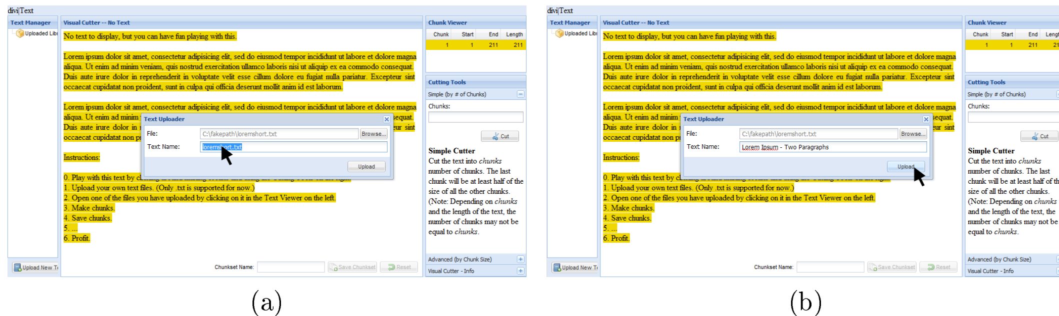


Figure 4.2: The Open file window. The file "loremshort.txt" has been selected and the "Open" button is about to be clicked.

Figure 4.3a shows that the file is ready to be uploaded and a name has been automatically filled in. In this example, assume that this default name is unsatisfactory, so by double-clicking in the "Text Name" input field, the name is highlighted and ready to be overwritten. The text is then renamed "Lorem Ipsum - Two Paragraphs" (shown in Figure 4.3b). Once overwritten, the user is ready to upload the file and does so by clicking the "Upload" button (also shown in Figure 4.3b).



(a) (b)

Figure 4.3: The user has decided that the automatically generated text name was not satisfactory and double-clicked the field (a) and set the name to "Lorem Ipsum - Two Paragraphs" and (b) is ready to upload the text by clicking "Upload".

To open the file into the cutting window, the user clicks the text in the tree with the name "Lorem Ipsum - Two Paragraphs" as shown in Figure 4.4.

This opens the text into the Visual Cutter as shown in Figure 4.5.

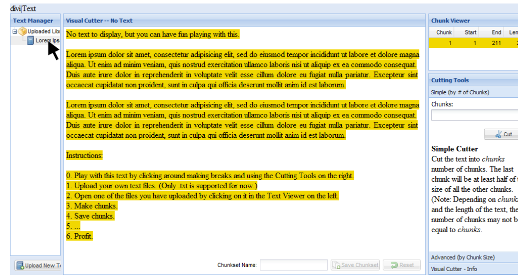


Figure 4.4: The text has been uploaded and the text is ready to be opened by clicking the text in the Text Manager. Note, that in this example the Text Manager pane (left) has been shrunk in width to the smallest size in order to maximize the Visual Cutter Panel area.

4.1.1 Using Visual Cutter

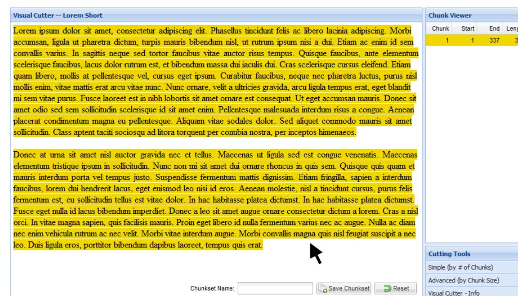


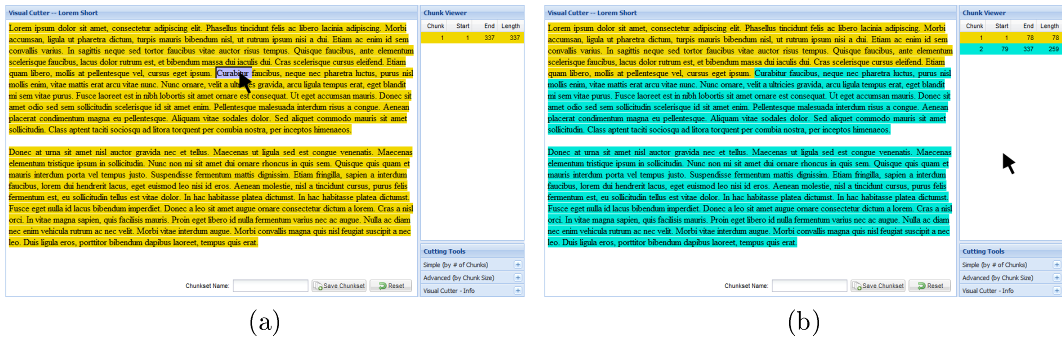
Figure 4.5: The “Lorem Ipsum - Two Paragraphs” text has been opened and is ready to be cut with Visual Cutter.

Assuming that a text has been opened, Visual Cutter is active and can be used to chunk the text into smaller, user defined pieces. This section shows how to cut the text into four chunks, making and correcting a mistake along the way.

Figure 4.5 shows the “Lorem Ipsum” text just after loading it into the Visual Cutter. The user wishes to cut the text into four chunks: (1) “Lorem” to “ipsum.” (2) “Curabitur” to “himenaeos.” (3) “Donec” to “imperdiet.” and (4) “Donec” to “erat.” This should produce four chunks, each about half a paragraph in length, each ending on a sentence boundary.

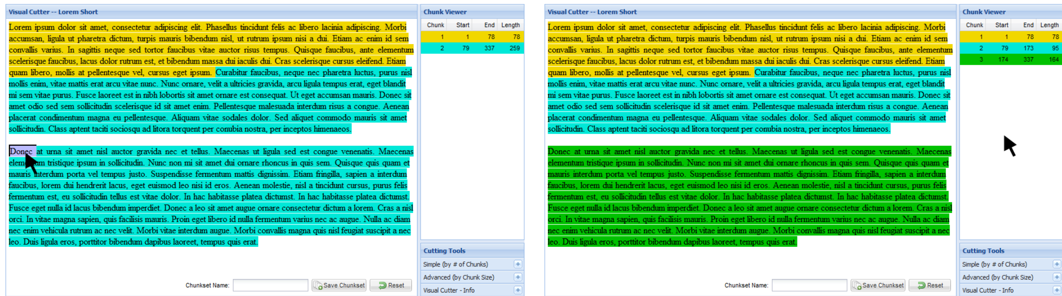
To accomplish this, the user first moves the mouse to the start of the second chunk, “Curabitur” and clicks the word (or the space after the word which is simultaneously highlighted), thus setting the first boundary. This is shown in Figure 4.6(b).

Similarly, to create the “Donec” to “imperdiet.” chunk, the user would highlight the word “Donec” and click it. The highlighting and result of the click are shown in Figure 4.7.



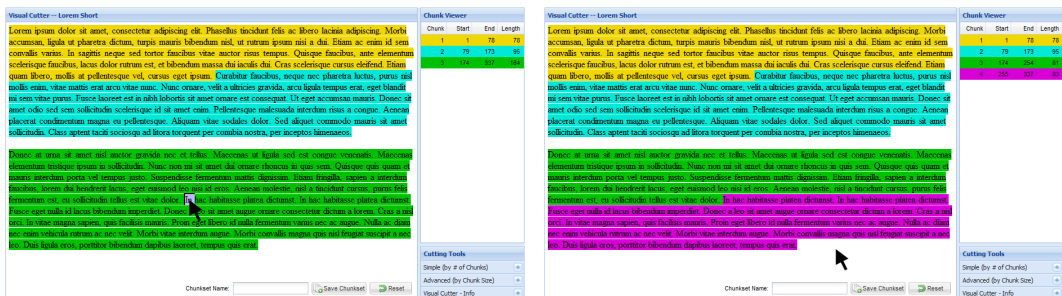
(a) (b)

Figure 4.6: The results of highlighting (a) and clicking (b) the word “Curabitur” in the Visual Cutter to split the first paragraph into two chunks.



(a) (b)

Figure 4.7: The results of highlighting (a) and clicking (b) the word “Donec” at the start of the second paragraph in the Visual Cutter.

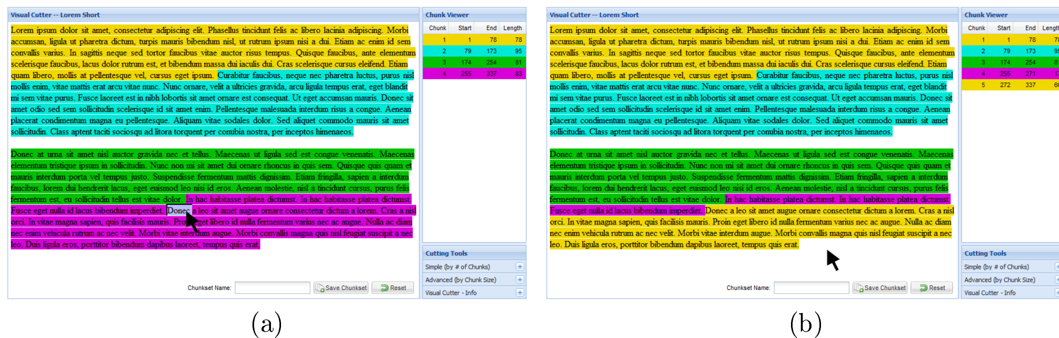


(a) (b)

Figure 4.8: The results of highlighting (a) and clicking (b) the word “In” in the second paragraph in the Visual Cutter. This is a mistake, as the user really wanted to click the word “Donec”.

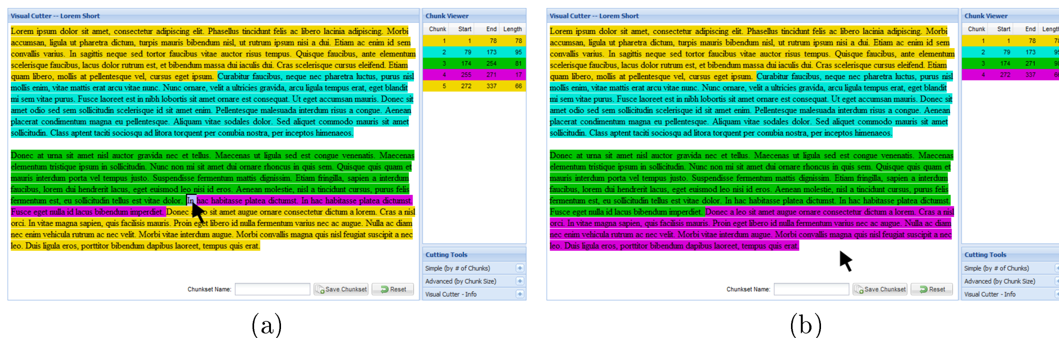
When creating the last chunk from "Donec" to "erat.", the user accidentally highlighted and clicked the word "In" as shown in Figure 4.8.

This is not the chunk that the user intended to define. To remedy the error, the user highlights and clicks the word "Donec", the true start of the fourth and final chunk (see Figure 4.9). This produces a fifth chunk. To remove the erroneous chunk beginning with the word "In" and ending with the word "imperdiet.", the user simply highlights and clicks the word "In" again, resulting in Figure 4.10.



(a) (b)

Figure 4.9: The user highlights (a) and clicks (b) the word "Donec" as was the original intention from Figure 4.8.



(a) (b)

Figure 4.10: To fix the mistake made in Figure 4.8, the user simply clicks the mistake word "In" again.

Assuming that the user is satisfied with this chunkset, the name of the chunkset is set to "Four Chunks" in the "Chunkset Name" text field. Clicking "Save Chunkset" saves the chunkset to the server.

4.1.2 Using Simple Cutter

In this version, the user wishes to create a chunkset containing four chunks of equal size. To do this, the Simple Cutter tool is employed.

Opening the "Simple (by # of Chunks)" pane on the right side of the screen, the user can specify a value of "4" chunks in the "Chunks" number field (see Figure 4.11). Clicking the "Cut" button *removes all previous chunks* and creates four almost equally sized chunks of 84 words (shown in Figure 4.12). The last chunk is one word longer than the other three chunks, but this is okay since the length of the text, 337 words, is not evenly divisible by 4 chunks.

The chunkset is saved with the name "Four Equal Chunks."

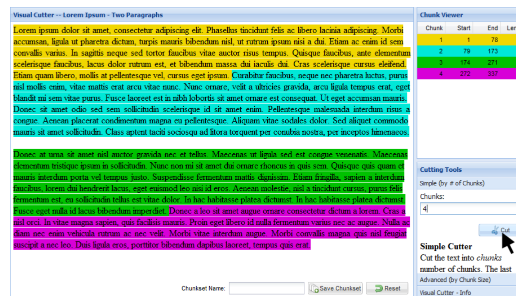


Figure 4.11: The user has entered a value of 4 in the Simple Cutter "Chunks" field and is about to click the "Cut" button.

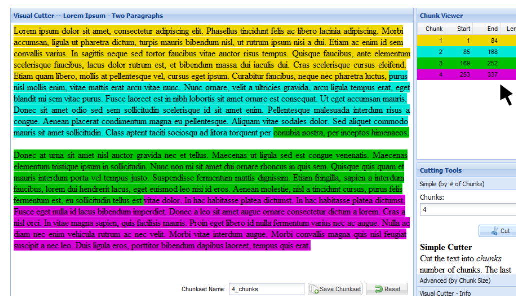


Figure 4.12: The user has clicked "Cut" and the text has been rechunked according to the 4 chunk parameter.

4.1.3 Using Advanced Cutter

Using the Advanced Cutter tool, assume the user decides to cut the text into chunks of exactly 80 words each. To do this, the user expands the "Advanced (by Chunk Size)" pane just below the Simple Cutter tool.

Figure 4.13 shows that the user has entered a value of 80 in the “Size” field and clicked the “Cut” button. This chunks the text into four chunks, three of size 80 and one of size 97. The final chunk is longer than the rest because 80 words per chunk is not evenly divisible by 337 words in the text. The extra 17 words was added to the last chunk because 17 words is not greater than half the size of all of the other chunks. This half value is the default value of the “Last Proportion” slider that the user did not change.

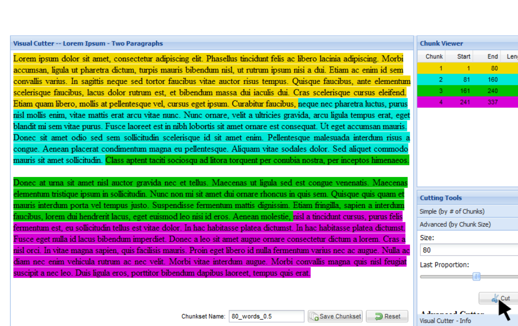


Figure 4.13: The user has chunked the text into chunks of exactly 80 words using the Advanced Cutter tool. The last chunk is 97 words.

Alternately, by dragging the slider to a value of 0.15, as shown in Figure 4.14, the user has decided that the last chunk of 17 words should belong in its own chunk, because $80 \times 0.15 = 12$ which is less than the 17 extra words. The resulting chunkset is shown in Figure 4.15.

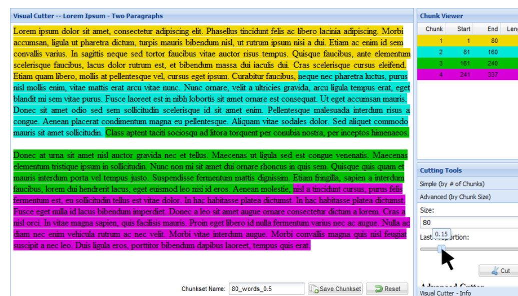


Figure 4.14: The user has decreased the “Last Proportion” slider to a value of 0.15 which requires that the last chunk must be at least 12 words to be in its own chunk.

As before, the chunkset is saved, this time as the user-defined name “80 Word Chunks + 17”. The resulting Text Manager having used all three cutting methods is shown in Figure 4.16.

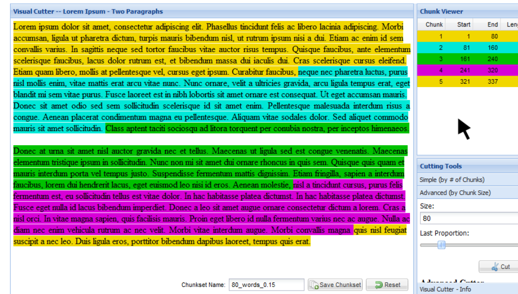


Figure 4.15: The result of an Advanced Cutter cut of chunk size 80 and a last proportion of 0.15. The final 17 words now constitute their own chunk.

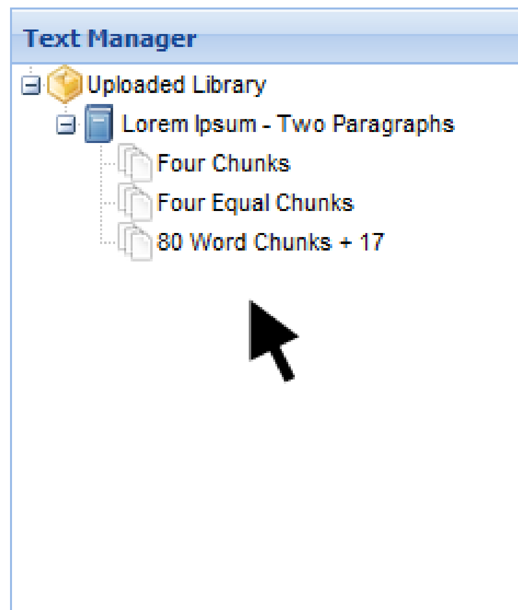


Figure 4.16: The Text Manager after walking through this toy example. There are three chunksets belonging to the "Lorem Ipsum - Two Paragraphs" text. The text with chunk breaks for any chunkset can be retrieved by clicking on the chunkset.

4.2 Real-World Examples

This section shows a real-world application of the diviText tool by cutting the poem *Beowulf* into fitts and mentions other possible uses of the tool.

4.2.1 Beowulf in Fitts

Anglo-Saxon scholars are interested in the epic poem Beowulf. Since Beowulf is naturally broken into 42 fitts, or sections, scholars tend to study the poem at the

fitt-level.

Since fitts are not evenly spaced, it is a time-consuming process to break the poem into fitts. Using old methods, the scholar would open the poem in a text editor, copy and paste each of the 42 fitts into their own file, and name each file manually. At the least, this is potentially error-prone.

Using the diviText tool, this is no problem and only takes a few minutes. To cut the text into the 42 fitts, only 41 clicks are required, plus clicks for uploading the text, saving the chunkset, and downloading the text. This is far faster than alternative methods.

4.2.2 Other Examples

Other uses for the diviText tool have been discussed.

One such use is to cut up the Federalist Papers to study the authorship of the papers. Of the 85 papers, 73 are known to be written by Alexander Hamilton, James Madison, or John Jay. This means that 12 of the papers have disputed authorship. Using the Visual Cutter, scholars can quickly cut a file containing all the papers down into 85 individual papers and get word frequencies for further analysis.

Another proposed use concerns texts of the Harlem Renaissance. Scholars of African-American literature of the middle twentieth century wish to determine how sections of text may have influenced or been influenced by sections of other texts or authors (Drout *et al.*, 2010).

Chapter 5

Conclusions

This thesis has covered a wide variety of topics regarding "text mining." diviText was developed to aid in the first steps of the text mining process.

Chapter 2 outlined the topics of data and text mining. Since text mining has specific needs, tools have been developed by people in the text mining field. Some of these tools are featured in Section 2.2 and Appendix A.

The need for a tool to aid in text segmentation led to the development of diviText. Chapter 3 detailed the functionality of diviText; Chapter 4 showed how users interact with diviText.

5.1 Future Work

There is still much that can be done to improve the functionality, usability, and stability of diviText in versions 2 and beyond.

5.1.1 Functionality

Many features were discussed when drafting diviText on paper but time did not allow everything to be incorporated into the final product. The most glaring omission is that diviText can only segment texts. The next steps would include adding a second stage in the pipeline that allows a user to proceed to perform clustering and/or classification analyses on their segmented texts through the diviText interface. This would involve implementing a way to group texts, chunksets, and chunks into logical sets. A single word count/relative frequency table (a merged set of texts) would be calculated and added to the ZIP file the user downloads.

Another feature that didn't make it into the final product was the ability to arbitrarily segment texts, for example, to make overlapping chunks. The original idea for Visual Cutter was to allow users to place arbitrary start and end points for

chunks of text. This would allow overlapping segments and the use of Simple and Advanced Cutters on subsets of the text.

5.1.2 Usability

While Visual Cutter is an advancement over previous segmentation tools that operated on the command-line, issues were discovered during use-case testing. The biggest issue was how users wanted to deselect chunks. Most users clicked the word prior to the word they should have clicked. At present, this is not so much a bug, but a feature; however, future consideration will be given to this issue depending on user feedback.

Users also wanted to upload texts that were not raw text, e.g. Word documents, XML, and texts with non-English character sets (e.g. Russian poems). Future versions could handle XML, at the very least. "Hooks" have been placed into diviText to handle such extensions. In addition, the capability to upload multiple files at once could also be handled.

Another feature in the list of "it would be nice to have..." is the ability for the user to control the coloring of text segments. The current colors were chosen because they were neutral but distinguishable. The lack of contrast in the colors, differences between monitors, and user-specific variables like colorblindness may be an issue for some users.

5.1.3 Stability

As with most version 1 tools, issues were encountered when users did unexpected things, for example, uploading texts that were not encoded as raw ASCII text. Texts that include symbols not within the range of simple Roman characters, Arabic numbers, and simple punctuation produce undefined behavior (usually uploading stops when the first non-ASCII character is found and the remaining text is truncated).

A few menus in the diviText interface do not work quite right. After right-clicking a text in the Text Manager, menu options do not show on subsequent clicks and the page must be refreshed.

Additional items to consider for future work include the fact that Visual Cutter does not handle large texts well. Since every word is a clickable object, each word has a non-negligible memory footprint, perhaps past the memory allocation limit in FireFox of 1GB of RAM. Currently diviText limits the total number of words to $75,000$ when using Visual Cutter, although longer texts can still be cut with Simple and Advanced Cutters. This is significant in the context of potentially hundred-thousand plus word texts. This could be fixed with a revamp of Visual Cutter. In version 2, Visual Cutter might only create an object when words and/or spaces are clicked and not for every

word. Also, when handling large texts, texts may not be rendered properly if words and corresponding following spaces become unaligned. The system might not alter the user's text until segmentation, and then only alter a copy of the original text.

While diviText is only in its infancy, it is a welcome replacement for old text segmentation tools. Positive feedback from users have shaped diviText into its current form and set our goals for future functionality.

Appendix A

Clustering Using Other Tools

To demonstrate clustering using the built-in functionality provided by other text analysis tools, a simple data set was created to give fairly expectable results. This dataset, seen in Table A.1, contains seven 'texts' labeled 'T1' through 'T7' with which contain the five words 'and', 'the', 'some', 'people' and 'that.' Each of the texts contains some combination of ten of these words each.

text	and	the	some	people	that
T1	10	0	0	0	0
T2	5	5	0	0	0
T3	0	0	10	0	0
T4	2	2	2	2	2
T5	0	0	0	5	5
T6	0	0	0	1	9
T7	0	0	0	0	10

Table A.1: The sample dataset containing seven texts (T1-T7), of five unique and ten total words each.

The first header row contains the list of unique words found in the collection of seven texts. Each subsequent row corresponds to a single text where the text name is the very first item in the row (e.g. 'T1' corresponds to 'Text 1'). Subsequent columns indicate counts of a word in the text row where it is found.

It should be noted that in each example, clusters are based on raw word frequencies. This works here because each text contains the same number of total words. In an example where this would not be true, relative frequencies (word counts divided by the total number of words in a text) would be used in place of raw frequencies to account for variation in the length of the texts.

A.1 Meandre

One of Meandre's pre-installed workflows is a clustering flow, seen in Figure A.1 (SEASR, 2010). Only one change has to be made for the flow to run correctly; the `file_url` parameter of the `Input URL or Path` component must be changed to point to the dataset either on the web or local machine. The input for this flow is a delimited file, like the one above, but with one small change, seen in Table A.2. Meandre requires an additional row (see row 2 in Table A.2) that designates a data type for each column.

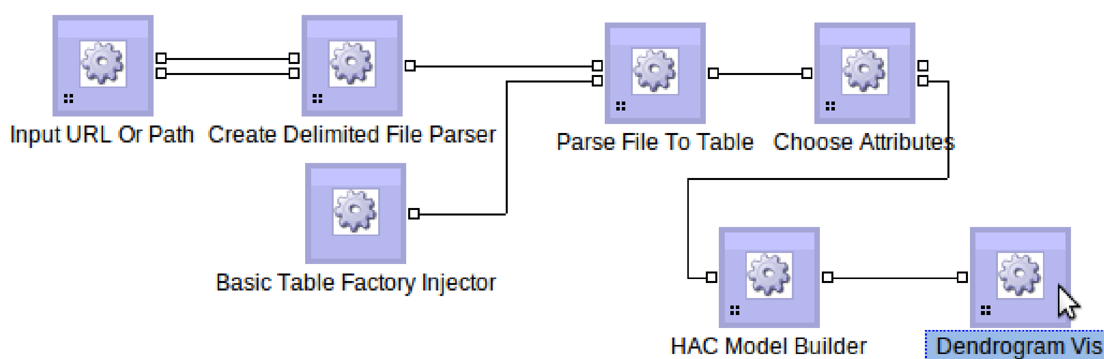
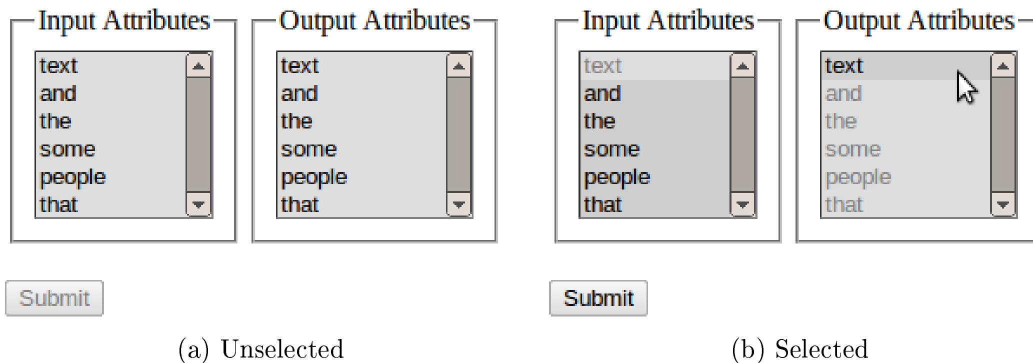


Figure A.1: The cluster flow as seen in Meandre.

text	and	the	some	people	that
String	int	int	int	int	int
T1	10	0	0	0	0
T2	5	5	0	0	0
T3	0	0	10	0	0
T4	2	2	2	2	2
T5	0	0	0	5	5
T6	0	0	0	1	9
T7	0	0	0	0	10

Table A.2: The sample dataset with changes required for Meandre.

Upon running the flow, Meandre prompts for input and output attributes. The input attributes correspond to the variables used in the cluster analysis; in this example, these are the word counts. The output attribute is simply what Meandre will use to label the pieces of the clusters, i.e. the text names. The initial unselected and subsequent properly selected attributes can be seen in Figures A.2a and A.2b.



(a) Unselected (b) Selected

Figure A.2: Input and output attribute selection. Inputs attributes are word types, and output attributes are the text names used to label items in each cluster.

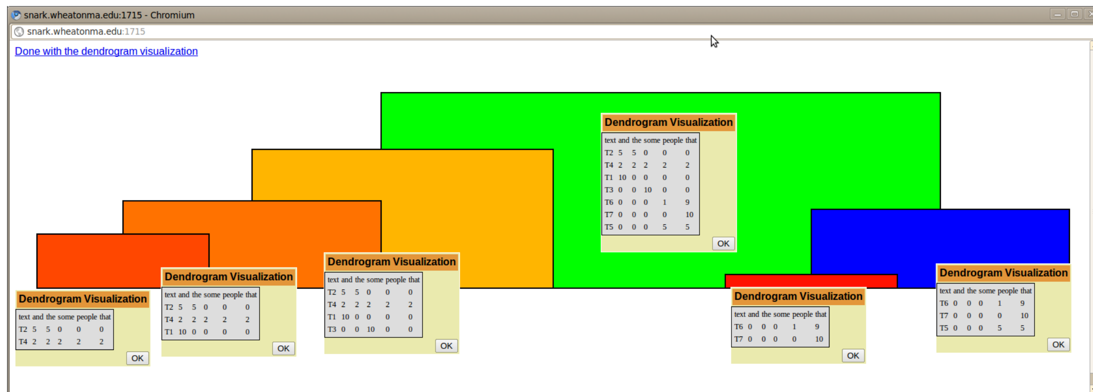


Figure A.3: Dendrogram created by Meandre when using the cluster workflow. The stretching and window placement were manipulated by hand to appear as shown. The dendrogram indicates through the smallest clade, located on the right, that T6 and T7 cluster most closely together. This is explained by the single word difference in the texts (T6 contains one instance of the word 'people' not seen in T7).

When the clustering is completed, Meandre creates the dendrogram seen in Figure A.3. Meaning is only applied to the output when a cluster is selected. When a cluster is selected a window pops up containing the texts placed in the cluster and the respective word counts. These windows must be manually arranged to easily identify clusters. This has been done in Figure A.3. These lists would easily get unruly with larger datasets containing more texts or, worse still, when more words are involved.

A.2 Python: NLTK and SciPy

Using Python, and the right tools from the right packages, a dendrogram can be obtained (Bird *et al.*, 2009). The science package, SciPy, contains the two required hierarchical clustering and dendrogram-building functions.

Listing A.1 is the script to run a cluster on the sample dataset. The data was entered by hand for simplicity's sake, but for a more complex example, texts and word counts would be read from files using readers like those found in NLTK.

Listing A.1: Script to build a simple dendrogram for the sample data.

```
1 import pylab
2 from scipy.cluster import hierarchy
3 from numpy import array
4
5 vectors = [ array(f) for f in [ [10,0,0,0,0], [5,5,0,0,0], [0,0,10,0,0], [2,2,2,2,2],
      [0,0,0,5,5], [0,0,0,1,9], [0,0,0,0,10] ] ]
6 texts = [ 'T1', 'T2', 'T3', 'T4', 'T5', 'T6', 'T7' ]
7 Z = hierarchy.linkage( vectors, method='centroid', metric='euclidean' )
8 d = hierarchy.dendrogram( Z, labels=texts )
9 pylab.show()
```

First, this script imports a few required packages. The `pylab` package is used to render the dendrogram. From the `scipy.cluster` package, `hierarchy` is imported. By importing these, needlessly long function calls can be shortened from `scipy.cluster.hierarchy.linkage` to `hierarchy.linkage`. From `numpy` (the mathematical package), the `array` function is imported because `arrays` are data types expected for text vectors.

The `hierarchy` class contains two important methods, `linkage` and `dendrogram`. The `linkage` method clusters a set of vectors, one vector per text. Here, the vector is the `array` of word frequencies. In this example, the `centroid` method is used for calculating distance in euclidean space. The other method, `dendrogram`, builds the dendrogram, Figure A.4, using the output from `linkage`, stored in linkage matrix `Z`, and the list of text names as `labels`.

A.3 R

R, the statistical and graphics programming language, is well equipped to handle this simple clustering example. Without any third-party packages, R is capable of creating publication quality dendrograms based on results of a hierarchical cluster. The script is seen in Listing A.2.

First, text names are put into an R list (line 1). The following seven lines create separate lists for the counts of words per text. The method `cbind` creates a matrix

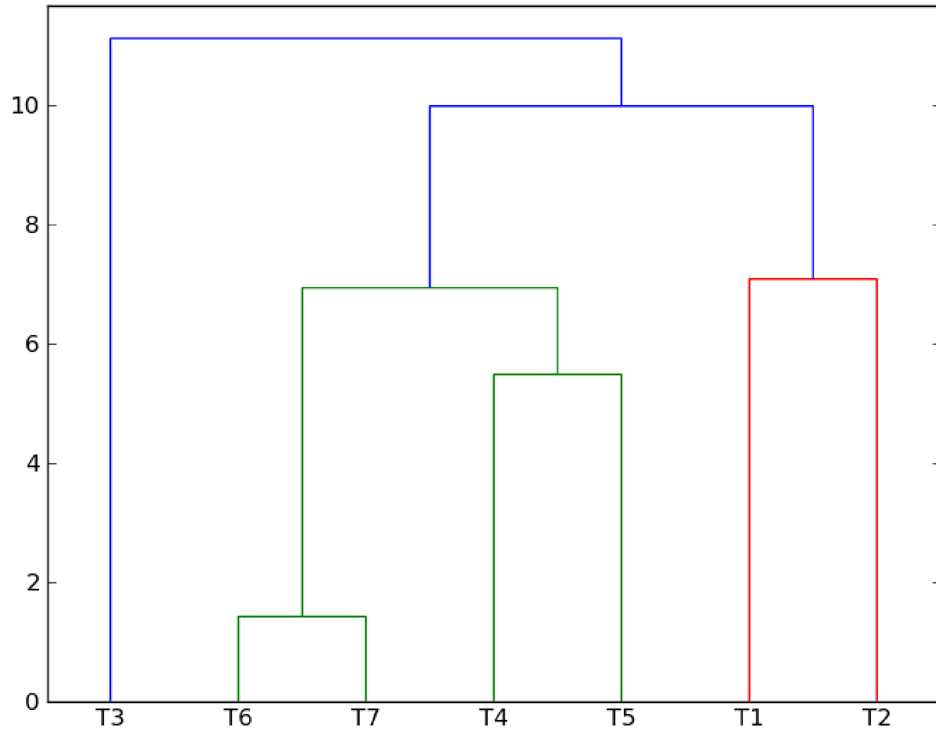


Figure A.4: SciPy's dendrogram output for the sample data. The left axis measures the amount of similarity. The topology in this dendrogram is quite different than the topology of Meandre's dendrogram due to the use of the "centroid" linkage method. SciPy directly links T1 and T2 while Meandre links T2 and T4, and then T1 to that clade.

of word counts, but this puts texts into columns, not rows. To correct for this, the `t`, matrix transposition, method switches rows and columns before calculating the distance matrix required to cluster (lines 13 and 14). This distance matrix, `vdist`, is passed to `hclust` to perform the hierarchical cluster with default parameters `method=complete`.

Following line 14, the clustering is finished, and all that remains is rendering the dendrogram. The `png` method builds a Portable Network Graphics (.png) file with the supplied name. The method `plot` uses the results from the cluster to render the dendrogram. Texts names in `texts` are supplied as the `labels` for each leaf, and `Cluster Results` is the dendrogram title. The final line, `dev.off()`, shuts off the writing ability and finalizes the PNG file. The results are shown in Figure A.5.

Listing A.2: R script to perform hierarchical cluster on the sample dataset.

```
1 texts <- c( 'T1', 'T2', 'T3', 'T4', 'T5', 'T6', 'T7' )
2
3 t1 <- c(10,0,0,0,0)
4 t2 <- c(5,5,0,0,0)
5 t3 <- c(0,0,10,0,0)
6 t4 <- c(2,2,2,2,2)
7 t5 <- c(0,0,0,5,5)
8 t6 <- c(0,0,0,1,9)
9 t7 <- c(0,0,0,0,10)
10
11 vectors <- cbind(t1,t2,t3,t4,t5,t6,t7)
12
13 vdist <- dist( t(vectors) )
14 result <- hclust(vdist)
15
16 png("cluster_dendro_from_r.png")
17 plot(result, labels=texts, main="Cluster Results")
18 dev.off()
```

A.3.1 R using RPy

By using the RPy package in Python, R commands can be executed through the Python environment and data can be passed from Python to R and vice versa.

The R code, with regards to clustering and rendering, is almost identical to that used in the R example in Section A.3, with only a few minor changes to account for the use with Python. This script is shown in Listing A.3.

Listing A.3: Script in Python implementing a cluster of the sample dataset using RPy to create an R environment.

```
1 from rpy import r
2 from numpy import array
3
4 vectors = array( [ [10,0,0,0,0], [5,5,0,0,0], [0,0,10,0,0], [2,2,2,2,2], [0,0,0,5,5],
      [0,0,0,1,9], [0,0,0,0,10] ] )
5 texts = [ 'T1', 'T2', 'T3', 'T4', 'T5', 'T6', 'T7' ]
6
7 r.assign( "vectors", vectors )
8 r( 'vdist <- dist(vectors)' )
9 r( 'result <- hclust(vdist)' )
10
11 r( 'png("cluster_dendro_from_rpy.png")' )
12 r.assign( "texts", texts )
13 r( 'plot(result, labels=texts, main="Cluster Results")' )
14 r( 'dev.off()' )
```

First, `r` is imported from the `rpy` package. This is the main method that runs the R environment in Python. The `array` method is imported from NumPy to build a matrix-like data structure that RPy can form into an R matrix.

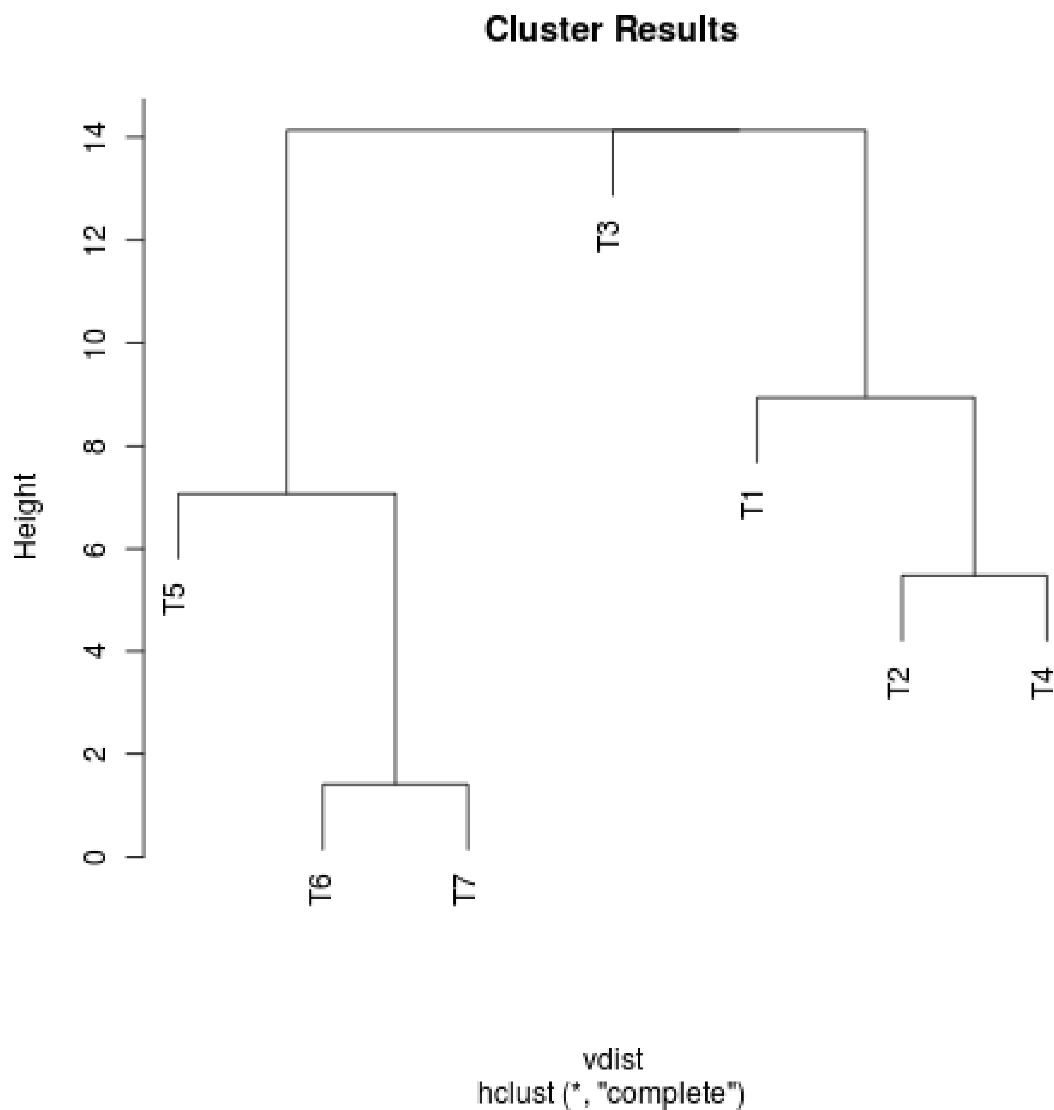


Figure A.5: Dendrogram resulting from R when clustering the sample dataset. Here, the topology slightly differs from that of SciPy but is similar to Meandre's. These differences are caused by different default clustering parameters. R and Meandre default to "complete" linkage.

The word counts are hand entered into a list of Python lists which is then put into an `array` so RPy can build an R matrix. The text names are then put into a Python list (line 5).

The `r.assign` method passes data from the Python matrix-like variable `vectors`

into the R matrix variable `vectors`. Line 8 calculates the distance matrix within the R environment, this time without the transposition function because the texts were built in rows in Python. This is then clustered using the same process as before.

Finally the PNG file is built, but before the texts can be used in the rendering, the Python list `texts` is assigned to the R vector variable `texts` (line 12).

The resulting dendrogram is exactly the same as Figure A.5. This method has proven best due to the ease of inputting the sample data and high quality dendrogram output.

Appendix B

Licensing and Source Code

This Appendix addresses issues not covered elsewhere in the main thesis regarding the licensing of this code and where the source code can be found.

diviText is available for free use, redistribution, and/or modification online at `http://cs.wheatoncollege.edu/~amos/divitext`.

B.1 Licensing

ExtJS is not a free JavaScript framework. It can, however, be used freely if all the source code that uses ExtJS is made free and open source under the GNU General Public License, GPLv3. The terms of the license can be found at `http://www.gnu.org/licenses/`.

In accordance with the GPL, the following text is added to the source of each file.

```
diviText is a graphical text segmentation tool for use in text mining.
    Copyright (C) 2011 Amos Jones and Lexomics Research Group

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Icons in both diviText and the images of diviText in this document are part of the FamFamFam Silk Icon Set licensed under the Creative Commons 2.5 Attribution License. This license provides provisions to both use and alter these icons with attribution to the original author. These icons can be found at `http://www.famfamfam.com/lab/icons/silk/`.

B.2 Source Code

The source code for all of diviText is made available in a number of ways.

The first is by accessing the "Download Source" link at the bottom of the diviText webpage. This link provides a ZIP of a recent snapshot of the code that is running live on the server. It will not always be updated after minor changes, but any feature addition or large bug removal should warrant an update.

The code can also be found by accessing the Google Code repository created for diviText. This can be found at `http://divitext.googlecode.com`. Google Code provides a free web front end that allows users to browse and obtain the code repository including old revisions of committed code. Using Mercurial, a source control program, users can clone the current copy of the code to their local machine.

This document was turned in to the Wheaton College Registrar with a CD containing the most recent (as of May 12, 2011) source code.

Bibliography

The British National Corpus, version 3 (BNC XML Edition). Distributed by Oxford University Computing Services on behalf of the BNC Consortium. `http://www.natcorp.ox.ac.uk/`, 2007.

Ács, Bernie; Llorà, Xavier; Auvil, Loretta; Capitanu, Boris; Tcheng, David; Haberman, Mike; Dong, Limin; Wentling, Tim; and Welge, Michael. A General Approach to Data-Intensive Computing using the Meandre Component-Based Framework. *Proceeding Wands '10 Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, 2010.

Archer, Dawn (ed). *What's in a Word-list?* Ashgate: Bodmin, Cornwall, 2009.

Berry, Micheal W. (ed). *Survey of Text Mining: Clustering, Classification and Retrieval.* Springer: New York, NY, 2004.

Bird, Stephen; Loper, Edward; and Klien, Ewan. *Natural Language Processing with Python.* O'Rielly Media Inc., 2009.

Carroll, Lewis. *Alice's Adventures in Wonderland.* Macmillan, 1865.

Cunningham, Hamish; Maynard, Diana; Bontcheva, Kalina; Tablan, Valentin; Aswani, Niraj; Roberts, Ian; Gorrell, Genevieve; Funk, Adam; Roberts, Angus; Damljanovic, Danica; Heitz, Thomas; Greenwood, Mark; Saggion, Horacio; Petrak, Johann; Li, Yaoyong; and Peters, Wim. *Developing Language Processing Components with GATE Version 6 (a User Guide)*, 2010.

Davies, Mark. CORPORA: 45-400 million words each: free online access. `http://corpus.byu.edu/`, 2008a.

Davies, Mark. The Corpus of Contemporary American English (COCA): 410+ million words, 1990-present. `http://www.americancorpus.org`, 2008b.

Davies, Mark. Word Frequency in Context: Alternative Architectures for Examining Related Words, Register Variation and Historical Change. In Archer, Dawn (ed), *What's in a Word-list?* Ashgate: Bodmin, Cornwall, 2009.

Davies, Mark. The Corpus of Historical American English (COHA): 400+ million words, 1810-2009. http://corpus.byu.edu/coha, 2010.

Drout, Michael D. C.; Kahn, Micheal J.; and LeBlanc, Mark D. Lexomic Tools and Methods for Textual Analysis: Providing Deep Access to Digitized Texts. National Endowment for the Humanities – NEH Grant #PR-50112011, 2010.

Drout, Michael D. C.; Kahn, Micheal J.; LeBlanc, Mark D.; and Nelson, Christina. Of Dendrogrammatology: Lexomic Methods for Analyzing the Relationships Among Old English Poems. *Journal of English and Germanic Philology*, 2011.

Friedman, Nir and Kohavi, Ronny. Bayesian Classification. In Żytkow, Jan M. and Klösgen, Willi (eds), *Handbook of Data Mining and Knowledge Discovery*, pages 282–288. Oxford University Press, Inc.: New York, New York, 2002.

Hart, Michael. Project Gutenberg. http://www.guenberg.org, 2011.

Healey, Antonette diPaolo; Haines, Dorothy; Holland, Joan; McDougall, David; McDougall, Ian; and Xiang, Xin. The Dictionary of Old English Corpus in Electronic Form. [CD-ROM], 2004.

Healey, Antonette diPaolo; Haines, Dorothy; Holland, Joan; McDougall, David; McDougall, Ian; and Xiang, Xin. Dictionary of Old English – Tools. http://www.doe.utoronto.ca/tools/tools.html, 2009.

Jurafsky, Daniel and Martin, James H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Education, Inc.: Upper Saddle River, New Jersey, 2 edition, 2009.

Kantardzic, Mehmed M. and Zurada, Jozef (eds). *Next Generation of Data-Mining Applications*. John Wiley & Sons, Inc.: Hoboken, New Jersey, 2005.

Kleinman, Scott. The Impact of Lemmatization of Lexomic Hierarchical Clustering of Old English Texts. Presented at Forty-sixth International Congress on Medieval Studies May 12-15, 2011.

Kohavi, Ronny and Quinlan, J. Ross. Decision-Tree Discovery. In Żytkow, Jan M. and Klösgen, Willi (eds), *Handbook of Data Mining and Knowledge Discovery*, pages 267–276. Oxford University Press, Inc.: New York, New York, 2002.

Kononenko, Igor and Kukar, Matjaž. *Machine Learning and Data Mining: Introduction to Principles and Algorithms*. Horwood Publishing: Chichester, UK, 2007.

Kraus, Lauren E. Using Cluster Analysis to Identify Relationships Between Old English Poems. Undergraduate Thesis, Wheaton College (MA), May 2010.

LeBlanc, Mark D.; Kahn, Michael J.; Drout, Michael D. C.; Brousseau, Matthew; Jones, Amos; Nelson, Christina; and Waltz, Brandon. Wheaton College Lexomics. `http://lexomics.wheatoncollege.edu`, 2010.

Leping, Vambola; Lepp, Marina; Niitsoo, Margus; Tõnisson, Eno; Vene, Varmo; and Villems, Anne. Python Prevails. *ACM International Conference Proceeding Series*, 443, 2009.

McCallum, Andrew Kachites. MALLET: A Machine Learning for Language Toolkit. `http://mallet.cs.umass.edu`, 2002.

Michel, Jean-Baptiste; Yuan Kui Shen, Adrian Veres, Aviva Presser Aiden; Matthew K. Gray, The Google Books Team Joseph P. Pickett, William Brockman; Dale Hoiberg, Peter Norvig Jon Orwant Steven Pinker, Dan Clancy; Nowak, Martin A.; and Aiden, Erez Lieberman. Quantitative Analysis of Culture Using Millions of Digitized Books. (Published online ahead of print in *Science*: 12/16/2010), 2010.

R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

Scott, Mike. WordSmith Tools version 5. `http://www.lexically.net/downloads/version5/HTML/index.html`, 2010.

SEASR. SEASR, 2010. `http://seasr.org/documentation/`.

TEI Consortium (ed). *TEI P5: Guidelines for Electronic Text Encoding and Interchange.* November 5, 2010. `http://www.tei-c.org/P5`.

Weiss, Sholom M.; Indurkhya, Nitin; Zhang, Tong; and Damerau, Fred J. *Text Mining: Predictive Methods for Analyzing Unstructured Information.* Springer: New York, NY, 2005.

Żytkow, Jan M. Decision Trees. In Żytkow, Jan M. and Klösgen, Willi (eds), *Handbook of Data Mining and Knowledge Discovery*, pages 54–56. Oxford University Press, Inc.: New York, New York, 2002.