1-1-2007

Generating and Presenting String Frequency Measurements of Project Gutenberg Texts

Ronald P. Reck

GENERATING AND PRESENTING STRING FREQUENCY MEASUREMENTS OF PROJECT GUTENBERG TEXTS

by

Ronald P. Reck

Thesis

Submitted to the Department of English Language and Literature

Eastern Michigan University

in partial fulfillment of the requirements

for the degree of

MASTER OF ARTS

in

English with a concentration in Linguistics

Thesis Committee:

Anthony Aristar

Helen Aristar-Dry

March 14, 2007

Ypsilanti, Michigan

Dedication

This paper is dedicated to the memory of Steven G. Lapointe. I was fortunate enough to meet Steve while attending Wayne State University and I enjoyed his enthusiasm and intellect. Steve's sharp mind and sharper wit invited me into the wonderful world of Linguistics, and for that I thank him. I hope Steve knows what a positive impact his teaching has had on my life.

Acknowledgements

A number of people supported this effort. I would like to thank my readers, Anthony Aristar and Helen Aristar-Dry; my friends John D. Porter, Ljuba Veselinova, and Tom Beckman; my wife, Olga Lorincz-Reck; and my mother, Dr. Ruth A. Reck. Last but not least, I appreciate the comments and answers I received from the semantic web enthusiasts that occupy the IRC #swig channel on irc.freenode.net.

ABSTRACT

The electronic age has increased the range of human capabilities to such an extent that the expectations about appropriate empirical linguistic analysis are changing. A hundred years ago, linguistics was largely an empirical manual process that produced information intended for humans. Today, the world is different as inexpensive computing power and the prevalence of information in electronic format encourages that, whenever possible, information be processed by automated and scalable means and the results be usable and understandable by computers. Creating sustainable and usable observations is best achieved through a standards-based approach that meets long term persistence and usability goals. This thesis presents a scalable architecture for creating linguistic observations in the form of string frequencies measurements and instantiates those measurements in a machine-readable standards-based format called Resource Descriptive Framework (RDF).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Introduction

The Problem

This thesis is concerned with generating and presenting string frequency measurements of text files. Unlike other possible approaches, this analysis employs a parallel architecture that can take maximum advantage of all the available computing resources. More importantly, the results from the analysis are rendered in a format that allows machines to interpret the findings so that they can have wider application both inside the field of linguistics and beyond. The expectation linguists have had in the past is that humans are the primary consumers of their observations. Now, it is reasonable to expand that belief. New information should be accessible and interpretable by both humans and machines alike. Given this new perspective, a reasonable question would be "what format should machine readable information take?" This thesis provides an answer to that question, not only in theory but in practice. An archive of more than fourteen thousand books, consisting of almost one billion words, is analyzed to create approximately five hundred million computer-readable assertions. These assertions describe text files and the words they contain. While the assertions are themselves intrinsically useful, they also serve as a prototype so that others may comment, criticize, or adopt similar strategies for their own observations.

Background

String frequency analysis has been used for decades as part of descriptive linguistics. While the initial efforts for determining string frequencies could not use computers, any current efforts would surely be shortsighted not to. The prevalence of computers and electronic data have encouraged modern string frequency analysis efforts to increase in scope considerably. The major contributions to string frequency analysis in linguistics are reviewed

in this paper's next section, which provides a historical perspective. This thesis builds upon a foundation explained in two complementary efforts by the same author, Metadata Cards for Describing Project Gutenberg Texts (Reck 2006) and Structuring Powerful Uniform Resource Identifiers for Lexical Entries (Reck to appear).

These earlier efforts were broader in scope as one dealt with general textual metadata and the other explained the design for any computer based lexical entries. The current effort employs the same concepts in that it presents a very specific type of textual metadata and presents it in terms of how lexical entries should be represented for maximum effectiveness and clarity.

Initial efforts have explained the problems and limitations in deciphering and determining the metadata for the text files found in the archive of free available texts called Project Gutenberg. Metadata Cards for Describing Project Gutenberg Texts (Reck 2006) describe the creation of independent files called "metacards" that depict more than a dozen attributes for most of the texts in the Project Gutenberg archive. In the current effort, the same set of metacards are extended to more thoroughly describe the contents of the text files. The previous metacards had an average size of 31 lines, and size variations occurred because any undetermined attributes were omitted. In contrast, the current metacards can be tens of thousands of lines long. Each metacard's length varies as it ultimately depends on the number of unique strings in the text file. The dramatic increase in file size occurs because each unique string in the text adds four lines to the metacard's length. While this appears to be a drawback, it allows a more exacting specification of the file's contents, and the metacards are always smaller than the texts they describe. Longer text files do generally mean longer

metacards, whereas previously no such correlation existed between the metacard's length and the size or complexity of the text file.

A second paper called Structuring Powerful Uniform Resource Identifiers for Lexical Entries (Reck to appear) describes the organization behind computer-readable lexical entries. There are many reasons why people would want to gather and organize lexical entries. Possible examples range from simple lists, glossaries, and dictionaries to more expressive formats like thesauri, taxonomies, and ontologies. These increasing levels of expressivity are on a continuum but ultimately are trying to relate units of meaning at the level of the word as a discrete unit. Different communities have different orientations, different needs, and different expectations. The only way to consistently approach the task of representation is to recognize and include the orientation of the consumer in the data model itself. If the data model can account for data producers and consumers alike, it is more likely to remain useful throughout the software lifecycle. The term "software life cycle" can be defined as the period that begins when a piece of software is a mere concept and ends when it is no longer used. Typically there are several overlapping steps in the lifecycle such as requirements, design, implementation, test, deployment, maintenance, and inevitable retirement. All software is subject to the software lifecycle.

Justification

The reason this effort is both important and useful is that it reflects the impact of computers on the humanities and more specifically the field of linguistics. Writing serves as an important mechanism by which culture is transmitted over time. The prevalence and pervasiveness of computers will continue to have an impact on the capabilities of linguistics as a science. Traditionally, writing has been preserved through the printed medium as books.

As a transition occurs that moves writing from being preserved on paper to being encoded in electronic format, the science of linguistics needs to adapt so that it embraces the increase of available information while ensuring that observations will remain useful for years to come. Initially the power of computers was only a fraction of what it is today. Moore's law predicts that the number of transistors that can be crammed into a single silicon chip while still keeping to the lowest cost will double annually. This means that the increase in computer capability is likely to continue to increase consistently for the foreseeable future. In preparation, linguistic analysis itself should be posed to leverage additional computer capabilities as they become available. The parallel processing model that is presented in this paper does just that by allowing multiple analysis instances to run concurrently. The model, while not particularly novel, can serve as a solid example for how language analysis should be designed to occur. If there is an increase in available input data but the approach doesn't reflect the ability to increase in throughput capability, there may come a time in which the amount of data to be analyzed will increase with a greater speed than the ability to analyze it. As previously stated, writing serves as the means by which culture preserves information. For hundreds of years that writing has been put down on paper. As the technology of paper quality has improved, any writing has been preserved for longer and longer periods without needing to be rewritten on new paper. As a paradigm shift occurs, paper will be used less because information is encoded in electronic format. Linguistics need to be cognizant of the best way to preserve electronic information. One way to ensure that electronic information is preserved over time is to encode it in a fashion where what is being said is interpretable by machines. If machines can read information, then they can also aid in any migration of information between the formats and modalities that will occur in the future.

In essence, the challenge is one of information interoperability. When writing is as clear as possible, it can have the greatest possible impact both for its originally intended use and for uses not yet conceived.

Significance

No answer taken in isolation can be considered the “right” answer. Answers, like tools, are correct only in terms of the question or problem to be solved. This thesis intends to propose an approach for parallel processing of textual information and the encoding of results. A clear proposal invites others to judge, criticize, and improve upon it. The objective of this effort is to make a clearly documented approach so that others may adopt or improve upon it. Collocations of letters can be referred to as strings, words, or tokens, depending on the context. No matter what they are called, they serve a foundational role in linguistics analysis. By providing a robust and scalable approach for the presentation of string frequency measurements, this effort creates information that can be used elsewhere without any complicated explanations. Moreover, the results of this analysis are available for other researchers to use. Later efforts can create and test hypotheses that correlate string frequencies with author attributes like gender, nationality, or native language without having to generate string frequency measurements.

String Frequency Analysis and Corpus Linguistics

Any review of string frequency analysis in the field of linguistics must point out that frequency measurements are firmly grounded as part of the empirical observations of language. As such, it is also important to recognize that there is a dichotomy between empiricism and rationalism. Empiricism is an orientation and claim that knowledge is part of observation and experience. At one extreme, empiricism postulates that the only source of knowledge is tied to experience. In contrast, rationalism depicts that truth is based on reason and reflection. It is likely that both orientations are relevant to an unbiased representation of human thought, existence and language.

The dichotomy of empiricism and rationalism is seen in linguistics as in many other fields. The Swiss linguist Ferdinand de Saussure's most notable contribution to the field of linguistics entitled "Course in General Linguistics" is recognized for its notions of the linguistic sign, the signifier, the signified, and the referent. These notions tease out the complex interplay between observation and reflection in the field of Linguistics. Saussure's ideas can be heard echoing decades later in Noam Chomsky's definitions of Competence and Performance. Chomsky's ideas and writings are often cited for shifting viewpoints from the Empiricism that dominated Linguistic thought of the 20th century toward an increased emphasis on Rationalism. The heart of the Chomsky's argument is that any empirical observation is necessarily limited in scope and therefore cannot fully account for language production.

Irrespective of which orientation serves the field of linguistics better, be it rationalism or empiricism, competence or performance, observations will always be an irreplaceable and useful component of language study. As such, observations about language use in the form of

string frequency measurements provide unbiased and foundational information about how language is used.

String frequency measurements are part of corpus linguistics. Corpus linguistics is defined as “the study of language based on examples of ‘real life’ language use” (McEnery & Wilson, 1996). It is not a branch of linguistics per se, rather it is a methodology. Leech (1992) argues that “the corpus is a more powerful methodology from the point of view of the scientific method, as it is open to objective verification of results.”

Any frequency analysis is tied to the corpus or data set that is being studied. This means that string frequency analysis is deeply rooted in what is termed Descriptive Linguistics. Descriptive Linguistics describes how language is used by a speech community. Linguistic description is often contrasted by linguistic prescription, which directs how language should be used properly. The following discussion looks at corpus linguistic efforts that directly involved string frequency analysis, followed by a more general review of important corpus linguistics efforts.

Father Roberto Bursa

String analysis itself has been going on for decades. According to Hockey (2000), electronic text analysis began as early as 1949 with the work of Father Roberto Bursa's analysis entitled “The Index Thomisticus.” Father Bursa's analysis of the works of Thomas Aquinas and related authors presented an alphabetical listing of 11 million words, which was quite an accomplishment to say the least. Bursa recognized that defining a word as a sequence of letters separated by punctuation marks or spaces had severe implications for an inflected language like Latin. This same problem occurs in English. Bursa favored an approach where words were organized by their dictionary headword. This combines a

straightforward data driven model to a model that attempts to unite strings and definitions. In English this approach captures the similarity between the various forms of the word “to go”; go, going, gone, and went. This would mean that alphabetization would have some influence on the organization but would not make the sole determination since these various forms of “to go” would properly be included under the same headword. Bursa's lemmatization required a monumental manual effort, which partly explains why the first volume of Index Thomisticus did not occur until 1973, a full 24 years after the effort began.

The Brown Corpus

Modern day computer-based Corpus Linguistics really started in the early 1960s with the creation of the Brown Corpus at Brown University in Providence, Rhode Island. The primary researchers Henry Kucera and W. Nelson Francis provided basic statistics about the Brown Corpus in their paper Computational Analysis of Present-Day American English (1967). Francis notes there were corpuses in British Columbia before that date but they were not based upon computers (W.N. Francis 1992). The Brown Corpus, which consisted of a little over a million words, was considered large at the time but is modest by today’s standards.

The British National Corpus

The British National Corpus (BNC) is a 100-million-word collection of samples of written and spoken language from a wide range of sources. Initial efforts for creation of the BNC began in 1991 and finished in 1994, with the first edition release being announced in February of 1995. Release of the second edition did not occur until 2001. The BNC project was carried out and managed by the BNC Consortium. The consortium members were led by Oxford University Press. Other members include dictionary publishers Addison-Wesley

Longman and Larousse Kingfisher Chambers and academic research centers at Oxford University Computing Services (OUCS), the University Centre for Computer Corpus Research on Language (UCREL) at Lancaster University, and the British Library's Research and Innovation Centre. The BNC is widely recognized and often cited as a large and useful data source for corpus analysis. Word frequency lists have been created and published describing the BNC by Geoffrey Leech, Paul Rayson, and Andrew Wilson in their 2001 book "Word Frequencies in Written and Spoken English: based on the British National Corpus."

There are several other important corpus efforts besides the Brown and BNC, and the following discussion describes a few of them.

The Lancaster-Oslo/Bergen Corpus (LOB)

The LOB corpus was an effort begun in 1970 at the University of Lancaster by Stig Johansson, in collaboration with Geoffrey Leech, and Helen Goodluck to create a corpus of British English to compare with the Brown corpus. Like the Brown corpus, the LOB contains 500 printed texts of about 2000 words each, or one million words total. The text publication year (1961) and the sampling principles are identical to that of the Brown corpus in the effort to promote the ability to make comparisons between the two.

International Computer Archive of Medieval and Modern English (ICAME)

ICAME is an international organization of researchers working with English machine-readable texts. ICAME has been organizing conferences since 1979; it produces a journal and releases corpora and software on CDROM. The corpora that it distributes include

- Brown Corpus
- LOB Corpus
- Frieburg-Brown (Frown)
  In 1992 Christian Mair set out to compile a set of corpora to "match" the Brown and LOB corpora. The difference in this data set was that it should represent language of the early 1990s and that it focused on American English.

- Frieburg-LOB (FLOB)
  This effort began in 1991 and focused on British English.
- Kolhapur Corpus
  Again the attempt was to create a corpus to compare against Brown and LOB. The salient differences here were that the corpus was of Indian English and that the samples came from published material from the 1978, instead of Brown and LOB's 1961.
- Australian Corpus of English (ACE)
  In 1986 this corpus was compiled at the Department of Linguistics at Marquarie University NSW Australia. The aim was to create a corpus of Australian English modeled again on the Brown and LOB corpora.
- Wellington Corpus
  This challenging effort began in 1987 to collect half a million words of informal conversational speech as well as other categories.

International Corpus of English (ICE)

The International Corpus of English is the name of an effort that began in 1990 to gather or create electronic corpora representing national or regional variations of English to stimulate analysis of national variety. Fifteen research teams used a common corpus design to create corpora of one million words of spoken or written English after 1989.

Methodology

The following section provides a description of the parallel analysis architecture that is at the heart of this entire analysis. This is followed by a chart with an overview of the successive steps that are used to analyze a file. Attention is given to any of the software dependencies required for each task, followed by a description of each of the simplifying assumptions and their repercussions. The last topic in this methodology section describes the structure of the results format in Resource Description Framework

The parallel process at the pinion of this effort for generating string frequency metadata has only three discrete steps. Initially, the data set is acquired. Second, a list of files is created that determines what is to be analyzed. Last, the task is started using the list of files, and the analysis operation is influenced by three simple run time parameters.

Step 1: Acquisition of the data set

The objective of this step is to get the data set local to the machine performing the analysis. This requires that the files are downloaded from the Internet. Project Gutenberg's main site warns about "spidering" their site so the conservative approach is to use a mirror site for any type of automated retrieval mechanism such as the one described here. It is likely they are not concerned with "spidering" per se; rather, it is likely they are concerned about excessive load being placed on their servers through automated retrieval. The Project Gutenberg warning against "spidering" is likely an imprecise use of language, which unfortunately characterizes much of the project. Since Project Gutenberg's main site explicitly warns against overloading their resources, they were indeed retrieved from one of the mirror sites. This is shown in the following example. It is recommended that the files are

stored under a directory with a unique name like "gutenberg" because that will be crucial in the second step.

Storing the Project Gutenberg archive's zip files required a considerable amount of free disk space. The data set took more than 16.2 gigabytes of disk space for the more than 80,000 compressed files and directories. Therefore, it is recommended to have at least 20 gigabytes of space free before attempting this step. The additional space will be required because the files will be uncompressed one at a time.

The process for retrieving files starts at the UNIX command line. At the UNIX command line interface (CLI) a command like "wget" makes it easy to download an entire ftp archive or website. The command has many features that lend themselves to precise control in the automated retrieval of a large amount of information. One extremely useful feature of the wget command is the ability to resume a download without overwriting any existing files. This is achieved by using the -nc (noclobber) switch. Users who share bandwidth with others will be happy to use of the --limitrate switch to prevent using up all the available bandwidth. An example from the wget man page shows that -limitrate=20k limits the download speed to 20 kilobytes a second. Although this switch slows down the download, it is a less intrusive mechanism for retrieving files on a shared connection. Last, this analysis focuses on only the zip format files instead of all the other file formats present in the Gutenberg archive. When one wishes to retrieve only the zip format files he uses the –accept=zip switch. A complete command for acquiring all the Project Gutenberg zip archives from a mirror site using these different command switches would look like the following when typed at the command line:

```
wget -r -nc --accept=zip --limitrate=20k http://ftp.archive.org/pub/etext
```

Should the download not complete in its entirety, it is safe to simply re-type this command exactly as it is written, and the download will resume exactly where it left off because of the -nc flag. There are implications to downloading files from a mirror site. The mirror site supports a faster download speed than the modest speed of the central site, but experience shows that the data on the mirror site did not contain the most up-to-date files. The decision to use a mirror site did not have a noticeable effect on the quality of the information, but it limited the data set as a whole. Technically speaking, there is a possibility of yet another limitation. It is possible that the files on the mirror site are corrupted while the information on the central site is fine. In fact, several of the files were corrupted. Since Project Gutenberg has not adopted any method for validating data quality such as a checksum, it would be up to a user to first determine that there was data corruption and then manually determine whether it existed on the central site or mirror site or was the result of the end user's data transfer. This architectural deficit was documented and articulated in Reck 2006.

Step 2: Limit or restrict the data set to narrative files

Project Gutenberg includes files that are likely to confound the results of a string frequency analysis; hence, they are avoided. Files that researchers might consider omitting include those that contain the results from sequencing of the human genome. The human genome files inflate the data set from 1 billion to more than 8 billion strings. Very few of the strings in the human genome files would be regarded as "words" to any serious language researcher. Unfortunately, the steps that limit the data set require a manual process. This analysis determined that several of the compressed archives were corrupted. As mentioned previously, checksums were not available from the producers of the data. It was too labor intensive and unproductive to isolate where exactly the problem occurred, so the problematic

files were omitted from the analysis by removing them from the file list once a corrupted file was identified.

The only thing required at this point in the analysis is a text file with a list of files that contain the absolute path to where they exist on the local system. Each file needs to be listed on a line delimited by an end of line character. The entire computational analysis is driven by this list of files, and the analysis cannot proceed without it. This step makes use of the UNIX "find" and "grep" commands, which are also required elsewhere in this analysis. The "grep" and "find" commands are very common utilities and are likely to be present in all current versions of UNIX/Linux.

The approach for listing the files for the analysis has two parts. The first step involves making a list of all the files on the operating system and storing the list in the file /00index.txt. The following command sequence creates the list when typed at the UNIX command line:

```
cd /; find . / -print >/00index.txt
```

The “find” command may take some time to run completely as it will read the entire stored memory of the computer and any mounted file systems. The larger or slower the hard disk, the longer the process will take. Once the list creation is complete, the second step involves trimming down the entire machine’s file list to only the relevant files. The original file list is shortened with the following simple yet lengthy command:

```
grep guten /00index.txt |grep zip  |grep -v "^\." |grep -v  "h.zip" |grep -v "/old/"  |grep -v "\-8" | grep -v hgp | grep -v PG2003-08 |grep -v pdf | grep -v 10681 | grep -v images |grep -v "\-0" |grep -v "\-doc" |grep -v rdf$ >/tmp/gutenfiles.txt
```

This command leverages the fact that the files were stored in a directory below a "Gutenberg" directory. The grep command further limits the listing to filenames with "zip." The repetitive use of "grep –v" omits files that are zipped but were determined to be problematic through trial and error.

It is likely that some of the files that were omitted during the list creation did not need to be omitted. Given that several thousand files are being analyzed, it is too labor intensive to check each problematic file manually to discover the cause of the problem, and this current script works well.

Step 3: Start the analysis task

The analysis task is run through a simple script called runmeta.pl that operates as a controller. runmeta.pl is presented in Appendix B. This "controller program" has three configuration parameters that determine how it operates. The word "load" is a UNIX term for representing how much work a system has waiting to be done at any given moment. A machine at load "10" has ten things waiting for time on the processor. The runmeta.pl script checks to see if the machine's load is under a specific threshold like "10,"and if it is, then it starts a task and waits a period of time called "wait" before performing the load check again. If the load is above the maximum threshold then the machine waits a "sleep" period before performing the load check again. This cycle continues until all the files are processed. The machine never waits for a specific file's analysis to complete; it merely determines based on the load whether another task is feasible. The three configurable components are the maximum acceptable load ($load), the time to wait between checking load and starting tasks ($wait), and the amount of time to wait once the maximum load is reached ($sleep). The time parameters for the variables $wait and $sleep are expressed in millionths of a second, so the

value “5e6” is .000005 seconds. Through careful manipulation and testing of these parameters, the analysis was capable of generating 3300 assertions a second for the analysis of the entire archive. Table 1 below shows some sample runs with the parameters and their influence on the analysis output.

Table 1:
Configuration Parameters for runmeta.pl

LOAD	SLEEP	WAIT	FILES	ASSERTIONS	ASSERTIONS PER SECOND	TOTAL SECONDS
10	5e6	8e5	228	7944422	15227	520
10	8e6	1e6	228	7691389	15696	490
25	8e6	1e6	228	757995	16370	463
10	8e6	4e5	228	7710264	13917	554
10	8e6	8e5	228	7843605	15409	509
15	8e6	1e6	228	7775545	16335	476
15	12e6	11e5	228	7775550	16614	468
15	15e6	13e5	228	7644214	15958	479
20	20e6	15e5	228	777550	16438	473
20	18e6	16e5	228	777550	16940	459
30	18e6	16e5	228	764442	15505	493

Table 2 reflects the same program after a substantial kernel and hard drive upgrade. This demonstrates that the hard drive and kernel can have a substantial influence on performance.

Table 2:
Configuration Parameters for runmeta.pl with Faster Hard Drive

LOAD	SLEEP	WAIT	FILES	ASSERTIONS	ASSERTIONS PER SECOND	TOTAL SECONDS
30	18e6	15e5	228	7944650	22961	346
30	18e6	11e5	228	7944650	31033	256
30	18e6	5e5	228	7944650	33380	238
30	18e6	9e5	228	7775773	35998	216
30	22e6	8e5	228	7579894	36267	209

The final line of this chart shows that the final values that were used for the maximum load was “30,” the time the computer slept when reaching the maximum load threshold was “22e6,” and the time that elapsed between jobs initializations was “8e6.” The controller program's operational flow is be summarized by the diagram in Figure 1 below.

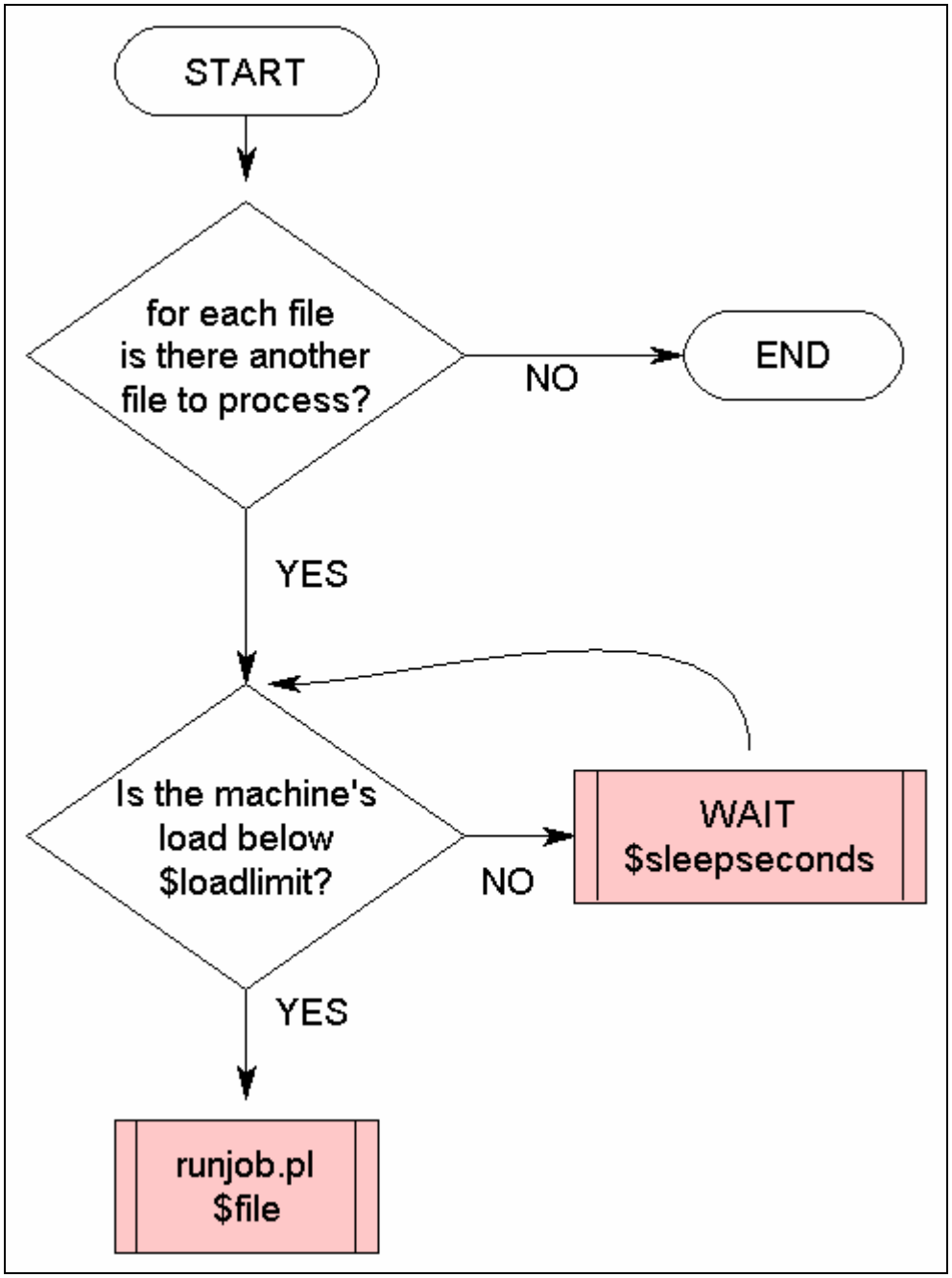


Figure 1:
Operational Flow of Controller Program

Each individual analysis task focuses on only a single text, and in the operating system these tasks are called jobs. Jobs operate independently and cannot adversely affect any of the other jobs that are running. Each analysis goes through approximately fifteen steps before the creation of the metacard is complete. Some of the steps are required for the task to complete fully, while others are optional. Some of the steps require that additional supporting software is added to the operating system. Table 3 lists the steps a job goes through before completing.

Process Overview

Table 3 lists the steps in the analysis process in chronological order. The third column shows whether a step is required, and the fourth column shows whether the step involves any software dependencies that are outside of the standard Linux operating system.

Table 3:
Process Overview

Step	Name	Required	Dependencies	Comment
1	Create Workspace	Yes		Out of scope - See Appendix TBD
2	Determine Archive Components	No	zipinfo	Out of scope - See Appendix TBD
3	Create Temporary File	Yes	File:Path File:Copy File::Basename	http://www.cpan.org
4	Determine File Type	No	file	ftp.astron.com Out of scope - See Appendix TBD
5	Convert end of line characters	Yes	dos2unix	www.thefreecountry.com/tofordos
6	Convert to UTF-8	Yes	iconv	`www.gnu.org/software/libiconv/`
7	Count Line and Characters	No		Out of scope - See Appendix TBD
8	Determine etext number	Yes		
9	Determine Producer	No		Out of scope - See Appendix TBD

Step	Name	Required	Dependencies	Comment
10	Determine number of sentences	No	style	www.gnu.org/ software/diction/diction.html Out of scope - See Appendix TBD
11	Determine Syllables per word and words per sentence	No	style	www.gnu.org/ software/diction/diction.html Out of scope - See Appendix TBD
12	Determine Release Date	No		Out of scope - See Appendix TBD
13	Determine string frequencies	Yes		
14	Print results	Yes		

As Table 3 indicates, there are fourteen individual steps in the creation of a metacard. Seven of the steps are considered “required” and the other seven are optional. Of the required steps three entail software that is not traditionally part of the Linux or UNIX operating system and needed to be added. The table’s “comments” column shows where any additional required software is found. Also, several of the steps are not directly relevant to the preparation of the file or the generation of string frequencies. These non-essential steps are part of the process so they are listed in the table, yet they are outside the main thrust of this paper and therefore are labeled “out of scope.” Additional information describing those steps is presented in Appendix TBD instead of the main paper.

Of special note are the two required steps that are not for workflow. Both step number 8 “conversion of end of line characters” and step number 9 “conversion to UTF-8” are based on a methodology adopted for file processing. While under ideal circumstances neither of these steps would be required, the lack of file metadata provided by Project Gutenberg and more importantly the overall lack of data integrity and data consistency dictated that a strategy be adopted to mediate unpredictable variations that could influence results.

The steps that are intended to simplify the process are considered simplifying assumptions. Those are detailed in the order they occur in the following section.

Simplifying Assumptions

The multi-step process detailed in the Process Overview section involved a number of simplifying assumptions. As this effort is a largely academic exercise and not intended directly for commercial consumption, these can be considered reasonable strategies for overcoming obstacles that would otherwise prevent useful results. This section reviews and details the assumptions while providing reasons that they were deemed necessary.

Unique Work Space: Since each archive needs to go through a multi-step process, it is useful to make sure that multiple threads or processes working in parallel cannot affect each other. Therefore, each file is processed in a separate directory. When the analysis on a file begins, a unique directory is created expressly for that file. The directory is named based on the epoch time at that exact moment. Epoch time is a time measurement in seconds since midnight on the morning of January 1, 1970, not counting leap seconds. Basically, it is just a big number. While there are no linguistic implications to this approach, a more scalable mechanism would use a directory name that is necessarily unique, which this does not do. A necessarily unique identifier in a parallel model would be the process id of the thread or a large random number or a combination of these two. Furthermore, it would be possible to check for an existing directory with the same name as the one being created, and if one is found then a new name could be selected. Since neither of these methods was used, a useful improvement could be made to this procedure in this area as a future effort.

Working with a File Copy: The analysis procedure makes a copy of the file to be analyzed and places it in the temporary directory. This protects the original file from

alteration and allows the analysis to be run again because unadulterated source files continue to persist. There are two requirements for this step. The first requirement is the Perl programming language modules File::Path, File::Copy, and File::Basename, which are available from the CPAN archive (http://www.cpan.org). These are likely to be already installed anywhere the Perl is. The second requirement is enough disk space to accommodate the uncompressed archive. Uncompressed files are likely to take up more than twice as much disk space as the original file. For example, a file like poe3v11.zip takes up 229376 bytes uncompressed but 602112 bytes when uncompressed. The Perl modules allow the file to be copied easily without having to manage the path information. This keeps the subroutine for performing this copy down to mere five lines of code.

Conversion of End of Line Characters: This step is to ensure consistency between the input files. Many of the files have a consistent use of end of line characters in DOS format. Previous discussions by Reck 2006 have highlighted the lack of quality control, uniformity, and predictability in the files of Project Gutenberg. Variability in the end-of-line characters will likely cause unanticipated results because of the extensive use of Perl' array data structure, which hinges on consistent end of line characters. The entire analysis is contingent upon managing the files line by line; therefore, this step is absolutely critical. Only 90% of the data set had consistent end of line characters, hence the need to make a conversion. This is a required step, and it depends on the "os2unix"command. The command was not installed by default in Debian Linux. Therefore, it needed to be compiled and added to the operating system. The command used in this analysis is version 1.7.6; it is credited to Christopher Heng and is distributed under GNU General Public License Version 2. Versions of the dos2unix command differ as to whether they use the -q command switch.

Once this command is completed, a temporary file from the archive is converted. Unlike other commands, there is neither an output file specified, nor do the results go to STDOUT. This change is likely to alter the characteristics of most if not all the files. Since neither the number of lines nor their characteristics affect the frequency measurements, it is felt that this alteration is acceptable. It is important to notice that the characteristics of the original file are documented before this alteration. The original file's attributes are captured and articulated in pg:ftype attribute.

Conversion to UTF-8: Much like the End of Line Conversion step, this milestone is to increase consistency in the input files. This step in the analysis ensures that input files have a uniform character set, otherwise known as a character encoding. The variability of character encodings was problematic in the early stages of this effort; hence this step was adopted as a "fix." There are three basic concepts surrounding the representation of letters or glyphs electronically. For clarity, the following discussion briefly reviews the three concepts: character repertoire, character code, and character encoding. A "character repertoire" is the set of characters a system supports. Well known character repertoires include ASCII and the ISO 8559 series. Character repertoires include decisions about how to divide writing systems into logic groups. Notable challenges arise in situations where languages, like Arabic and Hebrew, join glyphs together in certain situations. A "character code" specifies the one-to-one mapping between the members in character repertoires and codes called "code points." This is further complicated because there are several synonyms for code points including code number, code value, code element, and code set value. Differences between code points in character repertoires may be small; for example, the ISO8859-15 character repertoire uses the code point A8 to represent "LATIN SMALL LETTER S WITH CARON"

where the same code point in the ISO8559-1 represents “DIAERESIS.” Otherwise, these two character repertoires are practically identical.

A “character encoding” is a method of presenting characters in digital form by mapping code numbers into the sequence of octets. When a character is represented by a single 8-bit octet it limits character repertoires to only 256 characters, which is the heart of the problem. A powerful approach for solving this limitation is to use the variable length character encoding for Unicode called UTF-8. Therefore, in the name of simplicity and uniformity, this analysis converts all character set repertoires into UTF-8 before the string frequency analysis is performed. Metadata was gathered and expressed for the character set. This file metadata was captured only in cases where a line occurred in the first 50 lines of the text file with the words “Character set encoding:” This information was found, in only 72% of the files (10778 out of 14811). In situations where the metadata was indeed found, there was still variability in the label names that were used. In essence, any metadata supplied by Project Gutenberg is sufficiently unreliable to make programmatic decisions upon it. Given the otherwise unpredictable variations in input file encodings, this conversion is a required step. The “iconv” command is required for this milestone. The version of the inconv command used in this analysis is version 2.4. The command "iconv" was written by Ulrich Drepper as part of the GNU C Library. The "iconv" command may not be a standard component in the UNIX/Linux operating system and may need to be installed; it is available from http://www.gnu.org/software/libiconv/. The exact command syntax for achieving the conversion is as follows:

```
iconv -c -t utf8 –o $file-b4 $file
```

The command switch "-t" could be modified to convert to an encoding other than UTF-8, especially as the "iconv" command version 2.4 supports approximately 1145 different encodings. As the command documentation states, not all encodings are variations because a character encoding can be referenced by several different names. The point here is that there is a level of flexibility to the encodings and this is the place other modification could occur. Without question, there was a substantial variation in input files from Project Gutenberg, and conversions were necessary to achieve consistent results.

Determination of Etext Number: Each book in Project Gutenberg is assigned an Etext number. In the most currently released text files, this piece of metadata is generally available on a line marked by "Release Date:" Legacy texts suffer from a dreadful lack of consistency for indicating the Etext number. As pointed out in Reck 2006, even then the word Etext is not used consistently as there are also Ebooks. Unfortunately, determining the Etext is a requirement because the URIs for identification of the text file resource and string resource leverage the Etext number. In situations where an Etext number cannot be determined, the file is completely ignored and the metacard for that file is not created. Given Project Gutenberg's lack of consistently expressing this piece of metadata, a number of different approaches are used to try to determine what the number for a given file is. For any given file, it is not clear which approach was successful for determining the text number. Some books call the Etext number an ebook number; in those cases, the ebook number is regarded as the Etext number. In the majority of cases, the text number is found on the "Release Date" line. This analysis has a dependency on this step, but it is only an artifact of the resulting output format. Unfortunately, texts may be omitted from the analyses that are not otherwise problematic. It is likely that the omitted texts are the product of a certain producer/editor who

chose to adopt their own naming or labeling convention. An argument could be stated that certain editors' contribution to Project Gutenberg may actually harm the effort more than help it. Had an idiosyncratic labeling mechanism not be used, someone else may have performed the editor's role and possibly used some consistent labeling mechanism.

RDF as XML

Resource Descriptive Framework (RDF) is a method for specifying metadata about a thing. The method is more accurately called a "framework," and a "thing" is more accurately called a resource because of how RDF refers to things. One common way to write RDF is with the Extensible Markup Language (XML). A person who is new to XML is likely to notice that XML uses tags created from less than "<" and greater than ">" symbols called angled brackets. These tags, which are called elements, occur in pairs and are used to segment information. The first member of the pair is called a start tag and is represented by brackets enclosing a string or strings like <sample>. There is a corresponding and matching "end tag" that closes off the segment of information in a similar fashion except with the addition of a slash character like </sample>. The entire segment of information would look something like <sample> This is a sample that shows XML elements that contain a sentence.</sample>

XML elements are intended to be self-describing because the tag's name often describes the information that the section contains. It is beyond the scope of this paper to describe either RDF & XML in precise detail, but the crux of the matter is that RDF is used to describe a book and the strings it contains. More specifically, RDF describes the frequency of the strings inside a book. It is helpful to understand that both books and words are referred to as "resources," and each is described in a similar fashion.

Parts of an RDF statement: All statements in RDF have three components; in fact, RDF statements are sometimes called "triples." The three-part descriptions convey how a subject resource relates to an object resource based on a relationship called a predicate. To reiterate, the three parts of an RDF statement are the subject, the predicate, and the object. All string frequency statements for describing a book resource regard the book as the subject and use the predicate <pg:occurrence>. In each frequency statement, the object is the string that is being measured. When any resource is described using RDF there is always a start tag that contains a “rdf:about=.” Then, following the equals sign is a string called Uniform Resource Identifier, which is abbreviated URI.

def : A Uniform Resource Identifier is a standard means of addressing resources on the Internet using a formatted string. The string then can operate as a name or location for the resource. The Uniform Resource Locator (URLs) used in web browsers are a type of URI.

Sometimes the URI's are indeed bona fide URL web addresses. After the starting element that contains "rdf:about" there is likely to be other XML elements. Any other elements inside the scope of the rdf:about tag section reveal information that further describes the resource the URI points at. Simply put, the URI is just a way to point at something. In the metacards, the technique is identical for the book resources and word resources or actually anything else described in RDF. In the metacards, the start of the section that describes a book looks like:

<book:Book rdf:about="ftp.archive.org/pub/etext/etext04/agasz10.zip">

In this example, the book tag contains a URI, which indicates which file is being described. In this case the file is agasz10.zip, which exists in the “pub/extext/etext04” directory on the

machine “ftp.archive.org.” The elements that follow the book tag contain other tags that depict attributes that further describe the book. Each of the strings in the book operates as its own resource and in turn has it own individual tag. The word’s presence in the book is captured by virtue of its pg:occurrence tag occurring between the open book tag <book:Book and the close book tag </book:Book> for that file. The examples in the following discussion give a concrete example of the tags used to describe a word in a metacard.

For each of the strings in the book there are word resources tags like the following example for the word "abandoned":

<pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abandoned"/>

A sample metacard showing this line as it occurs in the metacard is shown in Appendix C.

It is helpful to recognize that this tag has three distinct components. The first component is the predicate “pg:occurrence,” which indicates that the object resource it points at "occurs" within the book. The second component is "rdf:resource," which indicates that the URI that follows it points at another resource. Last, the URI http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abandoned is a long way of specifying the word "abandoned" inside text number #7020. In summary, the URI represents that the word "abandoned" occurs in the file agasz10.zip, which is text number #7020. The rest of the frequency information for the string “abandoned” occurs in section of RDF that follows that URI. The compliment to this RDF for describing the book is RDF that describes the string as "abandoned." It is similar to the book RDF in that it starts with "rdf:about=" as shown in Figure 2.

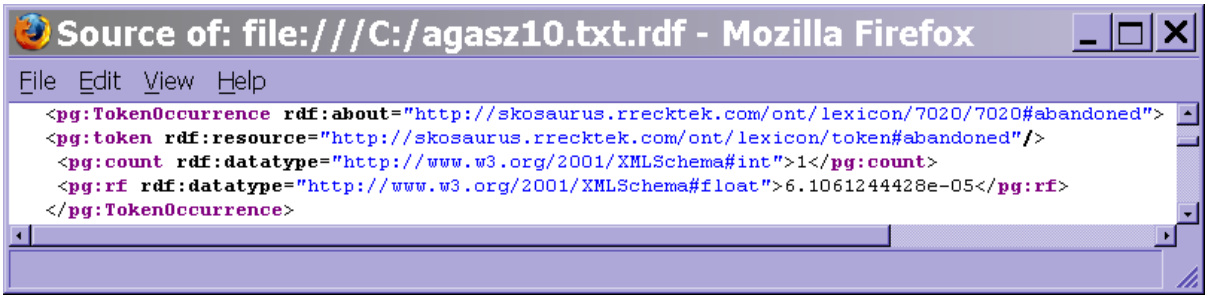


Figure 2:
Metacard Section Showing Relative Frequency for the Word “abandoned”

This section of RDF says that the resource "abandoned" (in text #7020) occurs only once, hence the “1” contained in the pg:count tags. The number of occurrences for a word always occurs in a line with <pg:count rdf:datatype="http://www.w3.org/2001/XMLSchema#int">”. In this case the count just happens to be the numeral "1". This integer for the number of occurrences (1) is divided by the number of total words in the book. The number of words occurs in the pg:wordcount tag. In this specific situation the pg:wordcount value is 16377 and computes to create the pg:rf value 6.1061244428e-05. An identical procedure happens for each of the unique strings in each of the books. The total number of RDF statements created to describe the archive is approximately 380 million.

Results

The following section will characterize the extensive results from the analysis. As has been explained previously, the results from the string frequency calculations are rendered in a specific type of XML called RDF. The set of results involve thousands of files called metacards, one for each of the analyzed files. Since there were 14,572 files analyzed, it means there are exactly the same number of metacards. The metacards can be considered large by today's standards as they range in size from 1 megabyte to 28 megabytes. Taken together, the metacards encompass 42,445 megabytes of data when they are uncompressed.

By today's standards this is a large amount of information and requires a substantial amount of resource to both generate and manage. The processing of metafiles was conducted in late 2006 on an AMD Athlon(tm) 64 X2 Dual Core Processor 4400+ with 4 gigabytes of random access memory, which cost approximately $2000 in 2006. In total, 474,723,464 RDF based assertions were created over about a four and a half hour period (15682 seconds). The assertion creation rate averaged 30271 assertions per second for the duration of the task. The task was run several times so that the optimal run time parameters for the maximum load, wait, and sleep times could be determined.

Most common authors

Table 4 lists the most common authors in the archive. While Mark Twain has many writings in the archive, they do not represent a large percentage of the archive as a whole. No single author represents a large portion of the 4816 different authors in the archive. Author metadata was determined in 13325 (91%) of the files.

Table 4:
Most Common Authors

Number of files	Percentage of data	Author as determined by metadata provided inside file
1133	8.5%	Various
149	1%	Mark Twain
136	1%	Anonymous
129	<1%	Honore De Balzac
125	<1%	William Shakespeare

Most common editors

Table 5 shows that four producers were responsible for 3435 (45%) of the texts in the archive. Producer metadata was determined in only 7416 (50%) of the texts. Three thousand ninety-three different producers were detected.

Table 5:
Most Common Editors

Number of files	Percentage of data	Producer as determined by metadata provided inside file
1337	18%	David Wiger
907	12%	Distributed Proofreaders
821	11%	Juliet Sutherland
370	4%	Charles Franks

Most Common File Types

As Table 6 depicts, 92% of the files in the archive were one of three types; ASCII English text, ISO-8859 text or Non-ISO extended-ASCII English text. What the chart does not communicate is that there were at least 95 different file types analyzed.

Table 6:
Most Common File Types

Number of files	Percentage of data	File Type as reported by the 'file' command
12361	83%	ASCII English text, with CRLF line terminators
1190	8%	ISO-8859 English text, with CRLF line terminators
272	1%	Non-ISO extended-ASCII English text, with CRLF line terminators

Number of files	Percentage of data	File Type as reported by the 'file' command
132	<1%	Data
130	<1%	ASCII English text, with CRLF, LF line terminators
125	<1%	Rich Text Format data, version 1, ANSI
74	<1%	ASCII English text, with CRLF, CR line terminators
54	<1%	UTF-8 Unicode English text, with CRLF line terminators

Number of Files in Archive

Table 7 shows that the vast majority (98%) of the zip archives contained only a single file.

Table 7:
Number of Files in Archive

Number of file in archive	Percentage of dataset	Number of data set files
1	98%	14556
2	1%	150

Characters per word

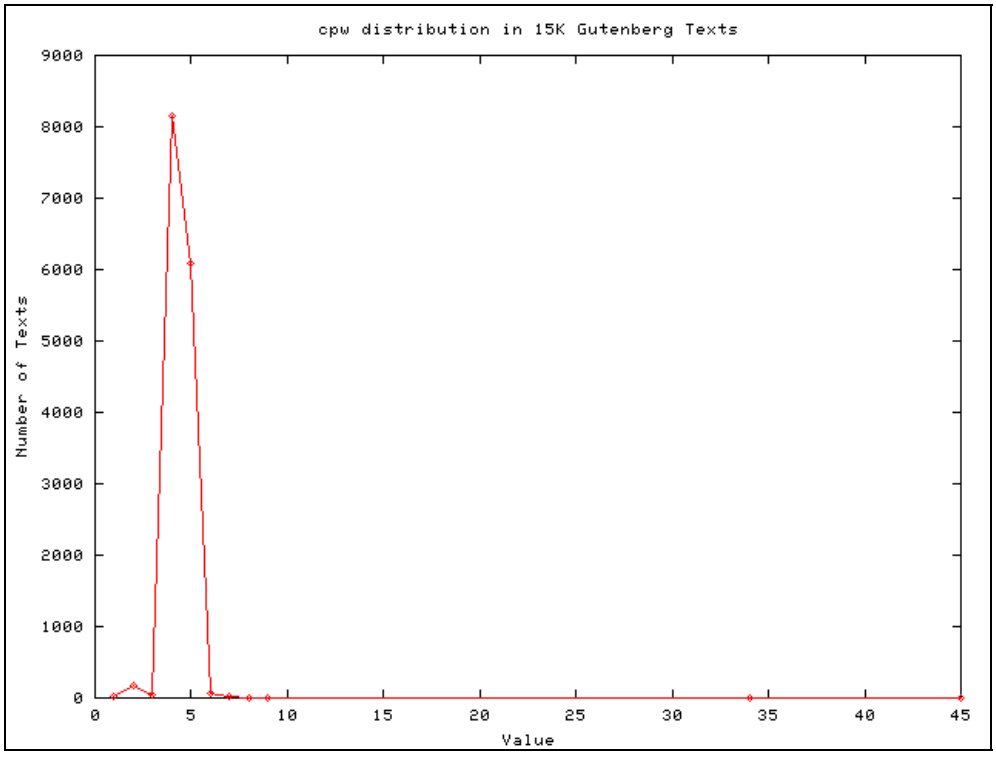


Figure 3:
Plot of Characters per Word for Project Gutenberg

Figure 3 shows the distribution of characters per word. The average "characters per word" was calculated for each of the books in the archive using the style command as explained in the "Methods" section of this paper. More than 8000 books had a value of "4" as the average word size. The next most common average word size was "5." The number of characters per word is likely to be correlated to the language of the text.

Words per sentence

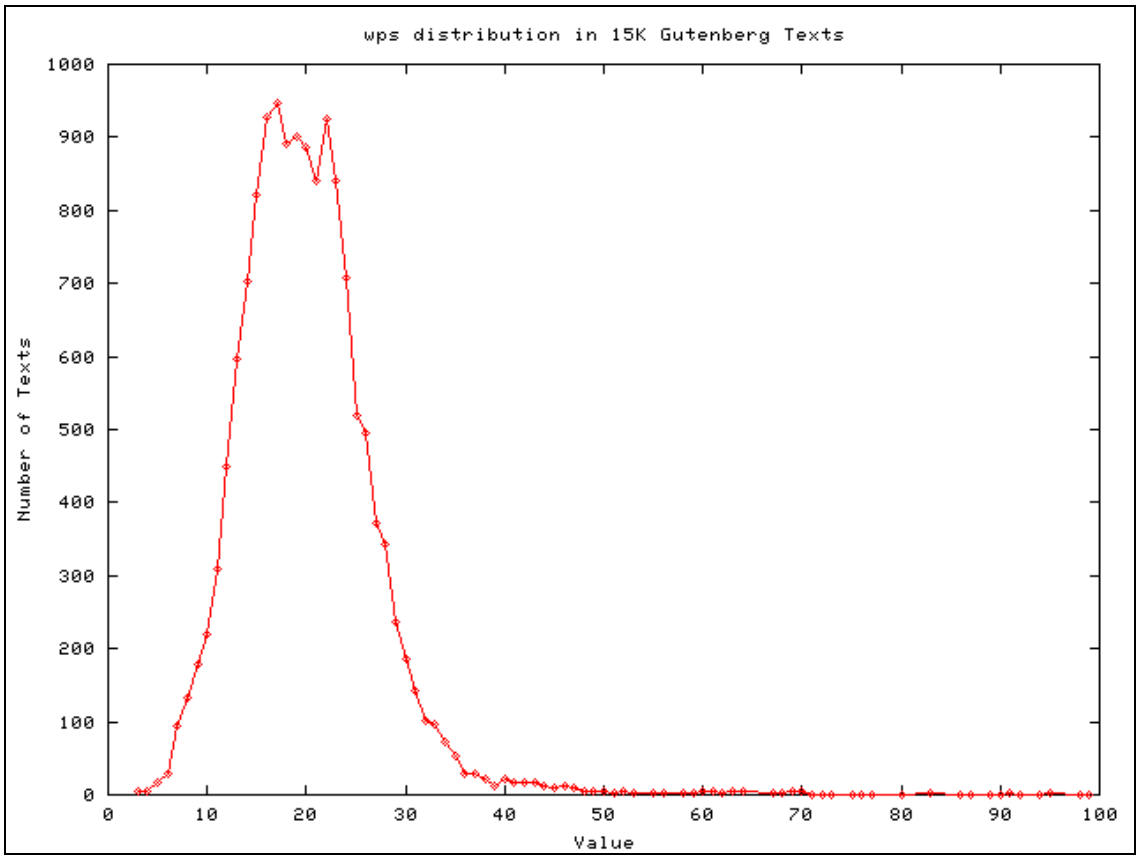


Figure 4:
Plot of Words per Sentence of Project Gutenberg

Figure 4 shows the average words per sentence as calculated by the "style" command as explained in the "Methods" section of this paper. A vast majority of the texts had average sentence lengths between 12 and 25 words.

Discussion

One thing should be obvious; all electronically accessible information is not equal, especially when it involves information interoperability. It is true there is a huge gap between information in the printed form on a page in a book and that page represented on a computer. Information represented in the computer is only the first step at making electronically accessible information maximally useful. At the most rudimentary level, a page can be represented as an image. This has some advantages over the printed page but still falls short of allowing the information to be accessed easily for machine processing. One reason for this is that the image is inherently targeted at how the information is formatted for presentation. Another logical step toward information interoperability is when the image's text is represented in a word processing format like Microsoft Word. Again, it could be argued that this is more powerful in some ways than the image, but like the image it is intended toward the presentation of the information and not access to what is written. It might be slightly more useful to have the same text without any formatting information at all and merely represented in pure ASCII. This would allow the greatest flexibility for computer consumption. Yet again, consumers could want an even more powerful and expressive format. To that end, the results of this analysis are both maximally accessible, since they are in a form of ASCII, and improved because the meaning of what is written is carried by the format. One way to achieve this is to structure the data, possibly using columns. In traditional string frequency analysis efforts, the results are presented in columns. It is not particularly difficult to understand that perhaps the first column contains the string being measured and the second column contains a numeral for describing the relative frequency for that term. Unfortunately, this requires that someone look at the presentation and keep track

of what it means for information to be in the first column. If that information is taken out of context of the column, the power behind the observation is lost. Humans project and interpret what it means to be in the first column every time the information is accessed. The analysis results from this effort take a step further in the approach of presenting information because of the use of RDF. The mere use of RDF encodes the roles and relationship semantics in the syntax of the RDF. A user does not need to determine what the information in column one means; the RDF says what it means by virtue of being in RDF. This is the power of using RDF to represent information.

RDF prescribes how to articulate a piece of knowledge called an assertion. This analysis made empirical observations about the use of language and created assertions describing those observations. If those assertions are separated from each other or regarded in other contexts, the information they portray is equally powerful. The format of the results imparts the meaning behind the observation. If any person or machine is capable of understanding RDF, the intention behind the assertions is clear without any further explanation. Unlike information presented in columns, someone does not have to keep track of what it means to be in the first column. What is meant is embodied in the RDF itself. It is the purpose to which RDF was designed and the purpose to which RDF was adopted to render the results of this analysis.

The word “machine-readable” has been used in corpus linguistics for many years. The meaning of the term is not exactly clear as the expectations of those who use it have changed as human capabilities change. It appears that many who have used the term meant something akin to “electronic accessibility.” This is likely because in the past, machines were

not capable of reading and leveraging the power behind what it meant to articulate a statement.

Working with RDF

Clearly, the product of this analysis is not the end; 42 gigabytes of RDF across 14,000 files describing in excruciating detail the contents of the Project Gutenberg archive is really only the beginning. These results see their true potential only when users create useful and interesting queries to search and understand the correlations latent in the results. Do women really use "flowery speech" more than men? Now, there is a good opportunity to find out. The next step in doing so would be to put this data into a searchable database of some kind.

There are at least two free databases that could be used to hold at least a portion if not all of the RDF information. In "Implementing a Government-wide Semantic Solution to Thesauri," Sall and Reck (2005) show how to use SOAP-based clients to enter and search data in the Kowari triple store database. Although it is no longer formally supported, Kowari purports to be able to hold one billion triples, which is twice as many as this analysis created. In "Applying XQuery and OWL to The World Factbook, Wikipedia and Project Gutenberg," Sall and Reck (2006) show how to enter and search data stored in the eXist XML database. In fact, the data resulting from this effort is the precursor to the string analysis described here.

References

Baayen, R. Harald 2001. *Word Frequency Distributions* Kluwer Academin Publishers

Beckett, D., Miller, E., & Brickley, D. (2002). *Expressing Simple Dublin Core in RDF/XML.* Retrieved from: http://dublincore.org/documents/dcmes-xml/

Charniak, E. (1993) *Statistical Language Learning* MIT Press

Chaudhri, A., Rashid, A., & Zicari, R. (2003). *XML Data Management: Native XML and XML-Enabled Database Systems Addison Wesley*

Daconta, M., Obrst, L., & Smith, K. (2003). *The Semantic Web* Wiley Publishing Inc.

Fensel, D. (2001). *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce Springer-Verlag Berlin Heidelberg*

PG - Project Gutenberg (2006). *Free eBooks.* Retrieved from: http://www.gutenberg.org/

Halliday, M.A.K, - et al.(2004) *Lexicology and Corpus Linguistics: An Introduction* Continuum International Publishing Group

Hjelm, J. (2001). *Creating the Semantic Web with RDF* John Wiley & Sons Inc.

Hockney, S. (2000). *Electronic Texts in the Humanities* Oxford University Press

Juraksky, D. & Martin, J. H. (2000). *SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* Prentice Hall.

McEnery, T. and Wilson, A. (1996). *Corpus Linguistics 2nd Edition* Edinburgh: Edinburgh University Press.

Powers, S. (2003). *Practical RDF* O'Reilly & Associates Inc.

RDF (2004). Resource Description Framework. Retrieved from: http://www.w3.org/RDF/

Reck, R. P. (2006). *Metadata Cards for Describing Project Gutenberg Texts* , OntoLex 2006: Interfacing Ontologies and Lexical Resources for Semantic Web Technologies. Genoa, Italy.

RFC 1766 (1995). *Tags for the Identification of Languages* Retrieved from: http://www.faqs.org/rfcs/rfc2046.html

Sall, K. & Reck, R. P. (2006). *Applying XQuery and OWL to The World Factbook, Wikipedia and Project Gutenberg*, XML 2006. Boston, MA.

Sall, K. & Reck, R. P. (2005). *Implementing a Government-wide Semantic Solution to Thesauri*, XML 2005. Atlanta, Georgia.

APPENDICES

APPENDIX A - Additional Metadata Elements

The following sections provide supporting details for how certain metadata elements were determined and the dependencies or impact of the methodology for doing so. The information expressed by these attributes is outside the core focus of this paper but is provided for completeness.

Determination of archive components

Objective: The purpose of this step is to determine the number of files that are in the compressed archive. Although the program flow is not directly influenced on the result for this step, this number is expressed in the metadata in several attributes as shown in Table 8 below.

Table 8:
Metadata Attributes

File Attribute	Predicate
Number of files in the archive	pg:fcount
Size of compressed file	pg:csize
Size of uncompressed file	pg:ucsize
Compression Ratio	pg:cratio

Most Project Gutenberg text archives contain only a single file. For the purposes of analysis, it is ideal if the archive contains only a single file; hence in situations where there are multiple files in the archive only the first file is analyzed. Table number TBD shows the number of archives that contained single or multiple files.

Requirements: The command “unzip –Z” is used to determine information about the archive. Although it is not immediately apparent, this command sequence actually uses and requires the “zipinfo” command. The version of ZipInfo used here is 2.42 and ZipInfo is created by Greg Roelofs. The analysis can proceed without this command, but it is likely to

exist in most versions of Linux. If the command is absent, the analysis will produce an error but it should continue.

Implications: There are no linguistically significant theoretical implications to using the zipinfo command. A more robust analysis would base the program flow on the number of files in the archive. The reason that the number of files in the archive did not influence the program flow is that each piece of metadata is linked to the archive, not to the file the archive contains. The distinction between the file in the archive and the archive itself is not exactly correct since the properties of the archive are not exactly the properties of the file. For example, Mark Twain is the author of the book "Huckleberry Finn," but he did not author the archive that contains the book. Similarly, none of the attributes of the file contained in the archive are actually the attributes of the archive itself, but drawing a distinction between the two was not formative in this analysis.

Determination of File Type

Objective: This step in the analysis determines what a file's type is. The results are expressed in the metadata predicate pg:ftype and the textual description comes from the results from the UNIX 'file' command.. This piece of metadata might be confused with the file's character set. The information about file type often contains information about the character set but includes information describing the end of line characters. The file types for the data set had a large range including approximately one hundred different file types, many of which were non-textual. A summary of the most common textual types as reported by the 'file' command appears in Table TDB.

Requirements: This step requires the UNIX 'file' command. The version of the command used in this analysis is 4.17. This command is available in most versions of Linux

or UNIX, and is available by anonymous FTP on ftp.astron.com. According to the man page for 'file' it is "Copyright (c) Ian F. Darwin, Toronto, Canada, 1986-1999. Covered by the standard Berkeley Software Distribution copyright."

Implications: While this information from this command is not directly used in the logic of the processing, the information about the type of file could be used to select files of a specific format for further analysis or conversion. More importantly, this information could be used to avoid files that are not text such as graphics (jpg, png) or sound (mp3). This information is likely to be useful in an expanded text analysis effort where various file formats like Portable Document Format (PDF) or Microsoft's DOC files are converted to text before being analyzed. Adding this capability would not be a significant challenge.

Determination of Count for lines and characters

Objective: The objective of this step is to count the number of lines and characters in the file.

Requirements: This requires the UNIX/Linux command 'wc' which is a standard component in most if not all versions of the operating system.

Process: The command 'wc' is used to determine the number of lines and characters. Although 'wc' does count the number of words, that count actually comes from the string frequency analysis step.

Implications: The count values for the number of lines and characters has no direct impact on this analysis. The crucial implication is that the number of words is correct.

Determination of Producer

Objective: The purpose of this milestone is to determine the person who was responsible for editing the text file for release by Project Gutenberg. Possibly, this

information could be used to isolate individuals for praise or condemnation based on the quality and consistency of their efforts. This information is articulated in the pg:producer predicate. Project Gutenberg does not make consistent use of the term “Producer.” The heuristic used for determining the producer variable checks for the word “Produced” and the other free variation “Prepared by” or yet a third permutation “Transcribed by.”

Requirements: There are no special requirements for this step, and the step itself is optional as there is no adverse reaction to the producer being undetermined.

Determination of Number of Sentences

Objective: This step determines the number of sentences in the file. Although the information about the number of sentences is not directly used in this particular analysis, it is expressed in the metadata in the event that future analyses might correlate authors with sentence length.

Requirements: This step requires the “style” command, which is generally not part of a standard UNIX/Linux operating system. The “style” command is included as part of the “diction” package created by Michael Haardt and is available under the GNU software license. This step is not essential; hence, if the style command is not installed the analysis will still proceed but will leave approximately 11 file attributes unspecified.

Determination of the Average Number of Syllables per Word and Words Per Sentence

Objective: The objective of this milestone is to determine the average number of syllables per word and the average number of words per sentence. These averages are likely to be tightly tied to the author of the work.

Requirements: The requirements for this step are identical to the Determination of Number of Sentences step above,

Process: An excerpt from the output of the style command looks like the following:

sentence info:

317 words, average length 4.94 characters = 1.56 syllables

Determination of File Release Date

Objective: The purpose of this milestone is to determine the purported date for the file's release in Project Gutenberg file. This piece of metadata is particularly useless to anyone outside of Project Gutenberg. It does not actually reflect the data that the file was released; it reflects the project's goal for the file's release. If the project exceeds its initial goal by releasing the file earlier than expected, files can often be actually released prior to the release date as specified. To anyone outside of the Project Gutenberg world view, this can only serve to confuse. If half the effort was used to track and supply truly useful metadata like the author's actual publication date, the copyright date, or the book's original publisher that would be significantly more useful. As the date occurs in a variety of formats in the Gutenberg files, additional effort is exerted to consistently present the date in the metadata in accordance with the ISO 8166 format.

Process: The file release date is determined using the following regular expression:

```
if ($line =~ /^release\s+date:/i)
```

Requirements: There are no special requirements for this step, and it is optional.

Implications: This information is likely not only to be useless to individuals outside of Project Gutenberg, but also to confuse them. This information is slightly useful in that epitomizes Project Gutenberg's distorted view of providing metadata. The impression is that they are trying to demonstrate that their level of productivity exceeds the original goals they established. Since it is difficult to assess how reasonable the initial goals were, knowing the

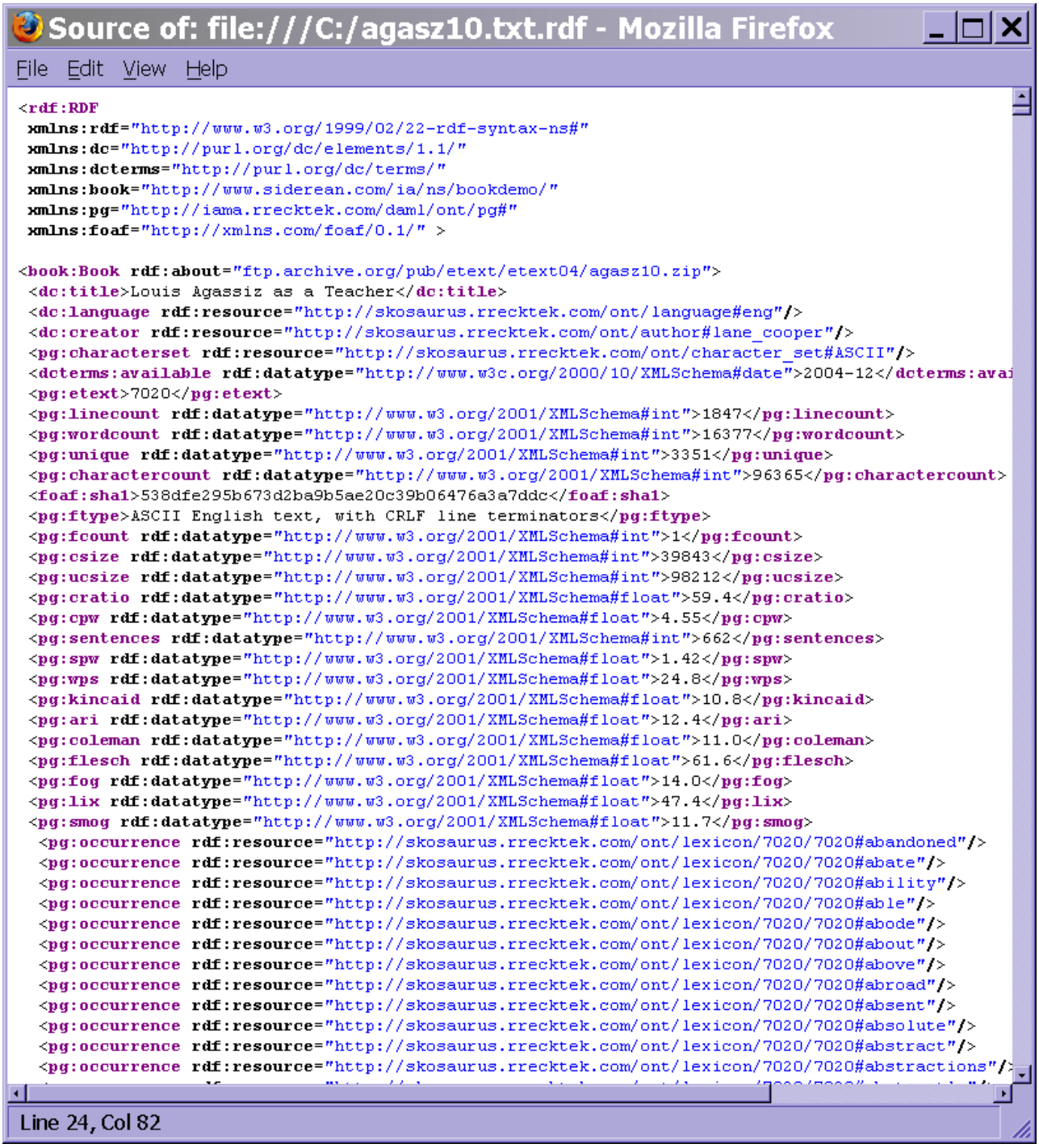
goals were surpassed, or by how much, expends energy that might been better dedicated to providing meaningful information.

APPENDIX B - runmeta.pl

```
#!/usr/bin/perl

###
### This is the ONE file to rule the entire job.
###
### Copyright 2006 Ronald P. Reck

use Getopt::Std;
use strict;
use warnings;
use Time::HiRes qw( usleep );

$| = 1; # autoflush

my $verbose;
my $start;
my $sleep = 22e6; # microseconds make this a config param.
my $wait  = 8e5; # between iterations of the for loop.

our ( $opt_l, $opt_v, $opt_s, $opt_h );
getopts('hvl:s:');
$opt_h and die "Usage: $0 [-v] [-s SKIP] [-l LIMIT]\n";

# this is the flag to turn on verbose messages
$verbose = $opt_v;
print "\n $0 is turning on verbose messages." if $verbose;

my $version = "1.0";
print "\n $0 version $version" if $verbose;

# this is the limit to keep load under unless
# otherwise expressed on the command line
my $limit = $opt_l || 30;
print "\n $0 is setting the load limit to $limit." if $verbose;

$start = $opt_s;

# start variable can help alot when restarting the job
$start ||= 0;

# get the files to work with
my @files = `cat /tmp/gutenfiles.txt`;
chomp @files;
print "\n $0 read ", scalar(@files), " files.\n" if $verbose;
```

```
my $filecount;

foreach my $file (@files){
   if (++$filecount < $start) {
      print "\n skipping $filecount" if $verbose;
      next;
   }

   while ( load_too_high() ) { }
   my $date = localtime;
   print "\n$date file number:$filecount: filename:$file:";
   if(fork() == 0){exec("/home/rreck/rbin/runjob.pl -f $file ")}
   usleep($wait);

}


###
### this check the machine's load
###
sub load_too_high {
   # this is the number of seconds to sleep when the load
   # is over the limit
   my $load = `uptime`;
   my ( $load1, $load5, $load15 ) = $load =~ /load average: ([.0-9]+), ([.0-9]+), ([.0-9]+)/;
   if ( $load1 > $limit ) {
      print "\n $0 thinks load $load1 is over $limit so sleeping $sleep usec" if $verbose;
      #sleep $sleep;
      usleep( $sleep);
      return 1;
   }
   print "\n $0 thinks load $load1 is under $limit" if $verbose;
   0 # load not too high
}
```

APPENDIX C – METACARD WITH LEXICAL ENTRIES




```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:dcterms="http://purl.org/dc/terms/"
 xmlns:book="http://www.siderean.com/ia/ns/bookdemo/"
 xmlns:pg="http://iama.rrecktek.com/daml/ont/pg#"
 xmlns:foaf="http://xmlns.com/foaf/0.1/" >

<book:Book rdf:about="ftp.archive.org/pub/etext/etext04/agasz10.zip">
 <dc:title>Louis Agassiz as a Teacher</dc:title>
 <dc:language rdf:resource="http://skosaurus.rrecktek.com/ont/language#eng"/>
 <dc:creator rdf:resource="http://skosaurus.rrecktek.com/ont/author#lane_cooper"/>
 <pg:characterset rdf:resource="http://skosaurus.rrecktek.com/ont/character_set#ASCII"/>
 <dcterms:available rdf:datatype="http://www.w3c.org/2000/10/XMLSchema#date">2004-12</dcterms:avai
 <pg:etext>7020</pg:etext>
 <pg:linecount rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1847</pg:linecount>
 <pg:wordcount rdf:datatype="http://www.w3.org/2001/XMLSchema#int">16377</pg:wordcount>
 <pg:unique rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3351</pg:unique>
 <pg:charactercount rdf:datatype="http://www.w3.org/2001/XMLSchema#int">96365</pg:charactercount>
 <foaf:sha1>538dfe295b673d2ba9b5ae20c39b06476a3a7ddc</foaf:sha1>
 <pg:ftype>ASCII English text, with CRLF line terminators</pg:ftype>
 <pg:fcount rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</pg:fcount>
 <pg:csize rdf:datatype="http://www.w3.org/2001/XMLSchema#int">39843</pg:csize>
 <pg:ucsize rdf:datatype="http://www.w3.org/2001/XMLSchema#int">98212</pg:ucsize>
 <pg:cratio rdf:datatype="http://www.w3.org/2001/XMLSchema#float">59.4</pg:cratio>
 <pg:cpw rdf:datatype="http://www.w3.org/2001/XMLSchema#float">4.55</pg:cpw>
 <pg:sentences rdf:datatype="http://www.w3.org/2001/XMLSchema#int">662</pg:sentences>
 <pg:spw rdf:datatype="http://www.w3.org/2001/XMLSchema#float">1.42</pg:spw>
 <pg:wps rdf:datatype="http://www.w3.org/2001/XMLSchema#float">24.8</pg:wps>
 <pg:kincaid rdf:datatype="http://www.w3.org/2001/XMLSchema#float">10.8</pg:kincaid>
 <pg:ari rdf:datatype="http://www.w3.org/2001/XMLSchema#float">12.4</pg:ari>
 <pg:coleman rdf:datatype="http://www.w3.org/2001/XMLSchema#float">11.0</pg:coleman>
 <pg:flesch rdf:datatype="http://www.w3.org/2001/XMLSchema#float">61.6</pg:flesch>
 <pg:fog rdf:datatype="http://www.w3.org/2001/XMLSchema#float">14.0</pg:fog>
 <pg:lix rdf:datatype="http://www.w3.org/2001/XMLSchema#float">47.4</pg:lix>
 <pg:smog rdf:datatype="http://www.w3.org/2001/XMLSchema#float">11.7</pg:smog>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abandoned"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abate"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#ability"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#able"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abode"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#about"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#above"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abroad"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#absent"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#absolute"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abstract"/>
  <pg:occurrence rdf:resource="http://skosaurus.rrecktek.com/ont/lexicon/7020/7020#abstractions"/>
```

APPENDIX D – checksanity.pl

```
#!/usr/bin/perl

###
### A program to validate RDF metacards
###
### Copyright 2006 Ronald P Reck

@files=`find -name '*.rdf'  -print`;

foreach $file (@files){
$filecount++;

chomp($file);
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
      $atime,$mtime,$ctime,$blksize,$blocks)
         = stat($file);

@answer=`rapper -c file:$file 2>>/tmp/deleteme.log`;
foreach $response (@answer) {
chomp($response);
# looks like it changed from statement to triples
#if ($response =~ /statement/) {
if ($response =~ /triples/) {

($undef,$undef,$undef,$count,$undef)=split(/\s+/,$response);
$total=($count+$total);

          }

if ($response =~ /Failed/) {
#rapper: Failed to parse URI file:13516-doc.doc.rdf rdfxml content
(@undef,$delete)=split(/:/,$response);
          ($delete,$undef,$undef)=split(/\s+/,$delete);
          &nuke($delete);
                    }

#print "\n$response";

#print "\n$filecount $total statements";

}
undef(@answer);
if (!defined $firsttime) {$firsttime=$mtime;print "setting firstime";};
$diff=($mtime-$firsttime);
$diff++;
```

```
$sps=($total/$diff);
$sps=int($sps);
print "\n$filecount $total statements: statements per second $sps for $diff seconds";
#print "\n$filecount $total statements :$ctime:$diff";
undef($diff);
}



sub nuke {
$delete=shift;
print "\n delete $delete";
exit 0;
}
```