R doc2vec Implementation

Final Project Report

Client: Eastman Chemical Company

Virginia Tech

Dr. Edward A. Fox

Blacksburg, VA 24061

CS4624 4/28/2017

Jake Clark Austin Cooke Steven Rolph Stephen Sherrard

Table of Contents

Table of Tables

Table of Figures

Executive Summary

Doc2Vec is a machine learning model to create a vector space whose elements are words from a grouping or several groupings of text. [3] By analyzing several documents, all of the words which occur in these documents are placed into the vector space. The distance between these vectors indicates how similar they are. Words which appear in similar contexts have a small distance between them in this vector space. This algorithm has been used by researchers for document analysis, primarily using the Gensim Python library. [11]

Our client, Eastman Chemical Corporation, would like to use this approach, when working in a language more suited to their business model. [4] A lot of their software is statistical, written in R. Thus, our job had the following components: become familiar with Doc2vec and R, develop Rdoc2vec, and apply it to parse documents, create a vector space, and make tests.

First, to become familiar with the language, we spent a few weeks with tutorials including the Lynda library, which was provided by Virginia Tech. [13] After we felt we were familiar with the language, we learned about two of the dominant algorithms used, called Distributed Bag-of-words (DBOW) and Distributed Memory (DM). [3] After learning these two algorithms, we felt that we were prepared to begin development.

Second, we developed a class structure similar to that of Gensim. [11] Keeping this as a skeleton, we developed a parsing algorithm which would be used to train the model. The parser analyzes the documents and computes a frequency for the occurrence of each word. The parser itself takes a list of physical documents stored on the system and completes the analysis, passing the frequency of words along the pipeline.

The next step was to create a neural network for training the model. We elected to use the built-in neural network library written in R called nnet. [18] A neural network takes an initial input vector as a parameter. For our purposes, it made sense to use a “one-hot” vector, which has only one input. [8][14] This can cut down on later calculation because the input vector is only of size one. Then this input is multiplied by several weights to be put into a hidden layer, handled by the nnet library. The values in the hidden layer are multiplied again by several weights to go into the output layer. [14] After creating functions which called the nnet library, we began work on testing. In the meantime, we decided to begin a design on our own implementation of a neural network. By creating a neural network anew, we get around the major problem with nnet, which is optimization. Since nnet is a black box that we cannot affect, we cannot be sure that it is optimized for our application. Since we use “one-hot” vectors, which are not a default application, it is likely that there is some way we can improve the speed in our library. We were not able to finish and test our neural net, so it is something left for future groups to work on.

Finally, we began testing. We created a Web scraper which grabbed a number of articles from Wikipedia. [29] We used this scraper to get a number of different documents. Specifically, we scraped information on the congressional districts of several states. This gave us document sets which can be quite large when using several states, or smaller by analyzing individual states. We performed tests on these datasets, the results of which we kept with our code.

Userôs Manual

This section includes a how-to on the operation of r_doc2vec, specifically in the RStudio Platform. [12]

Scraping from Wikipedia
We have provided a simple Python scraper that will pull the fulltext of a list of Wikipedia pages detailed by newline separated pageIDs (see congressList.txt for an example listFile). The script is executed as follows:

```
python getDocs2.py <Filename of list>
```

Running example_script.R
Begin by modifying the example_script.R file's path variable by adding the path of the directory housing the sample documents:

```
path <- "C:/Users/RJake/r_doc2vec/proj/R/other" # Change this to the directory of your
documents
```

Now, in example_script.R modify the vector_size variable to determine how many dimensions the vector comparison of documents will take into consideration. The most relevant n words for the samples presented will be used; make sure to choose this number wisely as it will severely impact the quality of your results. The default is 20 words. Selecting a higher value will result in more precise vectors, but will also drastically extend the amount of time for the comparisons to be completed.

```
vector_size <- 20 # produce vectors of this fixed size
```

Now set the present working directory in RStudio to the location of example_script.R. In RStudio, this is done by typing the following command into the console:

```
Setpwd('Directory containing example_script.R')
```

Now that the environment has been configured, run example_script.R. In RStudio, this is done using the following command:

```
Source('example_script.R')
```

The results will be saved in a file called results.csv; a sample can be seen in ***Figure 10.***

Developerꞌs Manual

This section will discuss each file in our Github repository provided by Eastman. [4] Here we will discuss the work that has been completed as well as the work left to be done for this project.

Introduction

We have split our development of the Distributed Bag of Words algorithm [3] into three stages. The first uses a neural network written in R called nnet. [18] In this stage the focus was getting the algorithm to a working state. In the second stage, we wanted to implement the neural network so that we could begin optimizing the training phase with techniques documented in Le and Mikolov’s paper. [3]. This implementation is based on a series of matrix multiplications and softmax calculations described by Richard Berendsen [15]. Once we had an optimized implementation, we wanted to start working on a version similar to gensim created by Radim Řehůřek [11] which implements this algorithm in Python. Once these phases were complete we would begin developing the Distributed Memory algorithm. [3] Listed below are the files located in our Github.

r_doc2vec/proj/R/doc2vec.R

This file stores the class definitions that we will need to implement the gensim model [11] in R. Other classes that are needed for implementing the gensim model should go into this file. For the end goal, the user should be able to create a doc2vec object and run the model on a set of tagged documents to provide vectors for each document, similar to how gensim works. [11]

r_doc2vec/proj/R/example_script.R

This file serves as a tutorial for how each function should be used to produce document vectors. In its current state, it details how to use the nnet library [18] implementation which is our first phase of implementing the DBOW algorithm. As further phases become implemented, this document should be updated to reflect those changes. For instance, when the second phase using matrix multiplication and implementing the neural network are completed, this file should source the proper files and function calls should be changed to the new ones. This file also prints the cosine similarity between each document, which could be changed once more work has been done on post doc2vec analysis. This includes such functionality as finding the k most similar documents or least similar documents since for a large corpus of n documents, printing every similarity produces output on the order of n^2.

r_doc2vec/proj/R/io.R

The purpose of this file is to provide functions to print user friendly versions of document vectors, the vocabulary, and any other information that a user could want to see that isn’t printed to a file. For saving the results of the doc2vec algorithm, see r_doc2vec/proj/R/saver.R. Currently we don’t have any functions implemented in this file, however it will be important to implement helpful printing functions as there may be users who run our code through the R command line who want a way to see what vectors have been produced and what files they represent. The end goal for this file is to facilitate complete transparency of data at each step of the algorithm. Additionally, this file should contain functions for plotting and displaying the results of Rdoc2vec. We recommend using R’s built in plotting functions in conjunction with the ‘Rtsne’ package[22] which allows for low dimensional embedding of high-dimensional data sets.

r_doc2vec/proj/R/neuralnetwork.R

This file stores our neural network implementation in the second phase. The network we are trying to implement is based off of a prior implementation of DBOW created by Richard Berendsen. [15] In this file we hope to create our own implementation of the network so that we may begin to optimize the network for our algorithm, including some mentioned in the original doc2vec paper [3]. Currently we have document code which implements the neural network. The next step for this file is to connect our neural network preparation functions from r_doc2vec/proj/R/neuralnnet.R to this network and to begin running our test scripts on this implementation. The end goal is to be able to run the DBOW algorithm faster than the implementation in the first phase using the nnet library [18] by adding several algorithm specific optimizations such as negative sampling. [3]

r_doc2vec/proj/R/neuralnetworktest.R
This file contains a test script to determine the correctness of our new neural network implementation in the second phase of developing the DBOW algorithm. As a result, it shows how the network should be used. This file can be improved since it currently isn't in a finished state. There is work to be done which can test different learning rates and other parameters that the neural network uses.

r_doc2vec/proj/R/neuralnnet.R
This file contains several methods to prepare the input to the neural network and contains the code to calculate the vectors using the nnet neural network implementation. [18] This file holds code that grabs a context from the document, samples that context, and then adds the sampled word to the list of examples for the network. [3] It also holds the nnet neural network implementation. This file should be modified for future phases where different implementations of the neural network are created. All that should be changed to connect any new neural networks is the run_batch function as this function should act as an interface to the network. If multiple methods are used to calculate the document vectors, the calc_vector function should be modified to either specify which network to use, or multiple calc_vector functions should be made and differentiated to reflect which network they use.

r_doc2vec/proj/R/parser.R
The purpose of this file is to provide two functions. The first is to parse a series of documents in a given directory; the second is to build a shared vocabulary between all of the documents parsed. In its current state, the parse function can be set to recursively descend into directories, use a list of stop words, and can discard infrequent words to reduce the size of the vocabulary. [3][18]. Future modifications can be made to this file with regard to parsing through several means. The first is that the parse function could be broken up to read one file at a time so that the process could be parallelized. Another possible modification could be to add a parse function that can read across multiple directories on different paths so that the user can specify which folders hold documents to parse so that they want to organize their documents how they choose. Finally, another modification could be a way to allow the user to append more documents to the list produced by the parse function, so that the user can run the doc2vec algorithm and then add more documents to the corpus later without having to re-parse all of the documents.
In its current state, the build_vocab function gathers all of the documents and combines the vocabularies of each document into one vocabulary. This function could be modified to work similarly to the parse function, where each word has a minimum frequency that has to be met.

r_doc2vec/proj/R/saver.R
The purpose of this file is to provide a way to save results of the doc2vec algorithm to a CSV file. This file contains one function, which takes as parameters a filename, an R dataframe data object, and an optional parameter that allows the data to be appended to an existing CSV file. The function uses the R ‘readr’ library [26] to implement a faster saving [27] method than is available through R’s base functions (write.csv, read.csv). The optional ‘append’ parameter is set to FALSE on default, so if not specified the function will overwrite / create a new CSV file for the data being saved. In the current version, this function also returns the data that was saved. Currently, if data is appended into an existing file, all of the data is returned (old data, in addition to newly appended data). In the case where no appending is done, the data passed to the function is returned in the case that the programmer wishes to reuse it for any purpose. In the future, this function could be expanded to accept multiple file types (handle the cases where the input is not a dataframe) and to save in different file formats depending on parameter values.

r_doc2vec/proj/R/word2vec.R
Similar to the doc2vec.R file, this file stores the class definitions for the word2vec algorithms implemented by gensim. [11] This file should be updated once the Distributed Memory algorithm is being worked on as DM requires word2vec in order to calculate the document vectors. [3] Methods which implement any word2vec implementation should go into this file, with the end goal that the word2vec from gensim is implemented in this file. Ideally, both skip-gram and continuous bag of words should be implemented, however only one is needed, so one of the algorithms should be chosen and implemented so that DM can be implemented first. [3][8]

Lessons Learned

In this section we will detail some of the issues we encountered and some of the solutions we came up with to solve these problems.

Timeline/Schedule

When we were initially planning the project, we wanted to write both algorithms associated with doc2vec: Distributed Bag of Words and Distributed Memory. [3] In order to implement Distributed Memory, we needed to also code word2vec since Distributed Memory takes a context of words and attempts to determine what word should go next. [3] Our original timeline required us to write the word2vec implementation in early March so that we could finish both doc2vec algorithms by March 27. It quickly became apparent that we needed to scale back our timeline. We conversed with our client and we agreed on implementing the DBOW algorithm first since we wouldn't need to implement word2vec to complete it, and then work on word2vec and the Distributed Memory algorithm with our left over time.

Research

One of the problems we had initially was different ways that the team went about researching the problem. Early in development, there was a difference of opinion between team members on whether the project should be built directly using a neural network, or if we should do a direct port of the Gensim library into R. This lead the two parts of the team to use a lot of time researching which option should be best when it may have been a better decision overall to make the decision early. Eventually, we made the decision to use the nnet library, because we felt that it would be an easier and quicker build overall, and could then work on a more direct port. Directly porting turned out to be a very ambitious task, and at this state in the project, it would not make sense to remove existing code to move backwards. Thus, we wasted time by not making a decision quickly, and instead falling back on research which did not really change the final decision.

Testing

In order to perform testing, we needed an implementation that worked correctly so that we could accurately report results. However, many of our first tests would not complete running. They seemed to run for such a long time that they would not complete. This did not allow us to determine any problems because we could not see where the problem in the code might be with such a long runtime. We got around this by using smaller and smaller testing datasets in order to determine the source of the problem, and were soon finally able to complete a test.

Ill-defined goals

From the beginning, this project was not about creating a fully-working, open-source library completely in one semester, since that is simply too big of a project for that timeframe. The goal of the project became instead to make a product that our client can try out and play with, to determine if the project is worthwhile to devote a lot of time to. By having the project not specifically laid out, at times we wasted time trying to fit the first, unrealistic goal. For example, the class structure was made with the idea to follow Gensim's model as closely as possible for the average user. This meant, however, that our finished skeleton build of doc2vec does not use many parts of the original functionality put into this initial class structure. Thus, we should have been more aware of our goals from the beginning, and defined a more specific metric of success.

Future Work
Several major features still need to be completed. First, we need the second and third phases of the DBOW implementation to be completed; see Developer's Manual. Afterwards we need an implementation of word2vec to create the Distributed Memory algorithm. This will require an implementation of Skip-Gram [8] or an implementation of continuous bag of words [3]. Once word2vec is completed, the Distributed Memory algorithm can be implemented. The neural networks used for the DBOW algorithm can be reused for this algorithm with slightly different input and output to the neural network [3]. Finally, we would like a way to display the document vectors we find in a plot using Rtsne [22] which is an R package that implements the t-SNE algorithm so that users can easily see how similar two document are [28].

Acknowledgements

Dr. Edward Fox
CS 4624 Professor
fox@vt.edu

Don Sanderson
Service Manager, Marketing Solutions
Eastman Chemical Company
donsanderson@eastman.com

Adam Spannbauer
R Programmer and Data Scientist
Eastman Chemical Company
adamspannbauer@eastman.com

References

1. Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition. Pearson Education, Inc., Upper Saddle River, 2005. Retrieved February 19, 2017, from Virginia Tech Library: http://proquest.safaribooksonline.com.ezproxy.lib.vt.edu/book/software-engineering-and-development/uml/0131489062.
2. Berezin, T. Writing a Requirements Document, 1999. Retrieved March 16, 2017, from Adelphi University: http://home.adelphi.edu/~siegfried/cs480/ReqsDoc.pdf
3. Le, Q. and Mikolov, T. Distributed Representations of Sentences and Documents. (May 2014). Retrieved March 16, 2017 from https://arxiv.org/pdf/1405.4053v2.pdf
4. Eastman Chemical Company Website, 2017. Retrieved March 16, 2017 from http://www.eastman.com/Pages/Home.aspx
5. yseam. Wikipedia in Python, Gephi, and Neo4j: Vizualizing relationships in Wikipedia. (December 2014). Retrieved March 16, 2017 from http://www.yseam.com/blog/WP.html
6. Keydana, S. Sentiment Analysis of Movie Reviews(3): doc2vec. (October 2016). Retrieved March 16, 2017 from Data Science Central: http://www.datasciencecentral.com/profiles/blogs/sentiment-analysis-of-movie-reviews-2-doc2vec
7. Wikipedia. Word2vec. (February 2017). Retrieved March 16, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Word2vec
8. McCormick, C. Word2Vec Tutorial - The Skip-Gram Model (April 2016). Retrieved March 16, 2017, from http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
9. Li, I. NLP 05: From Word2vec to Doc2vec: a simple example with Gensim (July 2016). Retrieved March 16, 2017, from https://ireneli.eu/2016/07/27/nlp-05-from-word2vec-to-doc2vec-a-simple-example-with-gensim/
10. Řehůřek, R. gensim: Topic Modelling For Humans (2009). Retrieved March 17, 2017, from https://radimrehurek.com/gensim/
11. Allaire J. Allen J. Atkins A. Avalos U. Bowsher P. Brennan J. Carney B. Carome A. Çetinkaya-Rundel M. Chang W. Chaudhari C. Cheng J. Clemens J. Fazzio S. Gadrow A. Garza R. Grolemund G. Hadley D. Henry L. Hester J. Karimeddini F. Kawaf T. Kipp A. Knast P. Lopp S. Luraschi J. McPherson J. Nolen S. Oberg R. Pylvainen I. Regenstein J. Borges Ribeiro B. Rickert J. Ritchie G. Shaver R. Ruiz E. Spiewak J. Stephens N. Ushey K. Walker C. Wickham H. Xie Y. and Yoder J. RStudio (2016). Retrieved March 16, 2017, from RStudio: https://www.rstudio.com/. This program serves as our IDE.
12. Niemann-Ross, M. Code Clinic: R. (April 2015). Retrieved March 17, 2017 from Lynda: https://www.lynda.com/R-tutorials/Using-exercise-files/372541/384659-4.html?srchtrk=index%3a9%0alinktypeid%3a2%0aq%3aR%0apage%3a1%0as%3arelevance%0asa%3atrue%0aproducttypeid%3a2
13. Franceschet, M. Betweenness Centrality. Retrieved March 17, 2017 from https://www.sci.unich.it/~francesc/teaching/network/betweeness.html
14. Berendsen, R. Implementing doc2vec. (January 2017). Retrieved April 10, 2017, from https://amsterdam.luminis.eu/2017/01/30/implementing-doc2vec/
15. Wikipedia. Cat. (April 2017). Retrieved April 10, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Cat

16. Wikipedia. Dog. (April 2017). Retrieved April 10, 2017 from Wikipedia: https://en.wikipedia.org/wiki/Dog
17. Ripley, B. and Venables, W. Package ‘nnet’. (February 2016). Retrieved April 10, 2017, from https://cran.r-project.org/web/packages/nnet/nnet.pdf
18. Google. Google’s R Style Guide. Retrieved April 11, 2017, from Google: https://google.github.io/styleguide/Rguide.xml
19. GitHub, Inc. Git - Documentation. Retrieved April 11, 2017, from Git: https://git-scm.com/doc
20. Donaldson, J. Package ‘tsne’. (July 2016). Retrieved April 11, 2017, from https://cran.r-project.org/web/packages/tsne/tsne.pdf
21. Krijthe, J. and van der Maaten, L. Package ‘Rtsne’ (August 2016). Retrieved April 11, 2017 from https://cran.r-project.org/web/packages/Rtsne/Rtsne.pdf
22. R Core Team. Formal Methods and Classes. Retrieved April 11, 2017, from https://stat.ethz.ch/R-manual/R-devel/library/methods/html/methods-package.html
23. Bates, D. Chambers, J. Dalgaard, P. Gentleman, R. Hornik, K. Ihaka, R. Kalibera, T. Lawrence, M. Leisch, F. Ligges, U. Lumley, T. Maechler, M. Morgan, M. Murdoch, D. Murrell, P. Plummer, M. Ripley, B. Sarkar, D. Temple Lang, D. Tierney, L. Urbanek, S. Schwarte, H. Masarotto, G. Iacus, S. and Falcon, S. What is R? Retrieved April 11, 2017 from R-project: https://www.r-project.org/about.html
24. The MathWorks, Inc. Matlab.(2017) Retrieved April 11, 2017, from The MathWorks, Inc: https://www.mathworks.com/products/matlab.html
25. Wickham, H. Package ‘readr’. (March 2017). Retrieved April 24, 2017 from https://cran.r-project.org/web/packages/readr/readr.pdf
26. STHDA. Fast Writing of Data From R to txt|csv Files: readr package. Retrieved April 24, 2017, from STHDA: http://www.sthda.com/english/wiki/fast-writing-of-data-from-r-to-txt-csv-files-readr-packag e
27. van der Maaten, L. and Hinton G. Visualizing Data using t-SNE. (November 2008) Retrieved April 28, 2017, from the Journal of Machine Learning Research: http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf
28. Goldsmith, J. wikipedia 1.4.0: Wikipedia API for Python. (November 2014) Retrieved April 28, 2017, from https://pypi.python.org/pypi/wikipedia/

Appendices

1.0 Introduction

This section will serve as an overview seeking to explain the purpose of the Rdoc2vec project. It also serves as an introduction to the project as a whole.

1.1 What is Rdoc2vec?

R is an open source statistical modelling language that provides statistics techniques and plotting. [24] It is similar to Matlab in that it is mainly used for specific applications. Where Matlab may solve a large dataset, R would be used to instead analyze that dataset. [25] Doc2vec is an algorithm which takes a set of documents and creates a vector space which can represent each document as a vector in that space. [3] It is based on word2vec, which similarly maps words into a vector space. By putting these words in a vector space, their distance can be considered, hence their similarity - that is, vectors which are close together represent similar words while vectors that are far apart are for very different words. doc2vec differs from word2vec in that varying amounts of text can be used as input. Where word2vec analyzes specific words, doc2vec takes large portions of text, sometimes called documents, sometimes called paragraphs. The vectors created from doc2vec are analyzed in a similar way as well. Importantly, the vectors are placed in a multi-dimensional vector space, which is usually much larger than 3 dimensions. [3] By using R, the data can be more easily visualized in either a 3-dimensional or even a 2-dimensional space. Thus, Rdoc2vec is doc2vec implemented in R. By being written in a statistical language, data visualization will come much more naturally.

As will be explained in ***Section 3***, Rdoc2vec uses two main models of computation. First, is the distributed bag-of-words model. [3] When using distributed bag-of-words, Rdoc2vec will first invoke the word2vec algorithm called continuous bag-of-words. This algorithm takes a set of words, without order, and tracks what other words around it are like, in order to determine word similarity. After this first training process, distributed bag-of-words will work the same way, but on a larger scale. It will analyze the earlier created word vectors in order to create paragraph vectors which themselves will be grouped together. [11]

The second model is distributed memory, or Skip-gram. [3][8] Skip-gram is trained to predict words within a given context. Skip-gram's underlying algorithm uses word pairs from a sample text to train a neural network that can be used to identify the semantic meaning of words in a number of "features" represented in a vector. [8]

2.0 Requirements
This section will detail the project requirements of Rdoc2vec as outlined by Eastman and decided upon by our team. Here we will discuss the program's intended users, the sort of data being processed, the environment in which it will be deployed, and a number of functional, supportability, performance, and interface requirements.

2.1 Who will use Rdoc2vec and why?

Rdoc2vec will be used by the client organization, Eastman Chemical Company. According to their website [4], Eastman is "a global specialty chemical company that produces a broad range of advanced materials, additives and functional products, specialty chemicals, and fibers that are found in products people use every day." [4] While they are not known for their software work, the client tells us that all engineers use R for their tasks every day. Statistical analysis is required on all of their products, so there are several R libraries which they use. For Rdoc2vec specifically, Eastman wants to be able to use the software to analyze large documents that they have in their possession for trends. They also want the other major benefit of releasing the project - open source recognition.

Rdoc2vec will be released to the R community as open source software. By doing so, Eastman can attach their name to a potentially very successful library, attracting new employees in the process (who may not initially be interested in a chemical company rather than a software one). Of course, this means that R's open source community will also use the library. Historically, doc2vec has been used to analyze large swaths of data. In one paper [3] published by Google, they used doc2vec on the IMDB database, to see if doc2vec itself measured up against other machine learning algorithms. [3] This study showed doc2vec's ability to analyze, because it was comparable or better than the other methods used. While this was simply a proof-of-concept study, there are many other applications for doc2vec. An organization called yseam [5], in their analysis on relationships between Wikipedia articles, were able to use doc2vec to find a strong relationship between PageRank of a given Wikipedia article and its betweenness centrality. [14] That is, they were able to show that Wikipedia articles with a similar PageRank could be easily grouped by their subject matter and similar sounding text. [5] Thus doc2vec showed an underlying relationship which had been unobserved.

With an implementation in the language R, more studies can be conducted into the doc2vec algorithm's effectiveness. R is a language created by and for statisticians, and doc2vec naturally lends itself to representation in data visualization. We believe that these studies will be the major use case of Rdoc2vec.

2.2 Data involved in Rdoc2vec

The data used in Rdoc2vec will be documents containing words. Rdoc2vec will break down these documents by paragraphs and then by sentences and finally words in order to create a vector for the entire document. These paragraphs are used to get paragraph vectors and then word vectors. These vectors are then used to predict new paragraph vectors for paragraphs that have not been seen before.

When finished, Rdoc2vec will produce a fixed length vector from the input documents. This vector can be used for many machine learning algorithms.

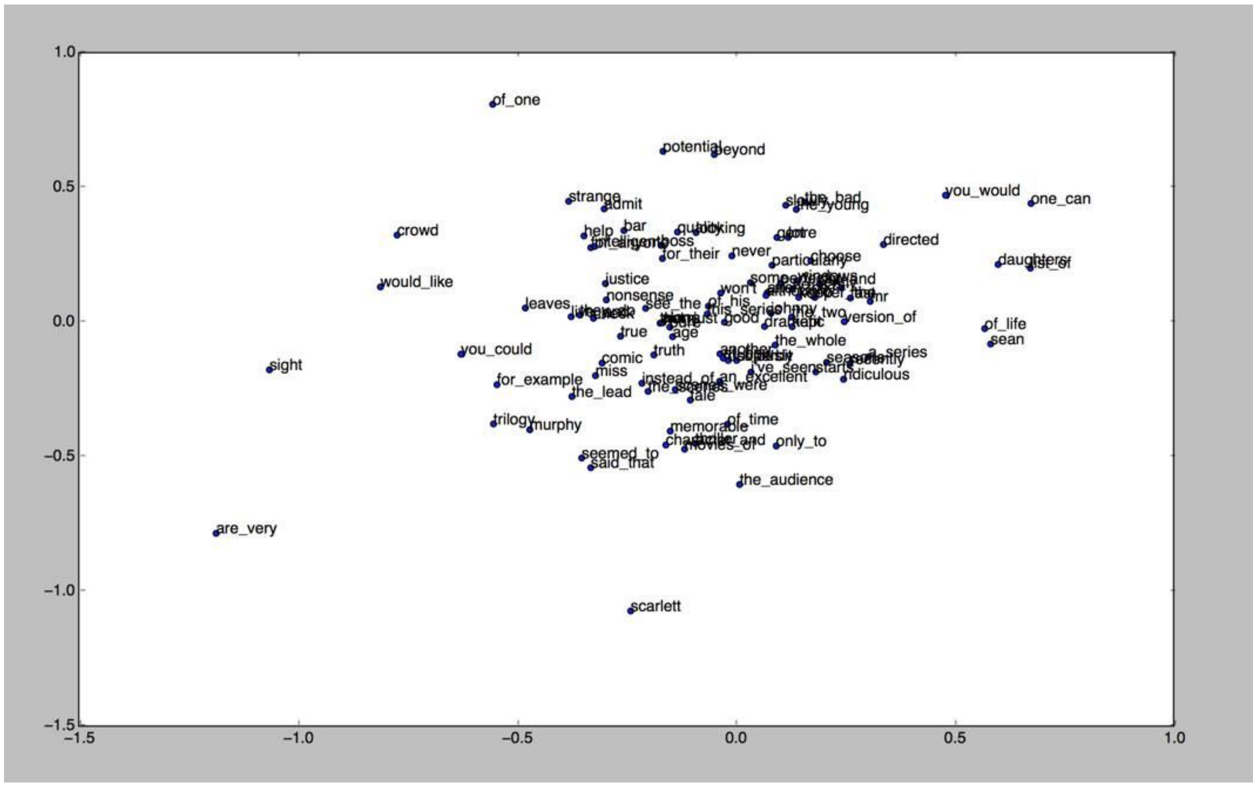


Figure 1 -- Sample plotting of vectors from doc2vec [9]

When finished, it is possible to plot the words from the document (***Figure 1***). The words that are plotted closer together are deemed to be more similar than those that are far apart.

2.3 Environments which will run Rdoc2vec

Rdoc2vec will be used to run an analysis on a set of documents that has yet to be provided; the content and type of these documents is yet unknown.

R is interacted with through a command line interface; it is available on all major operating systems. For the purposes of this project, we will be developing in RStudio [12] which standardizes the interface between operating systems. We have speculated that it will likely be used on both Windows and Linux machines.

One possibility is that our solution will be deployed on a web server running RStudio. This seems likely as a large amount of computing is required by Rdoc2vec, particularly when comparing the resulting vectors to a large database of documents. It is unlikely that individual user's machines will have the resources necessary to complete this sort of task in a timely manner.

2.4 Functional Requirements

Rdoc2vec should take variable length segments of texts including sentences, paragraphs, and whole documents as input and process these into vectors. The Distributed Bag-of-Words and

Distributed Memory algorithms will be used to achieve this goal. [3] Distributed Bag-of-Words will take a document as input, and sample words in the document to form a classifier. Distributed Memory takes a paragraph and a series of words to predict the next word in the sequence. This requirement is important and is program critical since variable length documents will be used for input. [3]

Rdoc2vec should be able to specify which algorithm to use. Rdoc2vec should be able to specify a directory to read documents from. This will give Rdoc2vec the capability to conform to the user's use case instead of having the user conform to Rdoc2vec. This requirement is important, but the ability to specify algorithms is not necessarily important since separate files could be kept for both algorithms, however this will affect maintainability and usability. The ability to specify a directory is critical since having a hard coded file path will not work for all users and use cases.

Rdoc2vec should be able to store and read vectors that have already been calculated so that they can be further trained or processed with new documents. The requirement to store these vectors is important so that they can be used for further calculation and processing, however the ability to read these vectors for more processing is secondary to the above requirements and will serve as an additional capability after functionality has been completed.

2.5 Performance Requirements

In terms of performance, there are two metrics to cover: accuracy and speed. In both, we anticipate to make Rdoc2vec comparable to other implementations of doc2vec. There are implementations of doc2vec in Python and C++, and we believe that using these implementations as a comparison benchmark will be a good metric for the library.

An analysis using the gensim [11] model (gensim created the first doc2vec) by the group Data Science Central, found that doc2vec's distributed memory model had a 0.1% decrease in errors compared to word2vec's distributed bag-of-words model. [6] Also in the paper by Google discussing doc2vec, it was found that there was as low as a 9.4% error on document sets in the IMDB database [3]. The doc2vec implementation by gensim [11] is written in C++, and we hope to reach a similar error rate with the R implementation of doc2vec. This would mean that our implementation needs to be able to reproduce an accuracy of as high as 90%. This may be a lofty goal, but we are so far optimistic.

Different implementations and uses of doc2vec have varied run-times. Some metrics place as large an amount of time as 30 hours in order to train the algorithm on a certain large data set. Of course, this depends on two things: the size of the data set, and the speed of the machine. When running doc2vec on our personal machines, we do not expect a quick run - our laptops are not made for such large calculations and will likely not be reliable in a large test. For benchmarking, we expect to find a way to compare our usage by running a training algorithm on a proprietary server (likely provided by our client, Eastman). With that server we will be able to benchmark. We hope to have a comparable speed to gensim's C++ implementation, or at the very least be able to beat the time limit of 30 hours for one data set.

2.6 Supportability Requirements

Our solution is open source and will be available on the project Github page. Each component of our pipeline should be sufficiently modular that it can be modified without affecting the other components of the system drastically.

We will also maintain a project wiki for our own purposes and to help future developers understand and support our final product. This wiki will detail the specific function of each of the project's components as well as the reasoning that went into our design decisions.

2.7 Interface Requirements

Our main constraint with interfacing will be the size of the chosen input. If the input size is too large, running Rdoc2vec on our local machines may prove impractical due to long runtimes caused by insufficient hardware. In this case, running the program on a server or industrial computer would be necessary.

Our program will be compatible with any system that has the R language installed.

3.0 Design

This section will detail the planned design of the Rdoc2vec project. Here we will discuss our chosen design pattern and explore doc2vec's underlying algorithms with the help of a series of diagrams.

3.1 Design Pattern

Since Rdoc2vec is primarily a processing application, we will use a Pipe and Filter approach. We will need a utility to take as input a series of documents and parse these documents for relevant context. This utility would pipe to a training function which will take all of these contexts and map them to a vector space. This output can be further filtered to find similarity between documents via another utility. Since doc2vec has two different associated algorithms, namely Distributed Bag-of-Words and Distributed Memory [3], we will need several utility functions to handle input to be processed.

Using this design pattern will allow Rdoc2vec to process text independently from training which will help in maintainability and testability. By having smaller functions with more specific roles, there is a better chance that errors can be isolated and fixed without affecting functionality elsewhere in the program. This also helps testability by providing more modular functions which can be individually tested for correctness and accuracy.

We will also need an extra layer to interface with the user since we will need to know which algorithm to use as well as which files to use. This will likely include a series of options or flags which can be used to specify what the user wants. We will also want a post-processing layer which will format the output from the training function for storage to be used later, or to display to the user. These layers will help facilitate interactions between the user and the program.

3.2 Design of Rdoc2vec

Our project will be using R to create the doc2vec functionality. During the implementation phase, we will be utilizing R version 3.3.2 named "Sincere Pumpkin Patch". Previous implementations of these algorithms exist in Python, Java, and other languages, however our client requires a fully R implementation. We will be using base R capabilities throughout the project so that as much as possible of Rdoc2vec is written in R. We will be using RStudio, an open source IDE, to develop Rdoc2vec, as well as other graphical consoles for R.

As mentioned in the Requirements Analysis, see Section 3.1, we will use a series of pipes and filters to route information to the proper site for processing. A sample data flow can be seen in ***Figure 2***.

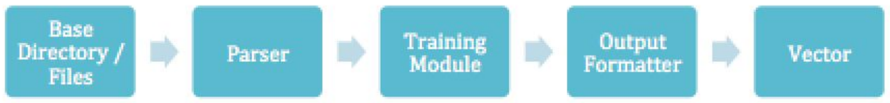


Figure 2 -- Sample data flow

In this case, we start with a base directory full of files which we would like to vectorize. Rdoc2vec will use a parser to read each file to create a pre-processing data set. For example, in

Distributed Bag-of-Words, each file will be read and sampled for a classifier. The training module takes data that has been properly formatted from the files through the parser. Then the resulting output from the training module is sent to another module which will apply formatting for displaying to the user or for saving to a file. The ultimate output is the desired vector which can be used for further calculations.

Currently, Rdoc2vec will serve as a command-line style application since no graphical interface is required. The program will be available as a package that can be imported for use in other applications.

3.3 Design Diagrams

A visual representation of word2vec can be seen in ***Figure 3***. On the left is a set of word vectors which are shown in multiple dimensions. The black dots are words which are similar and should be marked as such, while the white dots are other words, dissimilar to the black dots. The red line is a function that separates the similar words from the dissimilar words. It is clear that on the left, the words are not grouped very closely together. On the right is the same diagram but represented using some data visualization. Since the vectors are in a multi-dimensional space, many representations can show if words are close or similar to one another. This is why R is useful since it has so many ways to visualize data.

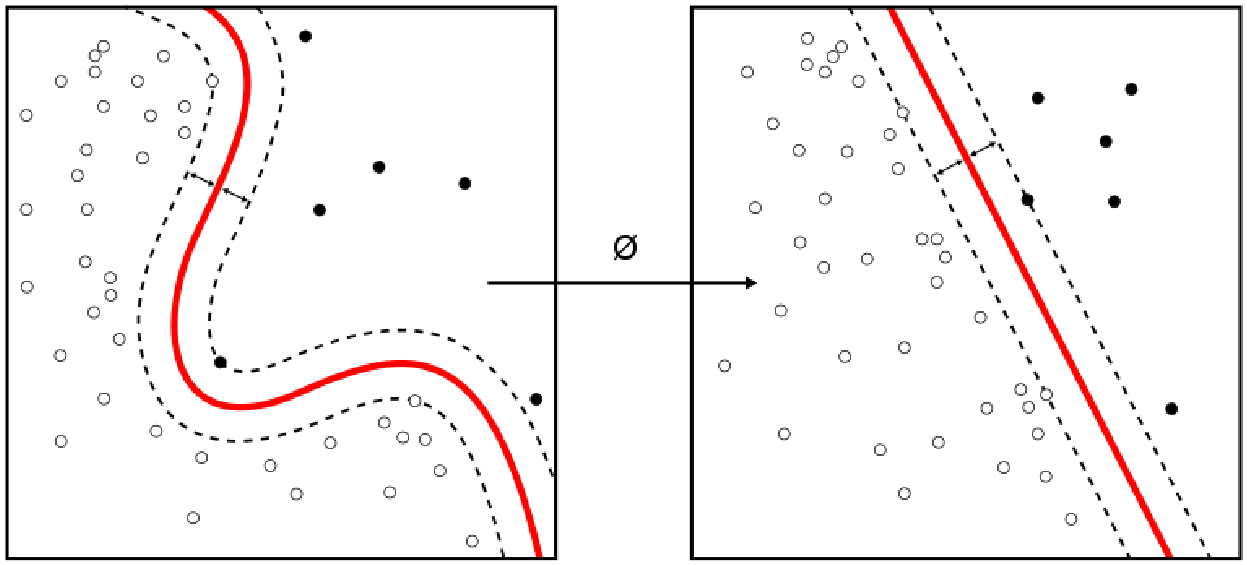


Figure 3 -- Visual representation of word2vec [7]

Figure 4 is from a paper on doc2vec [3]. It is another description of word2vec, specifically the continuous bag-of-words algorithm. The word matrix takes several groups of words which are grouped together. By averaging (or concatenating) which words appear next to which other words, a guess can be made at a word which connects all 3 of them.

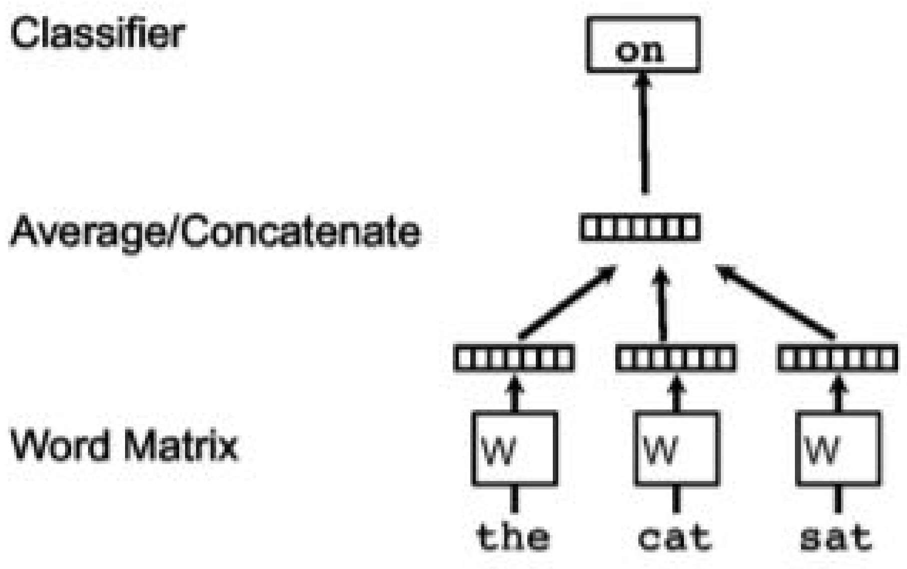


Figure 4 -- Visual representation of continuous bag-of-words [3]

Figure 5 looks similar to ***Figure 4***, but is instead a description of doc2vec and its algorithm, distributed memory. The word matrices are replaced by paragraph matrices. Each paragraph matrix is itself an iteration of word2vec on a large set of words. Then, these paragraphs are again averaged or concatenated in order to find a single classifier for any given word or set of text.

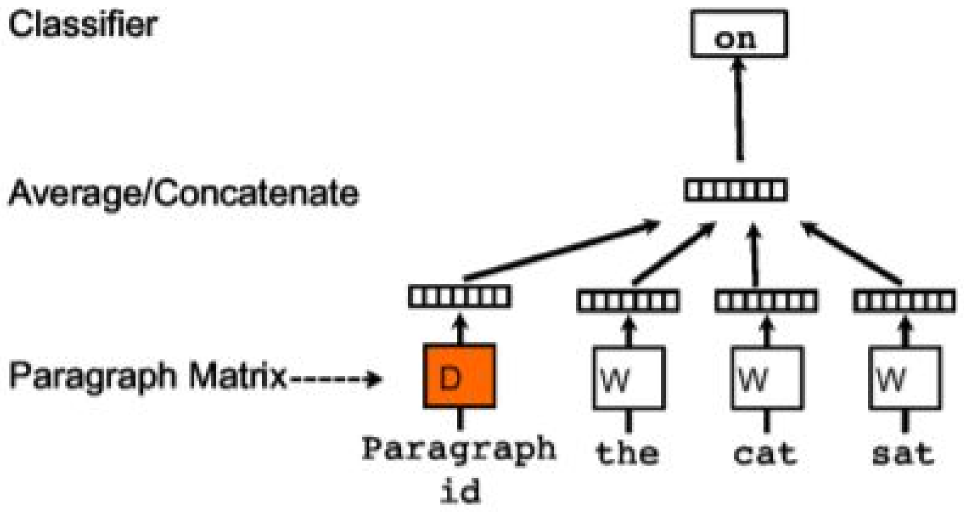


Figure 5 -- Visual representation of distributed memory[3]

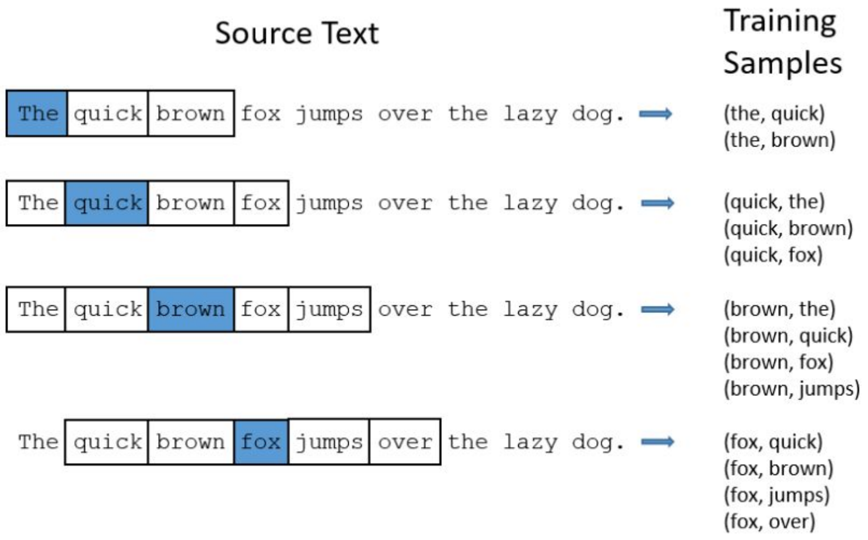


Figure 6 -- Skip-gram word selection process [8]

Figure 6 depicts the Skip-gram algorithm's word pair selection process. The word pairs seen on the right are used to assess the features of each word. Based upon the data collected from the training sample, Skip-gram can be used to predict the surrounding words given a single word. The next diagram shows this process in an abstract sense.

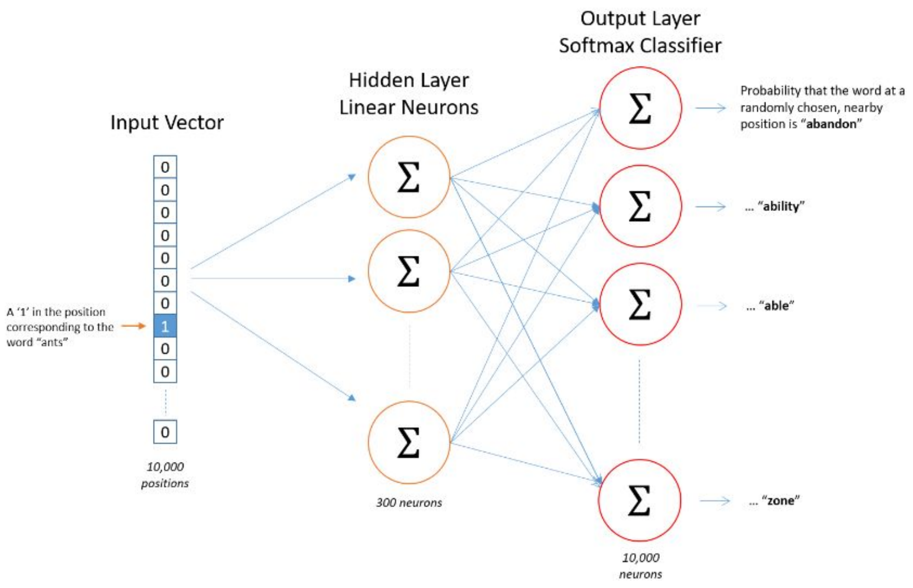


Figure 7 -- Visualization of the relationship between the layers of the skip gram model [8]

In the Skip-Gram model (***Figure 7)***, the input vector is used to select a specific word's features from the hidden layer; then based on those features, the probability of each other word in the dictionary being in the selected word's context is shown in the output layer. The sum of all the values in the output layer is 1. For the purposes of our project, we are interested in the values stored in the hidden layer as they can be used as vector representations of the semantic meaning of words.

3.4 Progress Report (2/19/2017)

In the first weeks of this project, we have been tasked to learn R. We have been given tutorials from such places as lynda.com [13]. We all feel familiar with R enough to begin working on the project. We also have many resources which we have thoroughly examined that specify how doc2vec works and should work in our implementation [3]. Finally, we also have access to the Github provided by Eastman where we will be contributing to the project.

According to our timeline, our next task to complete is learning the word2vec and doc2vec algorithms so that we can get started on implementation by March. We will first work on word2vec since it is the “small-scale” version of doc2vec. Once word2vec works, expanding to doc2vec should not be that difficult. We will also start working on base code and file creation for our project. That includes any boilerplate code which will be used for document input or output, or saving information, etc.

4.0 Implementation

This section will detail our current progress towards implementation as well as our projected timeline for the remaining tasks to be completed.

4.1 Progress Report (3/17/2017)

We currently have some boilerplate code written to facilitate future steps. These include class creation, file I/O, and parsing.

The project structure or class structure was an important design aspect. First, the code is designed from the ground-up using the R style guide published by Google. [19] There are a few important design aspects which have been followed. First, all function names which have assignment statements can only be separated in between statements, not in the middle of them, e.g.,

```
        Doc2vec <- function(document, dm = 0, alpha=0.1, size= 20,
                min_alpha=0.025)
```

Rather than

```
        Doc2vec <- function(document, dm = 0, alpha=0.1, size= 20, min_alpha=
                0.025)
```

Second, all variables follow the camelCase style, while function names are CapitalCamelCase. Finally, we will not be using curly braces in our code. This will make it read more Python-like, and is again supported by the Google Style Guide. [19] Another part of the project structure is the file layout, which follows the standard R project layout, as follows:

```
        proj/           # Contains all files
                R/      # Contains all R code, with .R filenames
                doc/    # Contains all algorithm documentation, including a copy of the README
                figs/   # Contains other algorithm description figures
                output/ # The output of the program will go here by default, but can be specified
```

We are in an early development process, so most of the information stored in these places is placeholder data. Within the R directory, there are several files which contain specific code: the parsing file, the saving file, the I/O file, the word2vec file, and the doc2vec file. Finally, all of this is stored at our Github page. Our code is written with this design process above in mind with the only caveat that another README is at the same level as the proj directory so that Github can display it easily. We follow the branch-code-test-merge design process of Github [20] for safety of the code.

One piece of code written during the boilerplate phase was a parser function which would be used to read all of the documents in a specified directory. This parser currently accepts as input basic text files. Each file read by the parser is processed by taking all of the words that appear in the text and counting the frequency of each word. The return value for this parser is a list of "documents" which consist of a document name and a list of word - frequency pairings. The word frequencies are important for building the vocabulary in later stages of the doc2vec algorithm. In its current state, the directory is searched recursively for files that it can read. Each word is lowercased so that words capitalized at the beginning of a sentence can be counted properly. This does cause an issue where some meaning can be stripped from the word, for instance if the text contains a name like Hill. Both Hill and hill will be counted as hill when the document is parsed. Each word is also stripped of punctuation in case it appears at the end of a sentence, or contains commas. This may also affect certain words, for example hill's and hills will be marked the same after parsing. This function requires the use of the methods-package [23] so that the setClass() function can be used. Potential future changes to this function include adding a parameter to determine whether to recurse to subdirectories when looking for documents, and removing common or stop words.

We have also begun work on printer functions to help visualize the data output. These functions will need to be customized to match Eastman's needs and the datasets we are given. At this time they will print out lists of words and their associated vectors.

4.2 Planned Work

Merge boilerplate code by 3/19
This stage of the implementation process will entail taking all of the code we have written to facilitate the actual algorithm such as class creation and file I/O, and integrating the code together. Every group member was tasked with a block of code to write, so every member will be involved with merging the code together since they will need to first submit their code, then review the code with the group. The end goals of this task are all of our base code committed to Github and creating a central R file which imports the functionality from the smaller files.

Write pseudocode for Distributed Bag of Words and implement by 3/26
This stage of implementation will require lots of planning to ensure the code is comprehensive. We all have a fundamental understanding of distributed bag-of-words, but need to diagram the code. Then we will write the code only after being sure that the pseudo-code is correct, as testing will be more difficult in the future. The end goal of this portion of the tasks is to have a working version (which is at least minimally tested) of DBOW, in both word2vec and doc2vec. For this stage, we need three group members working on pseudocode for the training method, focusing on Stochastic Gradient Descent and hierarchical softmax [3]. The extra member will write pseudocode for the similarity algorithm.

Setup for Training Vectors by 3/30
In this phase we are going to plan exactly how we will train/test the algorithm. In order to do so, we are first going to determine a database from which to gather data, and specifically which documents it is that we would like to test. We will prepare for running DBOW on these documents with both word2vec and doc2vec in order to analyze the difference and discover bugs. We will also determine exactly which data visualization tool we will be using to accomplish the task. Most likely this will involve a visit to the Visionarium for their tools to properly display our data. When this phase is finished, we will have a complete list of requirements for testing our doc2vec implementation. For this stage, we need two group members to determine how we can display the vectors, and two members to gather documents and samples to test.

Testing the Training Phase by 4/2
In the training phase, we will run our code on the data set we wish to analyze. In running the code we will ensure it completes without errors and returns the desired correct results. We will also, as mentioned above, use data visualization to see the results in a tangible form. Finally, we will compare the results of doc2vec to a word2vec run to ensure we are getting the proper speedup/more correct data. If there is more time in the training phase we will set up and train again repeatedly with different data, but this is simply a stretch goal. The end phase of this is to have at least one test phase complete which shows correctness of the algorithm. For this stage,

every member will test the algorithm using the samples that we found so that we may better benchmark the speed and accuracy of the algorithm. Once we test the training function, we will compare results to ensure consistency.

Similarity Analysis Phase by 4/6
In this phase of the algorithm, we will code the similarity analysis using the document vectors created from the Training Phase. This will include calculating the cosine distance between any two documents since document vectors that are closer in the vector space are more similar. [3] Then we will want to have a function which can sort these distances to find which documents are most similar to a particular document. We will have two members work on coding the distance between documents, and the other two members will take the data, sort by most similar, and then work on formatting output of these similarities.

Testing Phase by 4/9
In this phase we need to test our doc2vec algorithm using a series of test data in the form of plain text documents. First, we can test the algorithm's efficacy by seeing if documents we know to be similar are marked as similar since the algorithm should be able to find how similar any two documents are. Then we will compare our results with a fully implemented Doc2vec tool created by gensim which is also based on Distributed Representations of Sentences and Documents [3] which should provide a decent comparison. [9,11] As a group, we will run our algorithm and calculate the similarities, then compare these results with gensim.

4.3 Gantt Chart

Tasks	Date						
Merge boilerplate code	🟦	🟦					
Write pseudocode for DBOW and implement		🟦	🟦				
Setup for Training Vectors			🟦	🟦			
Testing the Training Phase				🟦	🟦		
Similarity Analysis Phase					🟦	🟦	
Testing Phase						🟦	🟦
	17	19	26	30	2	6	9
	March					April	

Table 1 -- A Gantt chart of planned work

5.0 Prototyping

This section will detail the effort to prototype doc2vec in R. Here we will discuss each component and its contribution to the doc2vec algorithm, as well as discuss how each component is used.

5.1 Parsing the Documents

The first step of the Distributed Bag of Words model is to be able to read documents and record a vocabulary paired with frequencies of each word. Using the read.delim() function in R, we are able to read in a text file as a matrix of words. We then flatten this matrix into an array where we can begin counting word frequencies and recording the words that appear in this text. Once we have a list of words, we start formatting the list. First we strip any punctuation from each word and then lowercase them as noted in Section 4.1. Then we start iterating over the list and adding to the vocabulary for this document, which our parse function does. During the prototyping phase we expanded the parse functionality to sort by frequency and return a list of document objects which store the name of the document and the vocabulary table of the document.

To demonstrate how this function works, first we shall set up a basic input file: test.txt.

test.txt contains the following words:

Test. This is a test. This tests for recording the frequencies of words in a document, and recording the list of words that appear in this text. Some of these words are repeated.

We will put this into a directory, assume that directory is called “/PATH/TO/TEST”. In R, we can create the desired list by calling parse(“/PATH/TO/TEST”).

The output generated is as follows:

```
[[1]]
An object of class "Document"
Slot "name":
[1] "TEST/test.txt"
Slot "words":
       word occurrence
1      test          2
2      this          3
3        is          1
4         a          2
5     tests          1
...
22 repeated          1
```

As seen by the output, we receive a document object which stores word - frequency pairings in the document. Also note that the words test and tests were not stored under the same word. This particular component of the DBOW algorithm will facilitate the next part of the algorithm, which is to build a shared vocabulary across all of the texts.

5.2 Building a Shared Vocabulary

This next step takes the list of documents and combines the vocabularies of each document into one vocabulary that we will use during the neural network phase of the algorithm. In essence, this step is an extension of the previous parsing component since it directly uses the output of parse() as input.

To demonstrate functionality of this function, we have two texts taken from Wikipedia. The first is from the article Cat, the second from the article Dog. [16][17] Shown below is the build_vocab() function.

```
document_list <- parse(“TEST/”)
build_vocab(document_list)
```

This gives us the output:

```
        word freq
1        the   31
2   domestic    3
3        cat    6
4         is    6
5          a   24
6      small    3
...
372 lifespan    1
373      six    1
374    danes    1
375    lived    1
376      ten    1
```

Comparing the size of the vocabulary that was built to the individual document vocabularies:

length(document_list[[1]]@words$word) = 247

length(document_list[[2]]@words$word) = 192

length(intersect(document_list[[1]]@words$word, document_list[[2]]@words$word)) = 63

Therefore, we have 247 + 192 - 63 = 376 words.

This is the output we want for building the vocabulary. Notice that the size of the two lists added together minus the intersection is the length of the shared vocabulary. This also gives us a total frequency count of each word across all of the documents.

5.3 Preparing the Input for the Neural Network

This section contains several small components which help build the input for the neural network. As part of the DBOW algorithm, for each word in the text, we need to prepare a batch for the algorithm in the form of input and output vectors. [3][15] As noted in [3], we need to be able to sample a text window, then randomly sample a word from that context to be used as an example output vector for one batch. In this case, for each word, we need to look at all words within a window size, pick one from random, and then vectorize the word based on its position in the shared vocabulary. [15]

To demonstrate the functionality of this component, we will reuse the Cat article from Wikipedia.[16]

```
context <- grab_context("TEST/cat.txt", 5, 2)
 [1] "cat"      "is"       "a"        "small"    "typically"
sample <- sample_context(context)
[1] "small"
vectorize(sample, vocab)
  [1] 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [39] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [77] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[115] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[153] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[191] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[229] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

All that is left before the neural network is creating the input. To reduce the number of weights that the nnet object has to keep track of, we use a single input node since the input is a one-hot vector, which is a vector where every entry is zero except for one, to represent the document, which increases the speed and reduces the storage of nnet since it no longer has to keep track of the weights for the other documents while training this document. [15][18] In this case, a one-hot vector is a sparse vector which has one or a few entries set to one, with all the rest set to zero. [8] All of the individual functions together demonstrate the functionality that DBOW needs in order to prepare the neural network for training the document vectors. [15]

5.4 Running the Network

The next step in the process is using the prepared input to run the neural network. In any neural network, there are three types of layers: input, hidden, and output. There can only be one input layer and one output layer, but any number of hidden layers. The process of moving to the output layer is as follows:

input layer -> hidden layer 1 -> hidden layer 2 -> ... -> hidden layer n -> output layer

The nnet library [18] being used can have any number of layers, and allows us to specify the size of each hidden layer. In our computation, we will be using just one hidden layer.

Then, the values in the hidden layer are put into a SoftMax function, which flattens the inputs to be within a certain range. The process is repeated similarly to move from the hidden layer to the output layer.

At the output layer, the weights will then be recalculated using the difference between the real and the expected value. Each layer's weights is recalculated individually using back propagation which will be discussed further in Section 6.2. Once the weights are recalculated, the neural network runs again with the new weights.

When running the network with nnet, we are able to specify the number of times that the neural network will run in total. This means we can decide on whether we value computation time or accuracy, since the two have an inverse relationship. The minor issue here is that nnet itself is a black box, which we cannot directly optimize.

5.5 Output

The output of the neural network is a vector of a specified size, which shows the relation to similar words from the original input hot-vector. Our nnet runs on the entire vocabulary, generating several output vectors. These vectors can then be used in a comparison to find how similar two documents may be.

For example, the results of one word are processed like so:

The input word "small" is vectorized as:

```
vectorize(sample, vocab)
  [1] 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [39] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [77] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[115] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[153] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[191] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[229] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Then after running the neural net on this specific word, an output layer of size 20 might be:

```
[1] 0.01 0.04 0.055 ... 0.009
```

These twenty values are obtained from the weights applied in the steps toward any hidden layers, and then in the step toward the output layer. Then, assume that the expected output is:

```
[1] 0.02 0.02 0.06 ... 0.007
```

So through back propagation, the weights are recalculated to get closer to this target value. The process is repeated as many times as the user specifies, and what returns for that entry in the neural net is the weights used to sample each word. After a complete run, the returned value will be a matrix consisting of these several vectors. So, each output layer becomes its own column in the matrix, which can then be used for calculation.

5.6 Example Script

To demonstrate the total functionality of the DBOW algorithm in action, we have a .R file called example_script.R. This script requires a path name to the folder that contains all of the documents, and then can be run as is to obtain document vectors and the similarity between each document. Using the Wikipedia articles on Cat and Dog, we will run the script to see the output it creates. [16][17]

```
source("example_script.R")
[1] "Finished Building Vocabulary"
[1] "Processed document vector for TEST/cat.txt"
[1] "Processed document vector for TEST/dog.txt"
[1] "TEST/cat.txt" "TEST/dog.txt"
      [,1]
[1,] 0.8960216
```

Thus the algorithm found an 89.6% similarity between the cat.txt and dog.txt files. This should be accurate since both articles are about an animal that is a common domestic pet. This script demonstrates the semantic similarity analysis that we hope to achieve by vector representations of documents and gives a rough approximation for how the algorithm should work. [3][15] Note that we used a package through R that provides a neural network implementation. [18] This network should provide us with accurate results, at the cost of being unable to optimize for our particular algorithm since we use the nnet functionality as a black box.

6.0 Refinement

This section will discuss which pieces of our project we hope to change or add to in the future to expand or improve the functionality of our R doc2vec implementation.

6.1 Future Work

In the coming weeks, we hope to add several supplemental features to our implementation of doc2vec. First, we would like to refine the parsing function in order to speed up the overall implementation. This can be done by adding stop words and a user selected minimum frequency for vocabulary words (discussed further in Section 6.5).

Another piece that we would like to add to the functionality of our implementation is a way for the user to display and plot the results of running doc2vec with our implementation. We plan to use the t-SNE algorithm [28] to compress our multi-dimensional results into a two dimensional form in order to plot and display to the user. There are multiple packages available for this purpose and we need to decide between using the tsne[21] or Rtsne[22] R packages.

6.2 MatrixMult Network

As is discussed above, the nnet is a black box, so we are unable to refine the algorithm to ensure that we have an optimized solution. So, we are creating our own neural network which uses matrix multiplication. This allows us complete control over the algorithm, and as an added benefit, will keep the Doc2Vec library free from as many extra packages as possible.

As discussed above, any layer is multiplied by a number of weights to reach the next layer. This can be modeled using matrix multiplication. If, for instance, a vector of values is the input layer, then the weights will be the following matrix where n is the input value, and m is the set of weights. There are m sets of weights, so there will be m elements in the hidden layer.

$$\begin{bmatrix} i_1 & i_2 & i_3 & \dots & i_n \end{bmatrix} \text{ and } \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} & \dots & w_{m,1} \\ w_{1,2} & w_{2,2} & w_{3,2} & \dots & w_{m,2} \\ \dots & \dots & \dots & \dots & \dots \\ w_{1,n} & w_{2,n} & w_{3,n} & \dots & w_{m,n} \end{bmatrix}$$

Figure 8 -- Matrix Multiplication applied to Distributed Bag of Words

Multiplying the two together yields the hidden layer (of course, a SoftMax function will also be applied, but that is just a linear operation on the matrix).

However, in our case, we will only be using one input value. Knowing this, we can subvert one major problem about the nnet: its storage. It will store input as a matrix either way, but we can store it as a singular value. This means that the storage of the input will be smaller. Also, computation will be quicker. Instead of an operation of matrix multiplication, it will simply be a scalar multiplied by the vector of weights, much quicker in terms of computation.

Another advantage is that we can also use multiple or singular hidden layers, tweaking as we go along. This testing can help us find bottlenecks in the code and decide if multiple layers or a singular layer will be faster.

With these two changes, our neural network can run much faster than the nnet implementation and will take up much less space.

The basic operation of the neural network is complete and in the testing phase. Meanwhile, the usage of a singular value has not yet been implemented, and the testing of different numbers of hidden layers (and different sizes of each layer) has yet to happen.

6.3 Implementing Distributed Memory

Another refinement that could be made to our product is to implement the Distributed Memory algorithm detailed in Le and Mikolov's paper [3]. To implement this algorithm, we would first need to implement a word2vec algorithm to give us word vector representations. [3] That would give us the ability to feed a neural network a one-hot vector concatenated with word vectors with an expected output to produce a document vector. [3] This can be combined with our DBOW algorithm by concatenating the DBOW produced vectors onto the DM produced vectors to create a more accurate vector representation for the document. [3] Implementing this algorithm would require a significant amount of time since there are more components that comprise this algorithm than in DBOW, even though there exists overlap between how the documents are parsed and some similarity when creating a batch to be trained by the neural network.

6.4 Following the Gensim Model

We decided to follow the gensim model for our doc2vec implementation for two main reasons. As discussed in Section 2.5, we are familiar with gensim's model as well as some of its performance benchmarks. We choose to follow gensim's doc2vec model because it is efficient and we aim to have Rdoc2vec be similarly efficient in execution. The second reason to follow the gensim model is that we believe it will be the most efficient way for our team to continue to contribute to this project before the end of the semester. As we have continued to develop Rdoc2vec through the semester, we have found success following this model and we believe that continuing down this path will allow us to achieve more before the end of the semester than attempting a different strategy.

6.5 Changes to the Parse Function

Important refinements can be made to the parse functionality that can improve the speed of the doc2vec algorithm. Currently the parse function does not have the ability to use stop words which can reduce the size of the vocabulary, which in turn improves the speed of the neural network since less weights are being updated. [15] We can also trim less frequent words using a user-defined setting which can improve the speed at which the network is trained. These two changes can improve the speed and reduce the size of the neural network for training document vectors. [15] Another change that could be made would be to specify a particular document to parse instead of parsing all of the documents in a directory at once. In this way, we could slowly build the shared vocabulary and begin to start parallelizing this functionality. Finally, we could use a natural language processing library that could combine words with the same root word. For example, we could have “test” and “tests” count for the same word “test” by stripping the -s suffix that makes the root word “test” plural. This could also reduce the size of our vocabulary and thus improve the speed of the training phase.

6.6 Any potential future changes

There are some potential future phases which may be implemented if the team has time, or we may recommend for anyone working on the project in the future.

A more direct port to the Gensim version is a future change. Keeping the input to the Doc2Vec as similar to Gensim as possible will allow users who have previously used the Gensim Python version to more easily work in R. Our current class structure is very similar to Gensim, but some of the input variables do not yet have a usage. For a complete library, all of the input variables need to have a real usage, just as they do in the Gensim implementation. [11]

In a future version, we hope to have the ability to use multiple threads on the problem. This is a feature which will take some time to implement, but has an amazing amount of potential. Since the two parts of the code, parsing and training of the neural network take a long time to complete, there would need to be multiple threads on each part, completing all future jobs in parsing before moving on to the training phase. This is a feature our team does not have time to implement, but could be useful in a future project.

Another feature which would be useful is the ability to train the model on documents found directly on the web rather than necessitating a file. Since this change would require web connectivity, it is also something we hope that a future team can work on to improve the library.

7.0 Testing

This section will outline our methodology used in collecting sample data and testing, seek to analyze our results, and discuss the problems we encountered.

7.1 Testing Process

To begin our testing process, we sought to obtain a set of sample documents that could be used to assess both the accuracy and the robustness of our implementation. Unfortunately, our client, Eastman Chemical Co., was not able to provide us with any sample documents (or even a description of their content or size) due to proprietary concerns. In response to this, we have opted to collect text from related Wikipedia pages for our testing.

Initially, we collected the HTML of these Wikipedia pages using Python's built in regular expressions and the BeautifulSoup library. We obtained the data sets detailed below in ***Table 2.*** When we began testing, it quickly became apparent that these data sets were unsuitable for our current testing environment; we will elaborate further on this in Section 7.4

Table 2 -- Initial sample document sets collected

Documents	Number of Documents	Size Range
US Congressional Districts	442	3-40kb
4-Octave Singers	42	3-140kb
Universities	24	2-35kb

When recollecting data sets, we discovered Wikipedia, a Python library [29] that serves as a wrapper for the Wikipedia API. This enabled us to collect cleanly formatted summaries and fulltext versions of articles queried by name. Our new, scaled down data sets are detailed below in ***Table 3***.

Table 3 -- Revised sample document sets

Documents	Quantity	Size Range
Virginia	6	9-19kb
Virginia2	6	1-2kb
Florida	27	1-2kb
Singer/VA	8	1-3kb

Unfortunately, the dramatic reduction in both the number and size of the documents undermined the quality of our results significantly. We will go into this more in Section 7.4

7.2 Testing with Gensim

In order to test our model, one benchmark will be using the gensim doc2vec implementation in Python. We will run the same input on both implementations and compare both speed and accuracy of the end result. By performing this test, we will know how our model measures up to the best comparison that is available. Gensim is the de facto doc2vec library, and so it is what we strive to reach with this project. However, there are two main issues with testing in this way.

First, the parsing of documents will take very different times. Since in the case of the Gensim model, we cannot measure the time it takes to parse any amount of data, we will have to simply factor this in. We will not be able to directly compare either parsing or training, which severely hurts our ability to detect a bottleneck.

Second, the calculation and comparison of the vectors will necessarily be different. Both R and Python have very different ways that they handle multiplication, estimation, and round-off errors. This means, first, that computation will vary wildly between the two during vector building. We will simply have to hope that these will not be a severe enough difference for us to have difficulty recognizing slowdowns. Second, this will affect the accuracy between the models. In building, any round-offs could change the final result by many orders of magnitude, and of course the computation of difference will be slightly different. By keeping in mind this issue, we hope for a value which is similar, but do not expect exact similarity.

We will also be showing the results of this testing to the client so that they understand how similar the results may or may not be.

7.3 Testing Dataset

For our training purposes, there are certain datasets that lend to our purpose over others. It is important that we avoid having a low number of documents, and that we use documents that will have related elements within them. Meeting these two requirements should allow us to use a dataset to test our implementation of doc2vec. To meet these requirements, we decided to use the data from sets of Wikipedia pages, as discussed in Section 7.1.

The related elements of the documents will help to group similar elements together in our end results which will show us if our implementation is working as we planned. If our documents had no similarity, then it would be harder to draw conclusions from our testing data.

After running tests on our datasets, our desired result is to have a set of fixed-length vectors with more similar documents closer together and dissimilar ones further apart. We will have a higher semantic similarity for documents that are found to be similar to one another (see Section 5.3 for an example).

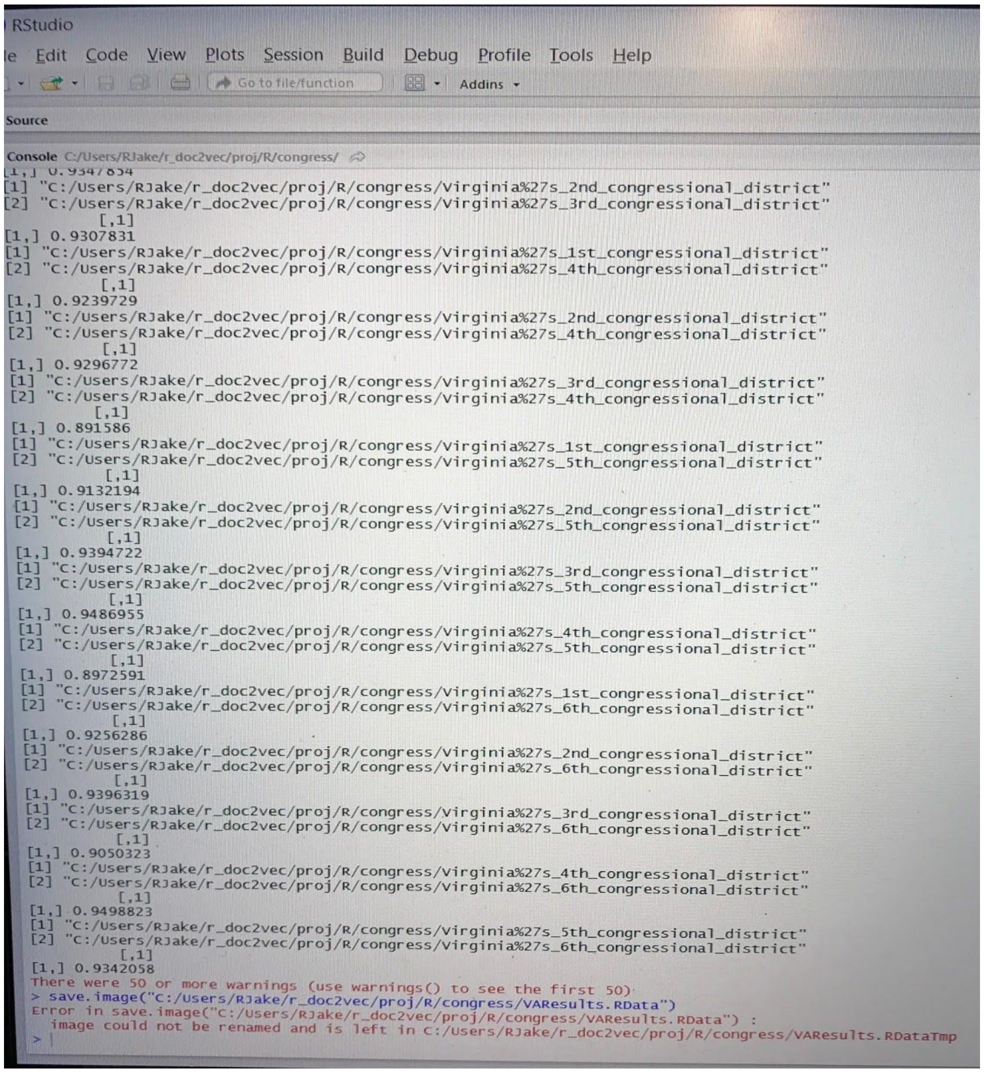
7.4 Problems with Testing

When we began testing our initial sample data sets (***Table 2***) it became immediately apparent that our machines lacked sufficient memory to vectorize documents of the size we had initially collected.

In order to correct for this, we attempted a variety of adjustments: Testing on smaller subsets of the documents collected; Shortening the vectors used to represent documents; Truncating the documents. We found that these solutions used in combination with each other were partially successful, but unfortunately the quality of our testing results was severely degraded.

We found that testing on any document sample sets of size (in the sense of quantity and length) resulted in memory allocation errors and Windows killing critical system processes resulting in blue screens and disk write errors rendering any data we were able to collect unsaveable.

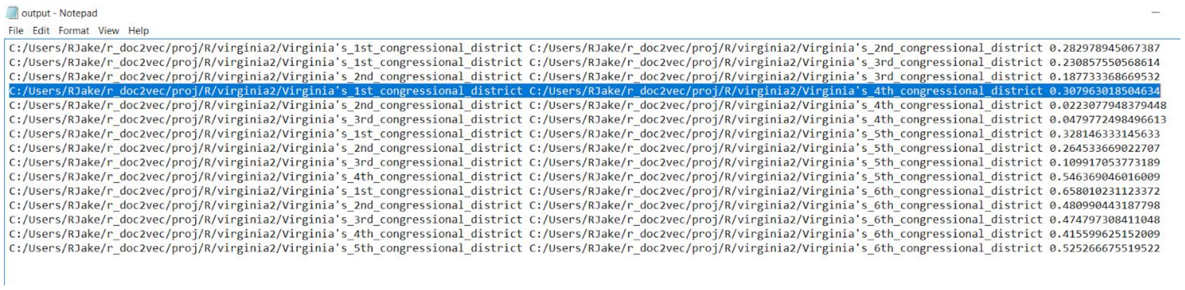
In ***Image 1*** you can see the vectorization and analysis of the documents for Virginia Congressional districts 1-6 (a tiny subset of the intended document collection) which display the high degree of vector similarity (>.9) one would expect from such tightly related documents.


RStudio
le Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Source
Console C:/Users/RJake/r_doc2vec/proj/R/congress/

```
[1,] 0.9347834
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_2nd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_3rd_congressional_district"
          [,1]
[1,] 0.9307831
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_1st_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_4th_congressional_district"
          [,1]
[1,] 0.9239729
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_2nd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_4th_congressional_district"
          [,1]
[1,] 0.9296772
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_3rd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_4th_congressional_district"
         [,1]
[1,] 0.891586
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_1st_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_5th_congressional_district"
          [,1]
[1,] 0.9132194
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_2nd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_5th_congressional_district"
          [,1]
[1,] 0.9394722
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_3rd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_5th_congressional_district"
          [,1]
[1,] 0.9486955
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_4th_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_5th_congressional_district"
          [,1]
[1,] 0.8972591
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_1st_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_6th_congressional_district"
          [,1]
[1,] 0.9256286
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_2nd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_6th_congressional_district"
          [,1]
[1,] 0.9396319
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_3rd_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_6th_congressional_district"
          [,1]
[1,] 0.9050323
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_4th_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_6th_congressional_district"
          [,1]
[1,] 0.9498823
[1] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_5th_congressional_district"
[2] "C:/Users/RJake/r_doc2vec/proj/R/congress/Virginia%27s_6th_congressional_district"
          [,1]
[1,] 0.9342058
There were 50 or more warnings (use warnings() to see the first 50)
> save.image("C:/Users/RJake/r_doc2vec/proj/R/congress/VAResults.RData")
Error in save.image("C:/Users/RJake/r_doc2vec/proj/R/congress/VAResults.RData") :
  image could not be renamed and is left in C:/Users/RJake/r_doc2vec/proj/R/congress/VAResults.RDataTmp
> |
```

Figure 9 -- Results of VA Congressional Districts 1-6 comparison, had to be captured via camera due to technical difficulties

The revised data sets were significantly truncated versions of the original documents (Wikipedia summaries only versus full text articles) and as a result contain far less information to use when creating vectors for comparison. Below in ***Image 2*** the results of the vectorization for the truncated versions of Virginia Congressional Districts 1-6 can be seen.



```
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_1st_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_2nd_congressional_district 0.282978945067387
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_1st_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_3rd_congressional_district 0.230857550568614
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_2nd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_3rd_congressional_district 0.187733368669532
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_1st_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_4th_congressional_district 0.307963018504634
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_2nd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_4th_congressional_district 0.0223077948379448
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_3rd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_4th_congressional_district 0.0479772498496613
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_1st_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_5th_congressional_district 0.328146333145633
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_2nd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_5th_congressional_district 0.264533669022707
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_3rd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_5th_congressional_district 0.109917053773189
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_4th_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_5th_congressional_district 0.546369046016009
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_1st_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_6th_congressional_district 0.658010231123372
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_2nd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_6th_congressional_district 0.480990443187798
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_3rd_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_6th_congressional_district 0.474797308411048
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_4th_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_6th_congressional_district 0.415599625152009
C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_5th_congressional_district C:/Users/RJake/r_doc2vec/proj/R/virginia2/Virginia's_6th_congressional_district 0.525266675519522
```

Figure 10 -- Results of Truncated VA Congressional districts 1-6 comparison

As you can see, the vectorization comparison between Virginia’s 1st and 4th congressional district yielded a value of 0.3079 as compared to the non-truncated document’s comparison’s value of 0.9307.

Unfortunately, this indicates the presence of an unsurmountable testing constraint. Without additional computing resources, we will not be able to test a sizeable collection of sufficiently large documents.