Tobacco Testimonies

CS 4624: Multimedia, Hypertext, and Information Access
Instructor: Dr. Edward Fox
Client: Dr. David Townsend

Virginia Tech
Blacksburg, Virginia 24060
June 5, 2019

Final Report
Team Members: Nick Onofrio, Nick Sorkin, Devin Venetsanos, Michael DiFrancisco and Campbell Johnson

Table of Contents

Executive Summary

Tobacco companies have had some of the best marketing strategies over the past century. It is well documented and known that tobacco produces both mental and physical health issues and yet these companies have found ways to remain as one of the largest businesses. The goal of our project is to assist Dr. Townsend in his research to understand Big Tobacco's strategies.

This will be done by taking as many of the fourteen million documents released by tobacco companies online and presenting the data in a meaningful way so it can be analyzed. This project will be hosted on a Virtual Machine provided to the team by Dr. Fox and the VT Computer Science department. The idea for the process of this project is to first begin by gathering the documents from online, turning them into a useable text format, then feeding these documents to a Doc2Vec machine learning tool that was created by Gensim. As we are using Gensim's model it is already pre-trained so we then need to take this data and cluster it so that it is presentable in a usable manner. Thus Dr. Townsend and many others can try and use this system to further their research.

The final product will be well documented for how to use the system and maintain it. This way Dr. Townsend can do what he wants with the system and any future developers for it can understand how the system works. This system is comprised of different online components such as a Gensim doc2vec model and a fast approximate nearest neighbor similarity package from Gensim to do the clustering of the data. This has all been stored and set up on the virtual machine provided by the CS department so it should be accessible as long as the user is connected to the campus wifi. Through this project our team learned many things about working with a client, working with new technologies, and how to go about tracking and presenting progress to others.

1. Introduction

Since there are over fourteen million documents released online from Big Tobacco companies it's nearly impossible for someone to scan through all those documents manually. That's where we are trying to aid people by taking those documents and presenting them in a meaningful form so that they can be analyzed by people to try and create useful findings. The process of how our system works is first we gather these documents from online, then we convert them into a usable form such as a text file. Next we feed these files to our Python script that converts these into vectors and uses the trained model to group these with specific tags and cosine similarity numbers. Lastly we use this information to cluster the data and present it in a meaningful manner that a person could look at to discern information.

1.1 Objective

Our objective is to create a system for Dr. Townsend that allows him to look through these documents to identify useful data in an efficient and effective manner. We also need this system to be usable by people without computer science backgrounds in a reasonable manner as well as have the ability for someone else to improve the system if a need arises. This system will have four key elements to it. The first being gathering as many of the fourteen million documents as possible. These documents come in all shapes and sizes from company memos, videos, advertisements, letters, etc. so the system has to be able to handle all of those. The second key element is Doc2Vec model and training it work with the data we are providing. The third key element is clustering the data so that it can be analyzed by users to create trends.The

last section of the project will be creating documents for Dr. Townsend so that he can understand how to operate this system to its full potential as well as allow any potential future developers to understand the system so they could add to the current system. Completing all of these different objectives would signify that we have accomplished our goal in the project and will allow us to call it a success.

1.2 Client

Our client is Dr. David Townsend who teaches entrepreneurship, strategy, innovation management, and data analytics. His research focuses on capital acquisition processes, angel investments, venture capital, growth strategies, and organizational development (the last 2 more geared towards our group). Some of his awards include: 2012 Best Social Entrepreneurship Paper Award for the Academy of Management and the Outstanding Reviewer Award by the Journal of Business in 2010 & 2011.

He has become motivated to study the reports on Big Tobacco because of the direct impact cancer has had on his life. It's also because he recognizes that the tactics used to keep consumers smoking were one of the best business strategies ever seen. Although he recognizes that their strategy was unethical he believes there are some valuable things to learn from these documents. Understanding consumers wants and needs are important to any major company and there is no question that Big Tobacco knew their consumers wants and needs.

1.3 Scope

The scope of this project is to create a system that Dr. Townsend can use to research big tobacco data in a more efficient and effective manner. This project is a semester long and is following software design principles of identifying requirements, designing the system, implementation of the system, and testing of the system. We will provide the system as well as the documentation for the system and then it will be up to Dr. Townsend and Professor Fox if there are any additions they want to add to the system. We wanted to maintain a level of sophistication with the project but also have it maintainable so that we can deliver a final product at the end of the semester.

We have also stretched the scope of the project by attempting to provide a scalable application that can be used to create clusters around different groups of documents. We have decided to incorporate a user interface that will allow a user to specify the type of documents that should be clustered. Our client specifically wants documents clustered from all documents tagged as being related to the Surgeon General Reports. We will still deliver that, but our system will also allow him to cluster documents tagged as a Deposition for example.

1.4 Constraints

For this project we had a couple different constraints. The first being time. We only had a semester to complete the project. We each have different class schedules, obligations, and trips planned at separate times. We therefore had to plan around these to find times for us to meet each week. This constraint was also especially difficult due to not having anyone with expertise in the areas our project revolved around. This meant that before we could perform any work we had to do plenty of research. We also went and talked to Dr. Fox a couple times this semester to help make sure we were on track because he was very knowledgeable about our subject matter and knew how to help.

Another constraint we had to work with this semester was resources. The CS department and Dr. Fox arranged for us to get a virtual machine. The virtual machine has an x86_64 architecture, 4 cores, 32GB of RAM, 2000GB of disk space, and runs on Cent OS, so we had to fit our project requirements to fit on this system. Thus we had to be careful with downloading select programs and the vast amount of tobacco documents.

We also had constraints with our current coding abilities. To complete this project we had to learn multiple different technologies that we hadn't been exposed to previously. For example none of us had been exposed to machine learning previously so that took some time to understand gensim and how their Doc2Vec model worked. We also had to learn about the AnnoyIndexer and clustering so that we could take the Doc2Vec data and make it useful.

Another constraint we had was sustainability and scalability. Although these constraints weren't specified by our client, we wanted to create a solution that would outlast just this semester. We wanted the system to allow our client to perform analysis

on given range of documents whenever he desired. Without that functionality then our system wouldn't be useful to our client.

2. Requirements

Our client, Dr. Townsend, wants to be able to analyze sentiments of the 14 million tobacco related documents that were released to the public. The major requirement that stems from this is that we need to be able to cluster these documents based on sentiment, essentially grouping together documents where the author had a similar tone and goal. This is important because once all of these documents are grouped together, it makes it much easier for non technical researchers to read, research, and identify key strategies that were used over and over throughout the past 70 years to influence public policy on the big tobacco industry.

To reach this overall requirement, there are 4 necessary steps that we must complete, each one handing off the finished product to the next until we have fully clustered and analyzed data. The four steps are gathering the documents, then formatting the documents into a useable manner, then feeding these documents to a Doc2Vec machine learning model and then lastly using the resulting data to cluster it and present it in a useable manner. Another requirement with this is that the training of the data with gensim model be accurate and that the clustering of the data make sense.

2.1 Acquiring the documents

All 28 million documents that were released by big tobacco companies are available to download individually through the University of California, San Francisco library website. This site offers an API using Apache Solr to programatically acquire the metadata for each document. This means that we must create a script to hit the API based off of certain search terms and store the necessary data, which can then be used to start the clustering process.

It wasn't until near the last quarter of our semester long time constraint that we realized you can't actually get the full text of each document from the API, which was our plan all along. Fortunately the UC San Francisco library website allows us to

download all 28 million documents in a zip file, which we talk about more in the implementation section.

2.2 Menu User Interface

The tobacco website has hosted a Solr API with documentation on their website. Which can be found here. In order to allow our client to interact with the API we have created a simple menu user interface that will simply take values for each field the API accepts and runs a query that reflects the values entered into the menu system. It then stores each document id, which allows us to look up the document in the root directory of our virtual machine where we have downloaded all of the documents. It is important to note that the API provided by the tobacco website only sends back metadata and as such we must rely on other means to get the full text of the document. Although this menu interface is slightly out of the scope of the requested project we are hoping it may come to be an important part of our clients research down the line when he wants more clusters based off of other keywords or topics.

2.3 Document Conversion

To be able to use the documents and feed them to the Doc2Vec model they have to be formatted correctly. Since we couldn't just analyze the PDF directly or use the API provided since it didn't give enough information, we decided to convert the documents. Specifically we wrote a Python script that would convert PDF's to TXT files so that we could then read in the TXT to feed the Doc2Vec model. We used PYPDF2 to aid in the conversion. Technically speaking, we first started by storing all the PDF's in one folder and then reading in the documents one at a time from the folder. Then we used the PYPDF2 PdfFileReader method to obtain the information from the file in a string format. We then wrote that information into a text file in a new folder and setting the name of the file to be the same as the pdf. Once this was all complete we were able to read in all the text files for the model script straight from the folder using glob.

```
# importing required modules
# https://www.geeksforgeeks.org/working-with-pdf-files-in-python/ is the source where we learned how to do this
#PyPDF2 is a pure python PDF Libraray capable of splitting
#merging together, cropping, and transforming pages of
#PDF files
import PyPDF2
#Techniques to match specified patterns according to Unix rules.
import glob
import os

fileNames = []
#filepath to the folder with txt files
path = '/home/user1/pdfs/*.pdf'
saveLocation = '/home/user1/txts/'
#glob.glob return list of path names as string
files = glob.glob(path)
#https://www.quora.com/How-do-I-read-mutiple-txt-files-from-folder-in-python idea to loop like this to read in files came fr
#Loop through all of the files in the folder
j = 0
```

In the photo above we start by importing a couple different things, most importantly being PyPDF2. That is the platform we use for most of the pdf conversion. After that we set up the locations for the directory where the pdfs are held and where we are going to place the text files.

```
j = 0
for name in files:
    # creating a pdf file object given a name
    # r is for reading b is for binary
    pdfFileObj = open(name, 'rb')

    # creating a pdf reader object
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    # printing number of pages in pdf file
    print("File # " + str(j) + " Page #: " + str(pdfReader.numPages))
    #opening the file we are going to write to
    strLength = len(name)

    locationName = os.path.join(saveLocation, name[17:(strLength-3)] + "txt")
    file = open(locationName, "w")
    i = 0
    #Loop through the number of pages in the object
    for i in range(pdfReader.numPages):
        # creating a page object
        pageObj = pdfReader.getPage(i)
        #write to the file. the encode portion is to make it ascii complient
        file.write(pageObj.extractText().encode('utf-8').strip())
        # extracting text from page
        #print(pageObj.extractText())
    file.close()
    # closing the pdf file object
    pdfFileObj.close()
    j += 1
```

In this picture our code begins by looping through each file from the given directory. We get the name of the file and then create a pdf reader object of that filename using PyPDF2. Then for each page in this file we write the text from it into a new file. We do this until we cycle through each page in that file. The text file is the same name as the pdf version and then we move on to the next pdf until each pdf file has been converted.

2.4 Doc2Vec

After the necessary data is acquired through the API, we can then start the process of analyzing the data. The first step here was to read in all the text files using glob and store that information in an array. After that we began using tagged document to create tokens and store the data. We cycled through each token in the array and stored them and then tagged them with the specific file they came from. Next we began building our Doc2Vec vocabulary. We did this with a vector size of twenty, an initial learning rate of 0.025 (otherwise known as minimum alpha), and epoch size of one hundred (the number of passes) and the distributed memory algorithm (PV-DM). Once the vocabulary was build we trained our new model and saved it. Once saved our other Python scripts could load the newly created model based on our vocabulary.

2.5 Clustering

After making vectors from each document, we can sort and group this data accordingly. Taking advantage of approximate nearest neighbor algorithms, we can group this data based on vector forms to help us gain an understanding of how different documents are related. These vector forms can be extracted using the Doc2Vec model that we have created as well as the complete corpus that is created via the call to the library database API and our local database of documents.

Once document vectors are extracted from the Doc2Vec model, we can use an array of all of these vectors to perform clustering. There are many different clustering algorithms available to us through different python packages, each with their own time complexities and vector formats for input into the algorithm, so we must decide which one fits our needs the best.

This is the most crucial requirement, as once these documents are clustered it makes it easier for business analysts to find connections and uncover deeper strategies that these tobacco companies employed.

2.6 Sentiment Analysis

Although clustering is our clients most important requirement, we can use special sentiment analysis tools to find out the common denominator behind what each cluster means. This way our client can know before even reading a document the basic idea behind each one, which allows him and his team to spend less time reading and more time analyzing different strategies.

3. Design

After meeting with Dr. Townsend and asking him about his needs he said that anything could be useful to him and he didn't have any specific preferences about the design. Therefore we had to come up with our own design plans for this system and we tried to mimic some design principles. Our main goal was to make the system easy for Dr. Townsend to use. Next we wanted to make it simple to input data as well as to see the data outputted.

The current design we were working on implementing would be a simple menu system, where our client could select certain specifications that he wanted to run analysis on. Users can select the desired filters they want on the collection, the type of embedding to use on the collection, and potentially the type of analysis to run on the cluster that is obtained. These specifications can be run, and the results will be shown in a window to the user, as well as some visualizations if time permits.

Towards the end of our project we started to realize that while having an easy to user interface would be super helpful we started to focus less on the menu and more on other parts of the application. We figured that making sure everything in our application connected correctly, was efficient, and produced accurate data would be the most important thing and for the moment we could live with a less fully functioning interface as long as it solved its purpose and allowed us the use of the system to be effective.

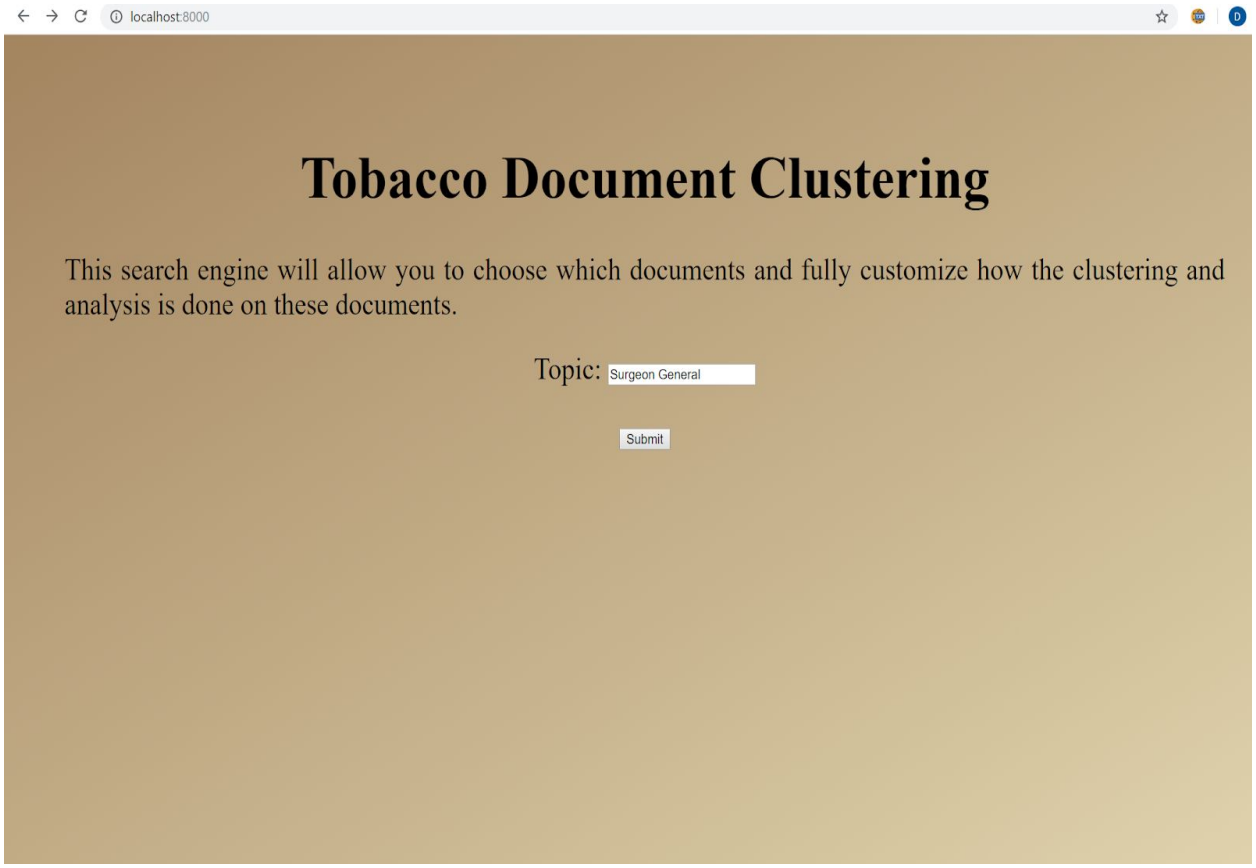
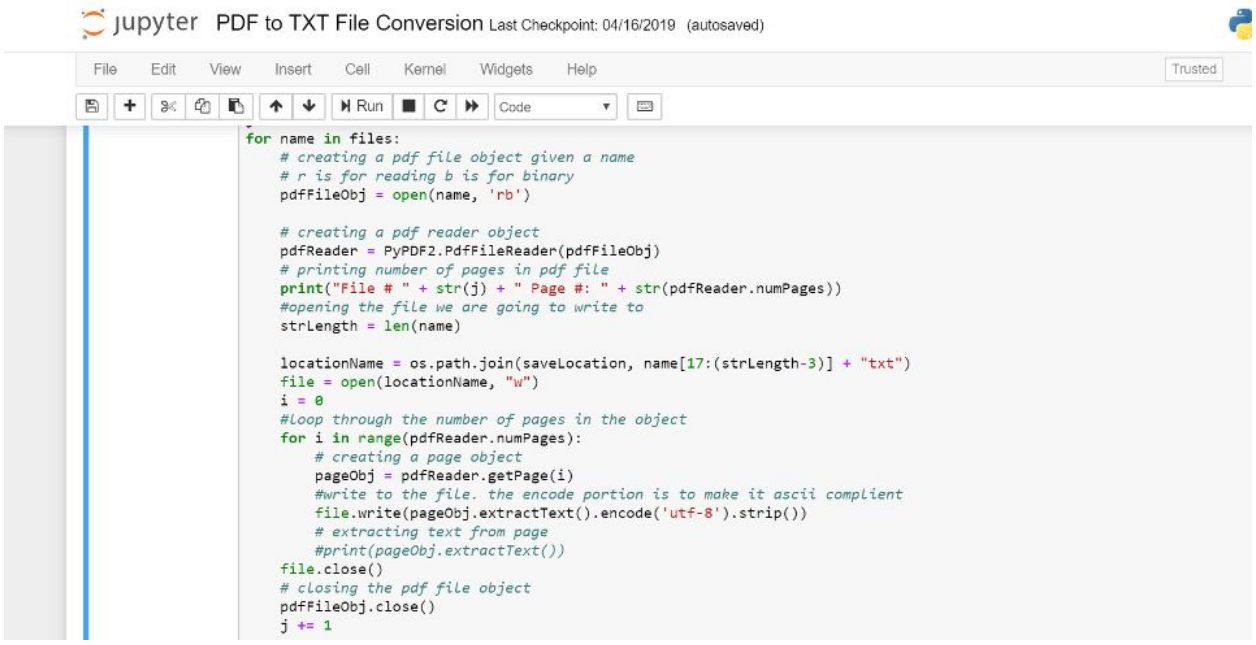


Figure 3.1: Our current Menu, it is simple and is solely designed to allow users to specify their desired constraints on the entire pool of documents hosted on the Tobacco Website.

4. Implementation

Based on our very specific set of requirements and what we eventually wanted to deliver to our client for further research, our team decided to use Python for all of the steps in our project. This was due to a few reasons, the first one being the biggest: Python is an open source language with a high user base and lots of external libraries. This means that if someone wanted to expand on our project in the future, the support and examples available for the Python language will allow virtually anyone to read, understand, and extend our code. This also means that there are many different external, open source libraries already available for us to use, meaning that we don't have to reinvent the wheel or pay for expensive professional packages. For example we used many gensim packages for our model, AnnoyIndexer and Scikit Learn Birch for assistance with clustering, nltk for tokenization of vectors, glob for reading in files, errno for assistance with errors, and PYPDF to help convert PDF's to text files.



The implementation, just like the requirements, has four basic parts that all work together. Because of this, we can create one Python project will multiple parts, where each part passes on its finished product to the next and the whole project itself can be run in one continuous script. The four main sections are the gathering of the documents, then the conversion of the documents into a usable format, then feeding the document data to the Gensim Doc2Vec model to build a vocabulary and create a new model, and then feed the data from the model to be able to then cluster it and present it in a usable manner.

4.1 Acquiring the documents

The document retrieval process begins with the interface that is already granted to the general public from the Truth Tobacco Industry Documents library. One of our challenges was extracting these documents to be able to use them in our application. We sought out to create a menu where you can query the documents by a variety of parameters to narrow the clustering to a specific set of documents. The goal of the menu is to call the API provided by the library and get a list of document IDs to train the doc2vec model. Once the array of document IDs is created from the menu, a Python script is ran to create a directory of all of the documents given by the array of IDs.

```
import sys
import shutil
import os


def getDocs(docIdList):

    pathLoc = 'home/user/dump'
    shutil.rmtree(pathLoc)

    try:
        if not os.path.exists(pathLoc):
            os.makedirs(pathLoc)
    except OSError:
        print('Error creating directory')

    j = 0
    for docId in docIdList:
        print(docId)
        id1 = docId[0]
        id2 = docId[1]
        id3 = docId[2]
        id4 = docId[3]

        path = '/home/user1/root/' + id1 + '/' + id2 + '/' + id3 + '/' + id4 + '/' + docId
        print(path)

        shutil.copyfile(path + "/" + docId + ".ocr", pathLoc + "/" + docId + ".ocr")

        docFile = open(path + "/" + docId + ".ocr", "r")
```

The directory will be deleted and recreated by the Python script each time the menu is used. The script takes each document ID and locates the file within our repository of documents. These documents are all separated by the lettering of their document IDs. The script above gathers the documents into a single directory so that the text can be scraped and inserted into the model. This process is dependent on having downloaded all of the documents from the online library onto the virtual machine. This process took a while to download the 28 million documents, decompress the documents, and create a system to organize the documents. Lastly, all of the documents are in OCR format for optimal data scraping and text formatting.

4.2 Menu User Interface

Our interface will be written in Python with Django. Using Python version 2.7.5, and Django 1.5.12. Since the overall goal was to be able to hit an external API, and then be able to run the rest of our algorithms on the result this tech stack made the most sense. Firstly, most of the packages our solution takes advantage of are written in Python, and from the start of the project we agreed upon using Python for our solution. Django is extremely helpful in creating web applications and most web applications must make a lot of API calls so it follows that Django is helpful in making API calls. This

also sparked the idea of creating a Menu user interface that could be interacted with on localhost. This would allow our client to cluster documents based on different topics whenever he wants. Instead of just providing our client with the clustered documents on the requested topic we decided to try to make the project more scalable. Our client mentioned that in the future he would want other documents belonging to a specific topic to be clustered and would have another team perform that work. We thought this approach may allow him to perform his own analysis in just a few minutes. The API can be interacted with by appending queries to the end of the base URL.


```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    "params":{
      "(q":"topic=\"surgeon general\" AND person=\"MASELLI\"",
      "wt":"json"}},
  "response":{"numFound":14907619,"start":0,"docs":[
      {
        "id":"yscd0000",
        "tid":"rgt00a00",
        "collection":["Council for Tobacco Research Records"],
        "collectioncode":["ct"],
        "availability":["no restrictions",
          "public"],
        "case":["MNAG"],
        "author":["EISENBERG AD, CTR"],
        "documentdate":"1994 December 20",
        "type":["letter"],
        "pages":1,
        "recipient":["SMILOWITZ H, UNIV CT HEALTH CENTER"],
        "mentioned":["SAB"],
        "description":"TRANSMITS MATERIALS RELATED TO APPLICATION PROCESS; ATT",
        "bates":"60024669",
        "dateaddeducsf":"2002 February 01"},
      {
        "id":"qscd0000",
        "tid":"tgt00a00",
        "collection":["Council for Tobacco Research Records"],
        "collectioncode":["ct"],
        "availability":["no restrictions",
          "public"],
        "case":["MNAG"],
        "author":["MASELLI M, CTR"],
        "documentdate":"1995 December 12",
        "type":["letter"],
        "pages":1,
        "recipient":["SMITH A, UNIV LOUISVILLE"],
        "description":"TRANSMITS PAPERWORK; ATT",
        "bates":"60024671",
        "dateaddeducsf":"2002 February 01"},
      {
        "id":"sscd0000",
        "tid":"vgt00a00",
        "collection":["Council for Tobacco Research Records"],
        "collectioncode":["ct"],
        "availability":["no restrictions",
          "public"],
        "case":["MNAG"],
        "author":["EISENBERG AD, CTR"],
        "documentdate":"1995 June 12",
        "type":["letter"],
        "pages":1,
        "recipient":["SMITH A, UNIV LOUISVILLE"],
        "mentioned":["SAB EXECUTIVE COMM"],
        "description":"TRANSMITS EXPLANATORY MATERIALS; ATT",
        "bates":"60024673",
        "dateaddeducsf":"2002 February 01"},
```

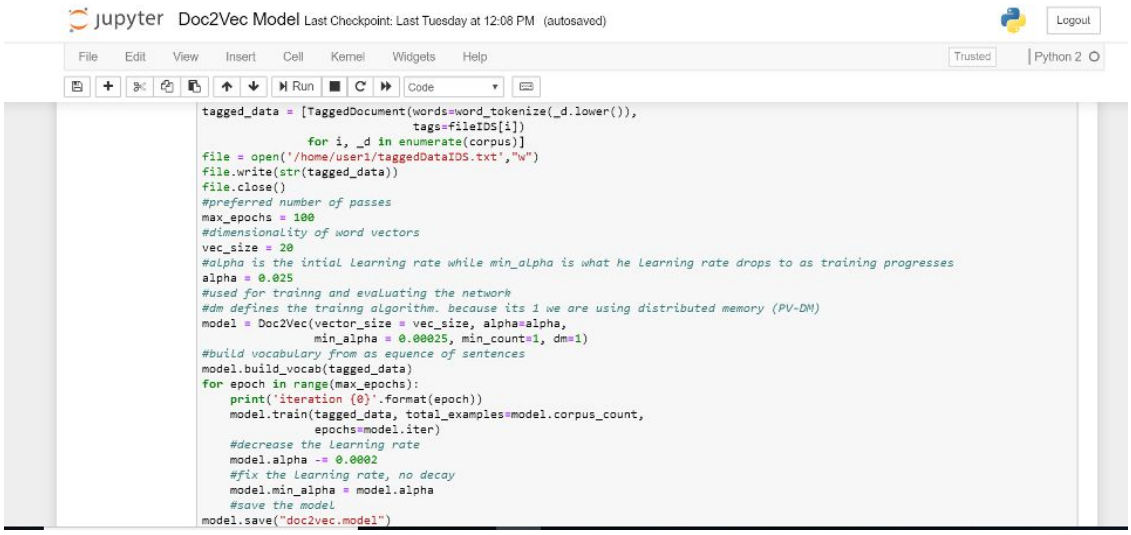

Figure 4.1: This shows a sample JSON response from the API. The API can respond with XML, XHTML, but we are specifically requesting a JSON response. This sample response shows all of metadata

that is in the response from the API. Our code only cares about the ID of each document, so we will pick out the ID from each of the documents in the response.

The Menu will simply take a user input on the topic desired and concatenate that on to the back of the base URL and make a get request to the constructed URL. It will be given back thousands of documents, but per the design of the API they will be given 100 at a time. The application will go page by page and grab the ids from every document. From here the application will cross reference the documents we have downloaded to the virtual machine to provide the entire text of each document found. We must cross reference the downloaded documents because the API only responds with metadata about each document. Once the clustering as been performed, the user will be redirected to a results page. The results page will list all of the clusters that were created from the program. Clicking on any of the clusters will display the contents of a text file containing the URLs for all of the documents in the cluster. The user will then be able to copy and paste that URL into their browser to find the exact document on the Tobacco website.

4.3 Doc2Vec

Since there are over fourteen million documents of different types (memos, emails, videos, etc.) we wanted to use Doc2Vec to tag the documents and use them for analysis. Once the documents had been stored and turned into text files we can then use them to feed into the Python script and be used to tag each vector with a specific ID based on the file it came from. After that we feed these tagged documents to then build a vocabulary. Once the vocabulary is built we then train a new model and once that is complete we save the model so that our other Python scripts and import the model and work with it. The time it takes to train the new model varies depending on the amount of documents being used. If you use a couple hundred documents it's fairly quick and can take less than a minute however it can take exponentially longer based on the quantity of documents used.



4.4 Clustering

After passing our set of documents through the doc2vec algorithm, we now have a massive map of vectors with each vector corresponding to each document. We then tried to use the Python library from Gensim, similarities.index, to cluster the vectors based on the approximate nearest neighbor algorithm. This package integrates our vector representation of documents with a well known package called Annoy (Approximate Nearest Neighbor, Oh Yeah!), which was created by Spotify to handle clustering for their recommended songs list.

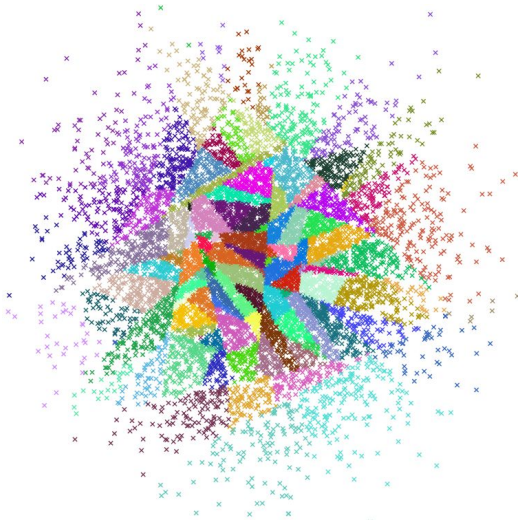
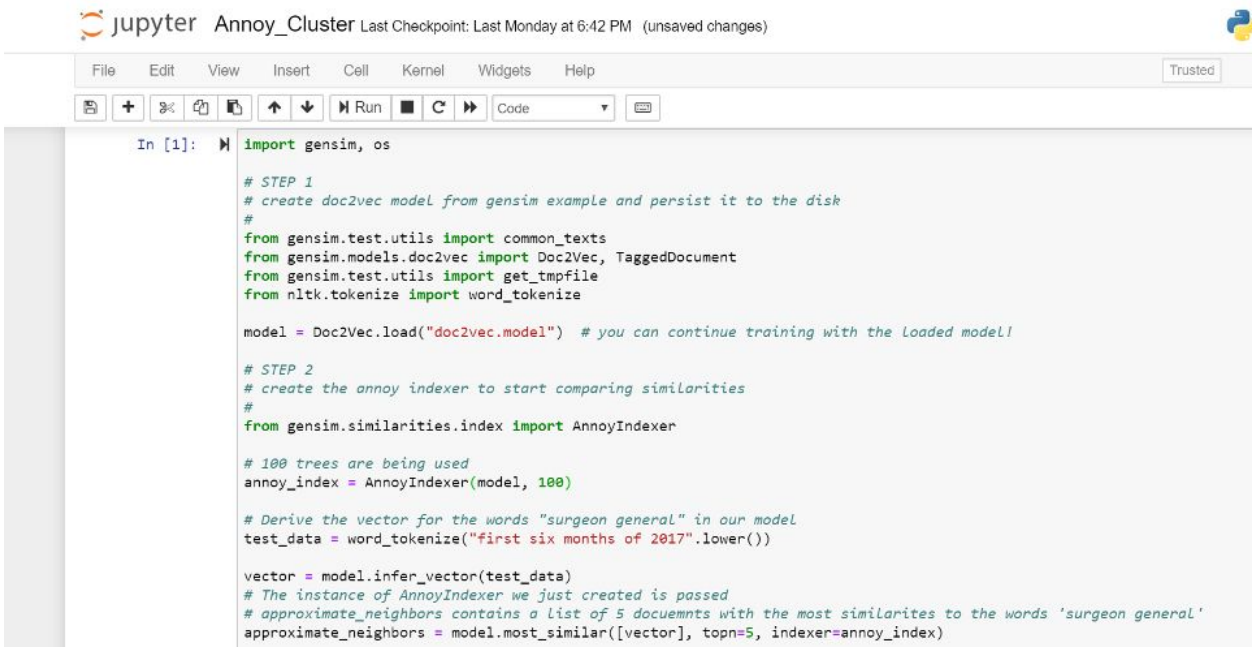


Figure 4.2: The Annoy logo, showing an example of how the approximate nearest neighbor algorithm works on a large data set

The gensim similarities.index package allows us to integrate the Doc2Vec model directly with a clustering algorithm that is based off of stored indexes. Doc2Vec has a specific method, most_similar, that shows the most similar documents given a vector input. Using the AnnoyIndexer, which is created directly from the DocVec model itself, we can use the vectors stored in the Doc2Vec model by calling the most_similar method and passing in the indexer itself.



The AnnoyIndexer worked great, but we realized that there was one big issue that prevented us from clustering. The most_similar method takes in a base vector to find other vectors that are most similar to to that base vector. This isn't actually clustering, where all the vectors are grouped, but rather checking what vectors are similar to one vector in particular. This made us change our approach, although the AnnoyIndexer is still very helpful in checking if our cluster results are accurate.

In order to actually cluster all of these documents, we needed to find a algorithm that takes all the vector representations of the documents we want to cluster and then performs the clustering. For this we decided on a well known machine learning python package called scikit-learn. Scikit-learn provides multiple different clustering algorithms, each with their own parameter constraints and time complexity. After weighing the pros and cons of each, we decided on using the Birch clustering algorithm. It simply takes in an array of all of the vectors corresponding to each document, which fits our needs perfectly. It then uses two methods, fit and prediect, so perform the actual clustering of

our document vectors. The fit method sets up the Birch object to know how many vectors we have and how long each vector is, so that the algorithm can most efficiently perform the clustering. The predict method then performs the actual clustering, and returns an array with each vector now corresponding to a specific cluster. We decided to use 10 clusters in our implementation, but it is possible to change this number as well as other parameters on the Birch algorithm to hone our results.

From this array of clusters returned from the Birch.predict() method, we can then extract which documents are in each cluster using a simple for loop. Figuring out then how to extract the actual document ID's then became the next problem we had to solve.

```
from sklearn.cluster import Birch
num_clusters = 10
brc = Birch(branching_factor=50, n_clusters=num_clusters, threshold=0.1, compute_labels=True)
brc.fit(X)

clusters = brc.predict(X)

print ("Clusters: ")
print (clusters)

cluster_ids = []

for i in range(num_clusters):
    c = []
    for j in range(len(clusters)):
        if clusters[j] == i:
            c.append(fileIDS[j])
    cluster_ids.append(c)

clus_count = 0
for x in cluster_ids:
    print("Cluster {}:".format(clus_count))
    print(x)
    print("\n")
    clus_count += 1

return cluster_ids
```

The birch clustering algorithm in action

One challenge that we ran into here was retrieving the related id's of each document once they are clustered. This is important because once we create the clusters, we then need to return the documents from each cluster so that our users can start analyzing the what makes these clusters unique and how the corporate strategies used align with each cluster. We solved this by keeping a dictionary of document ids that correspond to the id that is assigned in the Doc2Vec model. When we run the clustering algorithm, we are returned with a list of vectors in each cluster with the Doc2Vec id being the first value in the vector. From there we can use the mapping to

retrieve the actual document id, and return the list of clustered documents to the user. The below screenshot shows you how this ID mapping was accomplished via the fileIDS dictionary, and also shows how all of the documents we acquired are stored in a tokenized corpus and then changed into vectors via our Doc2Vec infer_vector method.

```
count = 0
for name in files:
    try:
        with open(name) as f:
            fileIDS[count] = name[17:len(name) - 4]
#had an issue with characters not being identified as ascii so use the idea here
#https://stackoverflow.com/questions/43358857/how-to-remove-special-characters-except-space-from-a-file-in-python/43358965
#to remove them
            final = " ".join(re.findall(r"[a-zA-Z0-9]+", f.read()))
            token = word_tokenize(final)
            #add the text to the corpus
            corpus.append(final)
            corpus2.append(token)
    except IOError as exc:
        #error handling
        if exc.errno != errno.EISDIR:
            raise
    count += 1

model = Doc2Vec.load("/home/user1/doc2vecNew.model")


X=[]
start_alpha=0.01
infer_epoch=100
for d in corpus2:
    X.append(model.infer_vector(d, alpha=start_alpha, steps=infer_epoch))
```

4.5 Sentiment Analysis

The sentiment analysis of our project is going to be highly dependent on how the clusters are formed. Our goal is to have a well constructed analysis page that displays both the high and low level data that is drawn from our word2vec and doc2vec programs. We want to give the user as much information available to them as humanly possible.

The initial design of the sentiment analysis is going to be a two page system. One page is going to be the navigational system that allows the user to travel through each cluster, document, and word to find all of the information gathered about each respectively. The documents and words will be listed in order of relevance graded by the word2vec and doc2vec criteria. For example, the documents with the highest similarities according to the model will be listed higher than the documents with lower similarities. This goes the same for the words/phrases within each document. If time permits, we would love to include a panel that would enable the user to choose how the

data is showed to them. This data can be distributed in many different useful and insightful ways so we should give that choice to the user.

The second page will be an overall display page of how the programs work together. The main information and analysis provided on this page will be how the clusters are formed, what criteria each cluster embodies, most used words/phrases across every document, the embedding selected by the user, document type selected by the user, and many other of the criterias that went into making the program unique. The most valuable information should be displayed on this page while the other page will provide the ability to perform a deep dive on the data collected.

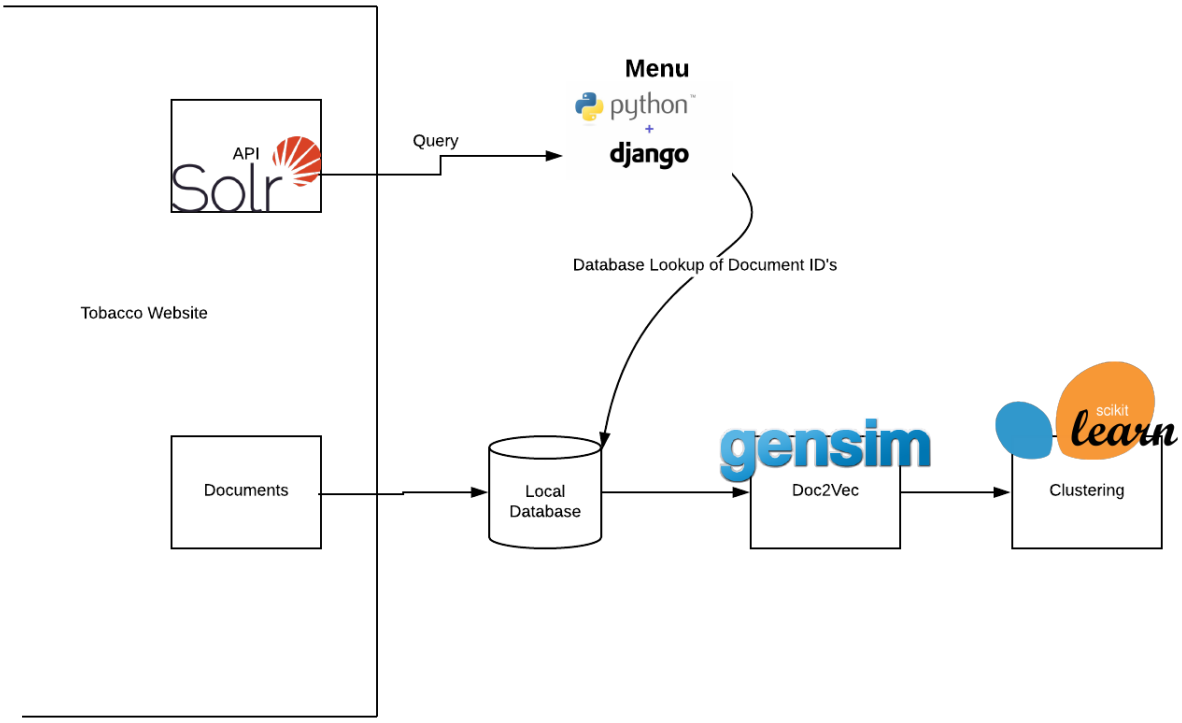


Figure 4.3: The overarching system design for our current solution. We have our Solr API which we can run queries against. We also have all of the documents, since the API only gives metadata and we can download those to our local machine. Our Menu will run queries against the API and match the resulting document ids with document ids stored in the local database. Then these resulting documents will be passed on to Doc2Vec and subsequently to the pretrained clustering algorithm we have chosen.

5. Testing and Evaluation

What is the point of any system if it doesn't work correctly? Our system reads in data, clusters it, and then analyzes it so it can be used for research. The system working correctly is crucial for our project. Ideally to test that it works correctly we need to feed it different data sizes and then check it's output and see if the documents correctly relate.

Our testing regime consists of building a portion of the program and then testing it immediately. It doesn't make sense for use to build our application and then only test it at the end of its initial completion. Instead, we are going to develop the document gathering program and thoroughly test it. Next, we are going to utilize the Gensim doc2vec framework on the gathered documents. We plan to test the framework to make sure the vectors are created how they are supposed to. This section may not require as extensive testing as Gensim is a reliable source.

Lastly, we will need to test the clustering of the doc2vec results to the embedding that was selected. The sentiment analysis section is dependent on the final clusterings and simply relays information. To do this, we used the gensim similarities package to test the validity of the clusters that were outputted by the scikit-learn package. If two documents were clustered together by the Birch clustering algorithm, we could feed one of these into the the AnnoyIndexer most similar method. If the other document in the cluster appears in the results of this method, we can confirm that our clustering is working successfully.

Very little testing is necessary for sentiment analysis once we validate that all of the data is accurate. The big key is that once we read in a set of documents and then send them through the Doc2Vec model and then cluster the data, if they data doesn't appear to be clustered correctly then we know something went wrong in the process of its creation. To fact check this we can run the data through ourselves, the use of online resources, as well as Dr. Townsend and Dr. Fox.

6. User's Manual

To get access to the Menu system that will allow you to create clusters on specific Tobacco Documents, talk to Dr. Townsend or Dr. Fox to obtain access to the github. Once you have access to the github download the executable Python script posted in the README. Navigate to where the script is located, and double click on the file. After it runs, the server should be running on the virtual machine. In a browser, type in "localhost:8000" and you should see the Menu system (Refer to figure 3.1). You can type in anything to the box labeled "Topic" and you should see results displayed in the browser, as a list of clickable files. This is dependant on choosing a topic that exists in at least one document located in the Tobacco database. Clicking on a file that is displayed in the browser will display a list of the URLs for all of the files in each cluster. To locate the actual file, copy and paste a particular URL in your browser. This will redirect you to the website that hosts all of the files in the Tobacco database, and will display the file you have selected. The files will also be downloaded to the virtual machine that can be navigated to by going to the clusters_txt_files directory in the path /home/user1/Menu/TobaccoTestimonies/TobaccoTestimonies/app/.

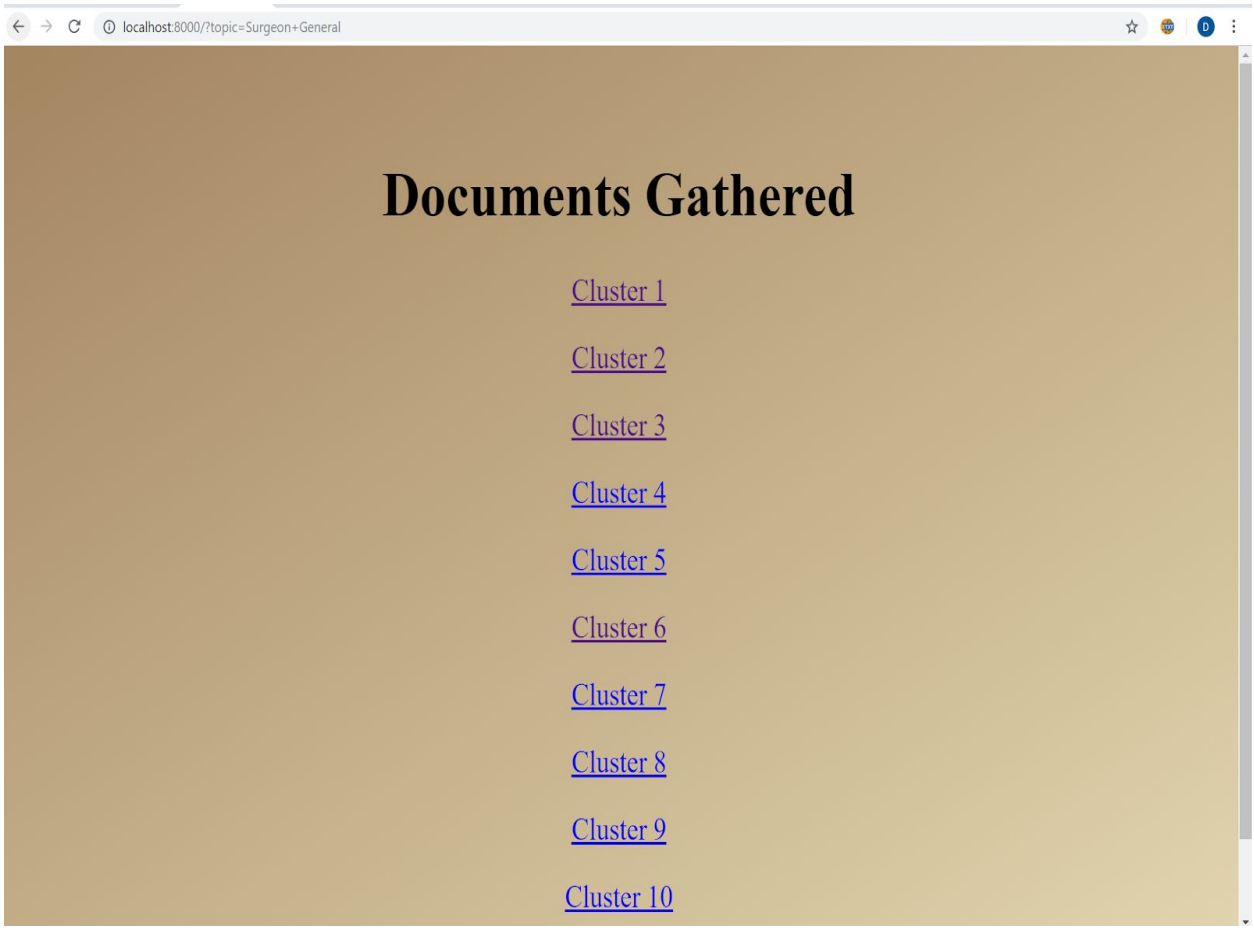


Figure 6.1: An example results page. Each of the links shown above, Cluster 1 for example will redirect your browser to a .txt file that holds the URLs for every file in a given cluster.

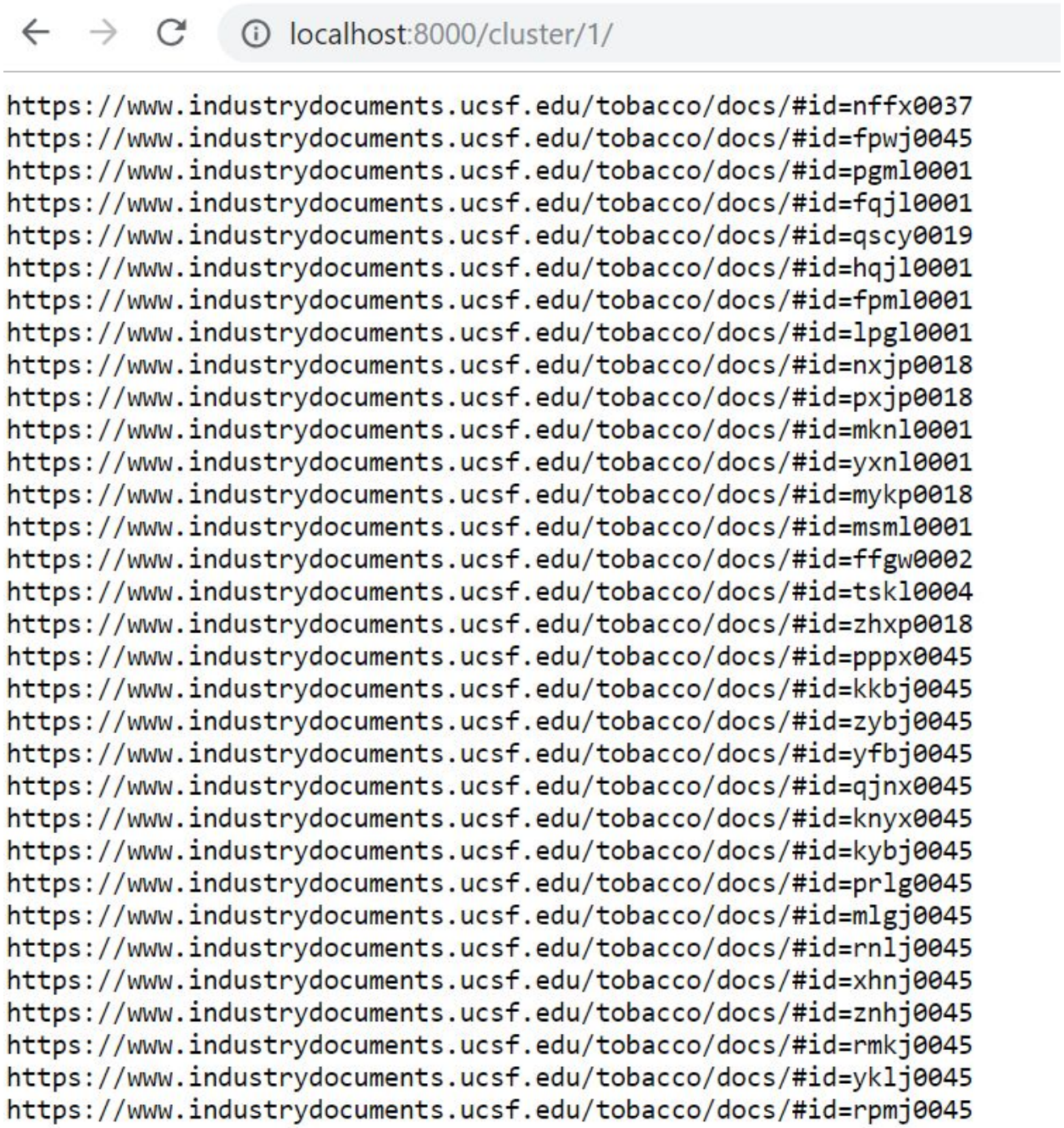


Figure 6.2: An example of what is displayed after clicking on one of the cluster links. Each one of these URLs when copy and pasted into your browser will display the file associated with a given ID.

If you would like to use this application while not connected to Virginia Tech's WiFi (Eduroam), you will need to set up a VPN connection through https://www.nis.vt.edu/ServicePortfolio/Network/RemoteAccess-VPN.html. You will first need to download 2-Factor Authentication. The Duo app is a great resource to establish 2-Factor authentication. You should follow the steps detailed in the above URL exactly before you attempt to log on to the virtual machine. Once you have your VPN setup you can run the script as explained above and will be able to use this application.

7. Developer's Manual

There are a lot of different ways that this system can be improved. All of the code is accessible through the virtual machine provided by Dr. Fox and the VT Computer Science department. There are four key components to this system. First there is the gathering documents for the fourteen million documents online released by Big Tobacco companies. Second is converting those documents into a usable manner so that they can be sent through the Doc2Vec model. Third is the Doc2Vec model which takes in a corpus and turns the documents into vectors. Lastly is the clustering component which takes the vector map created by the Doc2Vec model and then clusters the vectors based on their nearest neighbor.

You should start by gaining access to the github from Dr. Fox or Dr. Townsend. Once you have access please read through the README carefully, there will be a detailed explanation with links to all of the necessary files. The overall application is in Python and Django. If you navigate to the TobaccoTestimonies/ directory you will see two additional directories. First you'll see the app/ directory which contains all of code that runs our application. You will also see another directory named TobaccoTestimonies/ that will contain all of the settings for the Django application. You will also see a file named requirements.txt, if you do not have access to the virtual machine you will want to run "pip install -r requirements.txt" in that directory. This will install all of the required packages for the project. You may want to install a virtual environment in case any of the dependencies for this project intersect with dependencies for another project. If you choose to install a virtual environment you will want to wait until you have activated the environment to run the command to install all requirements. Once you have installed all dependencies you can start contributing to the existing code base. In the TobaccoTestimonies/ directory you will find all of the Django settings that the project depends on. In the app/ directory you will find all of the

code that creates the Menu system. In the app/ directory you will see two directories the static/ directory which contains all of the css for the project, and in the templates/ directory you will find the html for the project. There is also a forms.py file that contains the fields in the form. This form is created by the views.py file, which will render the html file in the templates/ directory and pass in the form it generated. The views.py file is also responsible for making the call to the API and handling the response it receives. You will also want to navigate to the manage.py file and run “python manage.py runserver” to be able to see the changes in the browser. As noted above in the User Manual, if you would like to access this application while not on Virginia Tech’s WiFi you need to establish a VPN connection by following the steps at https://www.nis.vt.edu/ServicePortfolio/Network/RemoteAccess-VPN.html. It is important to note that due to the size of the root/ folder that holds every file in the Tobacco database, this will not be included on the Github. If you would like to use this application on your personal machine you must download all of the files in the database to your machine. You will also need to change the static directory lookup that occurs in the app/views.py file. Most of these directory lookups reference exact paths that exist in the virtual machine. You will want to change all of these to look for the directories where you download all of these files.

As all of this data and code is stored on the VM you can also access it there. First to log in you ssh into tobacco.cs.vt.edu. For the safety of the system the password will not be divulged in this report as this report will eventually become public. If you are working on developing the system please contact Dr. Fox or Dr. Townsend for the password. Once on the system you will be logged into the root. I would recommend downloading a program like CyberDuck which will allow you to navigate through different folders and files and allow drag and drop. It helps because you don’t have to do everything through the terminal. If you navigate to user1 using the terminal command su - user1 it will log you into our main user. This is where we did all of our code development. If you open up two terminals you can access the Jupyter notebook where our Python scripts are stored. In the terminal logged into user1 you type in “jupyter notebook --no-browser --port=8080”. This tells your terminal you do not have a browser. Then in the other terminal you must tunnel your ssh so that you can pull up the jupyter notebook locally. This can be done by typing in the command “ssh -N -L 8080:localhost:8080 user1@tobacco.cs.vt.edu”. Then by typing in localhost:8080 on your local web browser and typing in the token password you can access all of our scripts. From there you should be able to do any development needed. The two terminals will not allow you to access them without cancelling the processes which will shut down the jupyter notebook. User1 has also been added to the list of sudoers so anything you might need to download is possible through user1.

The main files that we work with are Birch_Clustering.py, Doc2Vec_Model.py, and docGather.py. There were other files we worked with like Doc2Vec_Implementation and PDF_to_Text_File_Conversion, but these files serve a more specific purpose and are not a part of the overall system. We used the Implementation file to work with checking how the model trained and the second file to turn pdf documents into text files that the model could then use. Now for the three main documents, we will start with docGather.py.

```
import sys
import shutil
import os

def getDocs(docIdList):

    dirpath = os.getcwd()
    print(dirpath)
    pathLoc = '/home/user1/dump'
    shutil.rmtree(pathLoc)

    try:
        if not os.path.exists(pathLoc):
            os.makedirs(pathLoc)
    except OSError:
        print('Error creating directory')

    j = 0
    for docId in docIdList:
        print(docId)
        id1 = docId[0]
        id2 = docId[1]
        id3 = docId[2]
        id4 = docId[3]

        path = '/home/user1/root/' + id1 + '/' + id2 + '/' + id3 + '/' + id4 + '/' + docId
        print(path)
```

```
        exists = os.path.isfile(path + "/" + docId + ".ocr")
        if exists:
            shutil.copyfile(path + "/" + docId + ".ocr", pathLoc + "/" + docId + ".ocr")
        else:
            print(path + " not found")

        #docFile = open(path + "/" + docId + ".ocr", "r")
        #docFile.close()

docList = ['njmp0006', 'txhx0228', 'zznn0003']
getDocs(docList)
```

No we will talk a little bit about the Doc2Vec_Model.py file.

```
#set up logging information
#this is helpful for working with the model but if the infomartion gets to long commenting this out can be helpful
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
```

```
#import the gensim model
#Used the information here https://medium.com/@mishra.thedeepak/doc2vec-simple-implementation-example-df2afbbfbad5 to learn
#how to set up the model
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from nltk.tokenize import word_tokenize
from gensim.test.utils import common_texts
#NLTK is helpful for working with human language data. easy to use interface
#for classification, tokenization, stemming, tagging, parsing, semantic reasoning, etc
import nltk
#This allows you to use the operating system inferface. Allows you to work with the
#OS that python is running on
import os
#Techniques to match specified patterns according to Unix rules.
import glob
#Allows for errno system symbols. Helps with translating numeric errors to messages
import errno
#regular expression. this module provides full support for regular expressions.
#they are a special sequence of characters that help you match or find other strings or sets of strings
import re
#this is the body we will use to store all of the data
corpus = []
```

From the image above you can see that we did a lot of importing to be able to use different features and functions throughout the file. We also imported some logging at the very top to help identify our issues better.

```
path = '/home/user1/dump/*.ocr'
#glob.glob return list of path names as string
files = glob.glob(path)
fileIDS = []
corpus2 = []
#https://www.quora.com/How-do-I-read-mutiple-txt-files-from-folder-in-python idea to loop like this to read in files came fro
#loop through all of the files in the folder
for name in files:
    try:
        with open(name) as f:
            fileIDS.append(name[17:len(name) - 4])
            #had an issue with characters not being identified as ascii so use the idea here https://stackoverflow.com/questi
            final = " ".join(re.findall(r"[a-zA-Z0-9]+", f.read()))
            token = word_tokenize(final)
            #add the text to the corpus
            corpus.append(final)
            corpus2.append(token)
    except IOError as exc:
        #error handling
        if exc.errno != errno.EISDIR:
            raise
#Tagged Document is a single doc made up of words(unicode string tokens) and tags(list of tokens).
#Tags may be one more more unicode string tokens but typical practice for tags list is to include a unique integer id as the
#format of TaggedDocument(namedtuple('Tagged Document', word tags))
#enumerate adds acounter to an iterable. So we loop through the corpus tokenizing its value.
```

In this next picture we began by getting the file path with all of the documents we want clustered and setting up where the corpus is going to be saved. We then loop through each file, reading them in, and storing them in an array.

```
tagged_data = [TaggedDocument(words=word_tokenize(_d.lower()), tags=[str(i)]) for i, _d in enumerate(corpus)]
#tagged_data = [TaggedDocument(words=word_tokenize(doc, tags=[str(i)]) for i, doc in enumerate(common_texts)]
#tagged_data = [TaggedDocument(doc, tags=[str(i)]) for i, doc in enumerate(common_texts)]
#preferred number of passe
max_epochs = 100
#dimensionality of word vectors
vec_size = 20
#alpha is the intial learning rate while min_alpha is what he learning rate drops to as training progresses
alpha = 0.025
#used for trainng and evaluating the network
#dm defines the trainng algorithm. because its 1 we are using distributed memory (PV-DM)
model = Doc2Vec(tagged_data, vector_size = vec_size, alpha=alpha,
                min_alpha = 0.00025, min_count=1, dm=1)
#build vocabulary from as equence of sentences
model.build_vocab(tagged_data)
for epoch in range(max_epochs):
    print('iteration {0}'.format(epoch))
    model.train(tagged_data, total_examples=model.corpus_count,
                epochs=model.iter)
    #decrease the learning rate
    model.alpha -= 0.0002
    #fix the learning rate, no decay
    model.min_alpha = model.alpha
    #save the model
model.save("doc2vecNew.model")
print("Model saved")
```

Now the fun part begins. We then begin to tag these documents in the same order that they were read in. Next we start to begin training and building a useable vocabulary for the model to train on. We can also set different things for the model to train on for example: the training algorithm itself, the minimum learning rate, vectors, etc. Once the model has be trained we save the model so that other files and load the same model. Now we will talk about Birch_Clustering.py.

```
from gensim.test.utils import common_texts
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from nltk.tokenize import word_tokenize
import codecs
import nltk
import os
import glob
import errno
import re
#this is the body we will use to store all of the data
corpus = []
#filepath to the folder with txt files
#this needs to be where all the files pulled from the API are stored
path = '/home/user1/dump/*.ocr'
#glob.glob return list of path names as string
files = glob.glob(path)
fileIDS = {}
corpus2 = []
#https://www.quora.com/How-do-I-read-mutiple-txt-files-from-folder-in-python idea to loop like this to read in files came fr
#loop through all of the files in the folder
count = 0
for name in files:
    try:
        with open(name) as f:
            fileIDS[count] = name[17:len(name) - 4]
```

This is the beginning of the file. First we do all of our importants and then we set up arrays to contain data as well as file paths. When the documents are saved in the dump folder they are of type ocr so we want to import all of their names to begin accessing them.

```
count = 0
for name in files:
    try:
        with open(name) as f:
            fileIDS[count] = name[17:len(name) - 4]
            #had an issue with characters not being identified as ascii so use the idea here https://stackoverflow.com/questi
            final = " ".join(re.findall(r"[a-zA-Z0-9]+", f.read()))
            token = word_tokenize(final)
            #add the text to the corpus
            corpus.append(final)
            corpus2.append(token)
    except IOError as exc:
        #error handling
        if exc.errno != errno.EISDIR:
            raise
    count += 1
model = Doc2Vec.load("doc2vecNew.model")


X=[]
start_alpha=0.01
infer_epoch=100
for d in corpus2:
    X.append(model.infer_vector(d, alpha=start_alpha, steps=infer_epoch))

from sklearn.cluster import Birch
```

In this next photo we see that we are reading in all of the files and tokenizing them and storing them into an array. We do this because once we load the Doc2Vec model we are then going to do infer_vector on it with these tokens. This data will then get saved in an array we called X. Lastly we begin to import sklearn and Birch.

```
num_clusters = 10
brc = Birch(branching_factor=50, n_clusters=num_clusters, threshold=0.1, compute_labels=True)
brc.fit(X)

clusters = brc.predict(X)

print ("Clusters: ")
print (clusters)

cluster_ids = []

for i in range(num_clusters):
    c = []
    for j in range(len(clusters)):
        if clusters[j] == i:
            c.append(fileIDS[j])
    cluster_ids.append(c)

clus_count = 0
for x in cluster_ids:
    print("Cluster {}:".format(clus_count))
    print(x)
    clus_count += 1
```

In this photo we see that we begin using Birch with a set branching factor, number of clusters, and threshold. Next we fit X into it and begin predicting with X and recording the results in the variable clusters. Once complete with that we loop through and print our file in order and what cluster they are in as well as printing by each cluster and printing the names of the files in each cluster. We did this to help display where

documents were being clustered so we could check them as well as then provide that data to be displayed back on the menu.

8. Lessons Learned

8.1 Timeline

One of the issue we came across this semester was issues with timeline. It can be hard to plan out an accurate schedule in the beginning of a semester for a project that you don't have much experience in. However, if you can make an accurate schedule it can be extremely beneficial. We tried to create a plan in the beginning of the semester but it was not nearly as accurate as we would have liked. We underestimated how difficult working with machine learning could be. Luckily to try and solve this dilemma we met with Dr. Fox and he helped to set us on the right course. Another thing we didn't account for was people's time. We didn't factor in team members having other school work, other plans, vacations, spring break, etc. This threw off our schedule a bit as well because sometimes it would feel like we kept stopping and starting and it wasn't one continuous flow of work.

Throughout the semester we began to get better at managing our schedules and working better with our time. Once we got into a rhythm it became much easier to knock out different objectives and deliverables. We started to get accustomed to people working on different sections of our system and then being able to link them together and explain how they work. This level of concurrency helped us to speed up our timeframe and develop much more of the system.

8.2 Lack of Expertise

There is a lot going on with this project, a lot of which we haven't been exposed to throughout our education. It was difficult to make a plan of action at the beginning of

the project because there was such a huge learning curve for all of the team members. Subsequently we spent a lot of time deep diving into different topics with the goal to ultimately bring it all together in the end. This created confusion along the way because there were points in time where we all disagreed on what our solution should be. Since our goal at the beginning was to learn our respective topics we didn't spend enough time face to face. It started to become very unclear of where we were headed as a team, and it wasn't until the midway point that we all got on the same page and started to make significant strides towards a common goal.

Another issue with lack of expertise was a constantly shifting solution. We have tried to get a good basis of the topics needed for our solution but there is so much to learn it has been difficult to stay true to the technologies we have stated we would use. Since as stated previously we are working in parallel and attempting to bring it all together at the end due to how much is involved in our project. So each time one team member had to change their technology it impacted the work everyone was doing and ultimately became a set back. It would have been much more beneficial if we had a team member who was knowledgeable in the areas involved with our project. Since we did not there have been some expected stumbles along the way.

However, throughout the semester through trial and error we began to improve on our concurrency. We talked with Dr. Fox a couple times about our plans and timeframe and he was very willing to give advice. Through those and getting into a rhythm we began to work much better. We each started working on different parts of the system and improved at transitioning our parts together and being able to assist in the explanation of how each part worked.

8.3 Solutions

Some of our issues arose during the beginning of the project because we were unsure of how to piece everything together. Part of the solution to many of our issues was talking to Dr. Fox. He sat down with us and helped us fill in some of the holes in our project and guide us to how we could make a usable system. The most useful information we gathered from this meeting was the recommendations of frameworks that we should use to help us finish this project on time. One of these frameworks we are going to use is called Gensim. Gensim is a framework used to help us with the word2vec and doc2vec aspects of the project.

Sitting down together throughout the semester helped us to work through different issues and to create solutions to our problem. One example would be working with the Annoy Indexer. We were having issues getting that involved on our system so we started working in groups trying out different options. We had a couple people trying out sklearn and using that to try and cluster the data while other team members continued working trying to get the Annoy Indexer to work. We also had issues working with the API and trying to gather documents. This slowed us at the start because we were trying to work on everything incrementally and wait for each section to be finished before working on the next. We then divided and conquered the work we needed to complete. We had some people working with the Doc2Vec model and testing it with a couple pre-downloaded documents. Meanwhile we also had others working on different solutions to try and work around the API that wasn't working as we had planned.

8.4 Future Work

This was a system created under time and resource constraints so there can definitely be improvements. If we had more time and resources to continue this project we believe there could be improvements in the UI and how accurate the model is. By UI I mean improving the way the system is used. Currently it is accessed through a VM and we were thinking that if possible improving how the model is accessed would make it much better and easier to use. We also think that spending more time training the model and making it more accurate could also be a thing that we would like to see improvement in. The more accurate the model is the more information can be learned from this documents.

We have started creating a usable Django application that would make it easier for the average user to access the application. We have created a bare bones application, that is not especially interactive. Ideally, the Django application would allow a user to choose a trained model from a dropdown to run the clustering with. It would also allow the user to make more specific queries to the API. This would mean allowing the user to input more fields on the homepage of the Django application such as author, dates, etcetera. We would also like to see some visuals displayed on the Django application. It would be possible to add graphs, and various visual tools that could assist in research with the clustered files.

Due to the large amount of data associated with this project it was difficult to provide a solution that could work on any machine. The project is very dependant on

our virtual machine, and it would be ideal to let this application become public, but to do so there would have to be work done to find each document in a way that could be done on any machine. As of now, all of the contents of the files are downloaded onto the virtual machine to be accessed. There are so many documents, it is impossible to check them into GitHub and allow users to get them on their machines quickly. Finding a way for users to access these files without having to undergo this large of a download would be ideal.

In order to scale this application it would also be necessary to optimize the program. The program makes a lot of API calls since the provided Tobacco API only serves up 100 documents at a time. It also must write to a lot of files, create directories, and perform document lookups for all of these documents. For a query that responds with 100,000 documents it is almost impossible to run this program, and if it did finish executing it would take a substantial amount of time. Given that users will not be willing to wait days, possibly even weeks to receive their results it would be extremely important to make this application execute faster.

9. Acknowledgments

We would like to acknowledge our client, Dr. Townsend, for his passionate research into the deceptive tactics that big tobacco companies have been using for years. He has both a personal and professional reasoning behind his research, and his dedication to this project goes to show how important this research could be to recognizing these same practices taking place in other large industries. Dr. Townsend can be reached at his email, dtown@vt.edu.

We would also like to thank the members of our class for providing input and constructive comments on how to improve our project and its processes.

We would also like to thank all of the GTA's, especially Saurabh Chakravarty, that provided technical insight and pointers on how to develop word processing software, which a new area to all members of our group. Mr. Chakravarty can be reached at saurabc@vt.edu.

Lastly, we would like to warmly thank Dr. Fox for working with us when we struggled with the learning curve of the new software, and helping us decide the best architecture to deliver the most useful product possible to our client.

10. References

1. Glantz, Stanton. *Industry Documents Library*, 2002, www.industrydocumentslibrary.ucsf.edu/tobacco/.

2. “Google Code Archive - Long-Term Storage for Google Code Project Hosting.” *Google*, Google, 29 July 2013, code.google.com/archive/p/word2vec/.

3. “Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining.” *ACM Digital Library*, Association for Computing Machinery and Morgan & Claypool, 2016, dl.acm.org/citation.cfm?id=2915031.

4. Gensim: Topic modelling for humans. (n.d.). Retrieved April 25, 2019, from https://radimrehurek.com/gensim/models/doc2vec.html

Used these resources in the creation of the Doc2Vec model

5. KDnuggets. (n.d.). Retrieved April 25, 2019, from https://www.kdnuggets.com/2018/04/implementing-deep-learning-methods-feature-engineering-text-data-cbow.html

6. Doc2Vec Document Vectorization and clustering. (n.d.). Retrieved April 25, 2019, from http://techscouter.blogspot.com/2018/08/doc2vec-document-vectorization-and.html

7. Text Clustering with doc2vec Word Embedding Machine Learning Model. (2018, October 04). Retrieved April 25, 2019, from http://ai.intelligentonlinetools.com/ml/text-clustering-doc2vec-word-embedding-machine-learning/

8. Qaiser, R. (2017, May 12). How to Extract Words from PDFs with Python. Retrieved April 25, 2019, from https://medium.com/@rqaiserr/how-to-convert-pdfs-into-searchable-key-words-with-python-85aab86c544f

Used this to help create the pdf to text file script

9. Mishra, D. (2018, March 17). DOC2VEC gensim tutorial. Retrieved April 25, 2019, from https://medium.com/mishra.thedeepak/doc2vec-simple-implementation-example-df2afbbfbad5

Used this source in the creation of the Doc2Vec Model

10. Doc2vec tutorial. (n.d.). Retrieved April 25, 2019, from https://rare-technologies.com/doc2vec-tutorial/

11. Django (Version 1.5.12) [Computer Software]. (2013). Retrieved from https://djangoproject.com.

12. Python (Version 2.7) [Computer Software]. (2008). Retrieved from http://www.python.org.

13. Scikit-learn Birch Algorithm [Computer Software]. (2018). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.Birch.html

14. AnnoyIndexer Most Similar [Computer Software]. (2018). Retrieved from https://radimrehurek.com/gensim/similarities/index.html