Multimedia, Hypertext, and Information Access
AWS Tobacco Settlement Retrieval

Authors
Anamol Sitaula
Nishan Pokharel
Douglas Bossart
Aditya Kanuri
Abhinandan Mekap
Rahul Ray

Instructor
Dr. Edward A. Fox

Publisher
Virginia Tech

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
May 6, 2020

Table of Contents

Appendix

List of Tables

List of Figures

1. Executive Summary

The Tobacco Industry is one of the largest and most influential industries. It has spent hundreds of millions of dollars on advertising and marketing tactics to ensure dominance and control in the economy. This is especially evident when considering tobacco settlement cases where the enormous power and influence of the Tobacco Industry has allowed them to develop key strategies and tactics for trials and settlement cases over the past century. Our client Dr. Townsend is currently researching the tactics and inner-workings of the Tobacco Industry over the past few decades to expose the marketing and legal strategies as well as the key players who have been influential in the Industry. Dr. Townsend is utilizing the "Truth Tobacco Industry Documents", a library of documents created and facilitated by the UCSF Library for research purposes. Our project is meant to further enable researchers specializing in business, public health, law or computer science, who will benefit from easier access to tobacco settlement related documents, with enhanced search capabilities, extending the work of the Fall 2019 CS5604 Information Retrieval teams.

We studied the 14 million tobacco related documents from UCSF. We improved upon the indexing of the roughly 8000 depositions, to support line-wise as well as page-wise indexing. We modified and updated existing Python scripts to output the results in the required JSON format, and then pushed the documents into ElasticSearch. Furthermore, we also created another tobacco index and added another 3 million tobacco files to this index. All testing and evaluation work was done using Python scripts. We used the existing Kibana tool for the visual representation of the data.

2. Introduction

The current team is expanding on the work done by the Fall 2019 CS5604: Information Retrieval teams, forming an enhanced information retrieval system that will streamline and aid Dr. Townsend in his research regarding the Tobacco Industry. As mentioned earlier, the source of the documents is the "Truth Tobacco Industry Documents" supported by the UCSF Library. Thus, all outlined deliverables, and design decisions, are made to support the needs of Dr. Townsend. Thus, the developers will facilitate processing of the millions of settlement documents through various Python scripts to convert the OCR documents to the proper JSON format and enable searching through these documents via the GUI interface in the form of Kibana.

2.1 Objective

Our objective is to enhance the existing design for the search engine/retrieval system already developed by our predecessors. In Fall 2019 CS5604: Information Retrieval, teams worked to aid Dr. Townsend in locating and utilizing documents, to identify useful data in an efficient and effective manner [1]. Implementing line-wise indexing for Deposition Documents would streamline research efforts as researchers such as Dr. Townsend and future users would be able to locate the lines of interest, thus finding specific entries in addition to whole pages of long documents. Thus, when examining thousands of documents, the burden of effort would be reduced for users. Therefore, the developers formed the first primary objective towards the line indexing of the roughly 8,000 deposition documents. In addition, another primary goal is to provide access to all of the potential documents in the UCSF library as each of these documents can be critical and important for research requirements. The developers will try to finish the indexing of the metadata, possibly also supplemented by the fulltext associated with the documents contained in the UCSF library. We will achieve this through the use of several tools and software routines, including but not limited to Python, Kibana, and ElasticSearch. Additionally, we need to properly document all our work to ensure ease of use for future users, developers, and researchers. We should improve the search engine/retrieval system, thus helping all potential users and clients.

2.2 Client

Our primary client is Dr. David Townsend, a professor in the Department of Management at Virginia Tech. Dr. Townsend conducts research on technology-oriented companies. He focuses on areas such as capital acquisition processes, growth strategies, organizational development, and CEO decision-making. Currently, he is conducting research on the Tobacco

Industry. He wants to utilize the collective documents in the UCSF library to view trends related to practices of the Tobacco Industry in settlements and litigation. Dr. Townsend is currently conducting research on important cases in regards to the Tobacco Industry, analyzing individuals and their roles in related cases.

Our other client is Satvik Chekuri. Mr. Chekuri's MS thesis relates to one part of this effort. Mr. Chekuri is specifically focused on depositions.

2.3 Potential Users

All design decisions and documentation related activities are based on formed personas of expected users who will be interacting with our system. It is crucial to identify the demographics of individuals who utilize the application as the design and functioning of our Search Engine/Information Retrieval system is meant to be more accessible and efficient than the system provided by the UCSF library. Thus, the division of the intended users was formed to be the researcher persona and developer persona.

The first persona should be a researcher utilizing the Search Engine/Information Retrieval system for data collection purposes. The researcher is an individual that has some experience with basic programming and has a good understanding of using technology, but does not understand the full design of our system. The researcher can easily grasp the functionality of the GUI representation through the Kibana API due to previous experience with technology. The researcher will need a well-documented User Manual that goes over a step-by-step process to access the GUI tool, major components of the project, and related basic commands, as well as a general summary on the inner workings of the various parts of the implemented system.

The second persona is a developer who is going to modify the existing system implementation. Thus, the incoming developers will need to have documentation on all design choices made, thorough tutorials on all aspects of development and related program files, as well as access to the Virtual Machine and file system. The developer is an individual with experience in one or more of the technical skills and languages. Thus proper documentation of all important programming files and structures, is essential for the continuation of work. Just as our group has been able to understand, use, and expand on the work of the CS5604 class, the next group of developers working on this project should be able to do the same with our changes and additions.

2.4 Challenges

We had several challenges for this project. These include time, working remotely, and programming knowledge.

Another constraint was understanding the layout and location of the data in the VM. The developers easily located the deposition documents but extra time was needed to determine the locations of the other 14 million documents. We had to explore many of the files and understand the layout the developers intended to use.

Time was another constraint. Previously, when all the team members were on campus, we could only meet a few times every week. Designating time to meet to plan out the project and designating time to come together and work on the project was challenging. In addition, having only one semester to work on such a complex project was a large challenge.

Programming knowledge is another constraint. Not all members of the group have had experience working with Python. No one in our team had experience with Kibana and ElasticSearch. Having to learn how to use these was a major constraint.

Finally, an unexpected challenge was the COVID-19 lockdown that changed how the group would meet and also changed requirements such as all in-person events being canceled.

3. Requirements

Dr. Townsend wants the ability to search through a large amount of tobacco related documents. To achieve this, we need to complete several requirements.

The first deliverable that we are prioritizing is to write the scripts to be able to implement line-wise indexing on the roughly 8000 deposition documents. This can only happen after searching and tagging documents as being a deposition document or not. When all the documents are tagged, a script will run on those tagged as a deposition document, and line-wise indexing will be possible on each of those documents.

For the next deliverable we will be focusing on the metadata, and finish page-wise indexing for the remaining 14 million tobacco related documents. The script for page-wise indexing was completed by the CS5604 students so we would just need to run that script on these documents. We will need to figure out a way to do this efficiently as the CS5604 students were not able to complete this due to a time constraint.

The final deliverable is to complete the proper documentation of all our work and accomplishments. We need to complete the documents with the assumption that a new group can take this project on with no knowledge of the project at all. We will also make sure that our client, any new user, or a new research group that takes on this project will be guaranteed ease of use regarding the search engine/retrieval system.

4. Design

4.1 Introduction

After our initial meetings with our client, we found out that there weren't any set guidelines for our first deliverable on how the line-wise indexing should be formatted in the JSON structure. This left us with a lot of flexibility on how we could approach the design. Our client gave us access to the CS container cluster so that we could see the structure of the current page-wise indexing to give us a better idea on how to approach the line-wise. The only requirement for the line-wise indexing was when a user tries to search for a keyword in a deposition document they need to be able to get a reference to the line that the word was in to easily find it in the document. Our task was to create a structure that would make this easy for the user to read as well as be efficiently stored for later reference.

4.2 Technology Used

All of the code for the page-wise indexing was written in Python and the output files are all written to JSON formats. The page-wise code reads in .ocr files and converts them into JSON files. We decided to just add on to the existing page-wise code, so we coded the line-wise indexing in Python as well.

For the handling of all the documents, we used ElasticSearch, which provides a full text search engine [2]. ElasticSearch has a great platform to store all the enormous amounts of data in the 14 million tobacco related documents and allow them to be searched and indexed. The technology makes use of an inverted file index allowing it to retrieve search results quickly and efficiently.

A good platform to utilize to make ElasticSearch more user friendly and have all the features it provides to be accessible and useful is Kibana. Kibana is a front end dashboard for ElasticSearch that provides more in-depth information such as visualizations and advanced data analysis [3]. With Kibana, we are able to search, view, create data charts and tables, and index all the documents we are working with. This is extremely helpful when trying to give presentations on the project to give the audience a better understanding of what we are doing with in-depth visualizations.

4.3 Project Design

The original design of the Python file was to go through the OCRed documents and store the contents of the page as a string separated by spaces as well as the page number in a dictionary. If the document had multiple pages, there would be a list of these dictionary objects. These would be stored in a JSON format.

For the line-wise indexing we decided to preserve this structure and replicate it. We decided to add another level to the dictionary. The page dictionary would now contain another key that would be mapped to a list of line dictionaries. Each line dictionary would contain a line number as well as the contents of that line. We did this because we wanted the user to be able to see the entire line with the word(s) that they were searching for. When the user would search for specific keywords, they could receive the line number along with the contents of that entire line, as opposed to just the line number. Otherwise, if the user just received the line number they would again have to go back through the entire document to find that line. We made the decision to have the extra overhead, in favor of making the user experience easier.

4.4 Group Roles

The roles that our group members held were dynamic throughout the semester as our project and circumstances changed around us. Through good communication between team members, we were always able to focus our efforts in different areas when the time came, regardless of our respective roles. We also found that we were able to get more work done when we met up as a full team and tackled problems together.

After spring break, thanks to the new circumstances of our isolation, we took stock of where we were with our project, as well as what new roles might need to be assigned to keep things progressing smoothly. We added two more columns to our chart of roles, “Division of Objective” and “Alternative Role”. See Tables 1 and 2.

4.4.1 Division of Objective

This category splits the team into two groups, those with the objective of finishing “Deposition Documents” and those tasked with finishing the “Indexing” part of our project.

4.4.2 Alternative Role

This category was aimed at splitting up tasks that would help us stay organized, efficient, and up-to-date with our project as we entered into the next phase of our semester in isolation. The responsibilities include “Communications”, “Report and Presentation”, and “Task Organization”.

Table 1: Initial Roles

Name	Primary Developer Role	Secondary Developer Role
Rahul Ray	ElasticSearch	Kibana
Anamol Sitaula	Kibana	ElasticSearch
Nishan Pokharel	AWS	DevOps
Aditya Kanuri	Python	Testing
Abhi Mekap	Testing	Python
Douglas Bossart	DevOps	AWS

Table 2: Updated Roles as of 3/28

Name	Primary Developer Role	Secondary Developer Role	Division of Objective	Alternative Role
Rahul Ray	ElasticSearch	Kibana	Finish Indexing	Communications
Anamol Sitaula	Kibana	ElasticSearch	Deposition Documents	Report & Presentation
Nishan Pokharel	AWS	DevOps	Finish Indexing	Task Organization
Aditya Kanuri	Python	Testing	Deposition Documents	Report & Presentation
Abhi Mekap	Testing	Python	Finish Indexing	Task Organization
Douglas Bossart	DevOps	AWS	Finish Indexing	Team Leader

4.5 Project Plan

We have accomplished the goal of completing our first deliverable of implementing line-wise indexing on the roughly 8000 deposition documents with the script that we wrote. We will continue to work on our second deliverable of trying to index the rest of the 14 million documents page-wise. We have the script; however, we are trying to find an efficient way to do this as the students tasked with this before us had a time constraint and weren't able to finish. We may be facing the same problem. We are also having trouble gathering all of these 14 million documents and trying to get them in the same place. We will be working with our group advisor on how to do this as the layout structure of all the files is hard to understand.

4.6 Project Timeline

Our first timeline was created in preparation for our Presentation 1 assignment in early February. We had the timeline laid out as in Table 3 with the described deliverables.

Table 3: Timeline for Project Milestones

Deadline	February 18th	February 26th	March 7th	March 25th	April 8th	April 22nd
Description	**Background** - Document - Present	**Preparation** - Research - Learn Software	**Progress Check** - Reevaluate if necessary	**Final Stretch** - Tie it all together - Take on new challenges	**VTURCS Prep** - Presentation - Documentation	**Finish Line** - Final Report

Background (2/18)

- Read all relevant documentation
- Assign subteams for major guidelines
- Begin documenting self-guided research on the technology stack
- Brainstorm ideas of possible implementations

Preparation (2/26)

- Complete all relevant research
- Begin developing test cases to test OCR record line tabulation
- Start development of the reviewed OCR algorithm for line indexing
- Document all work on shared Google document

Progress Check (3/7)

- Achieve the goal of at least 25% of new OCR for line indexing
- ElasticSearch and Kibana developers will work on individual roles
- Tester and DevOps role will examine documentation
- Kibana will report group adequacy through examination of goals that were met

Final Stretch (3/25)

- Approximately 60% of original deadlines should be met
- Work towards development of Interim Report
- AWS developer begins transition to AWS
- Design additional deliverables based on user interest and project need

VTURCS Prep (4/8)

- Approximately 90% of original deadlines should be met

- Review all documentation and format necessary reports on project goals
- Begin preparing for VTURCS
- Prepare for Final Presentation

Finish Line (4/22)

- Completion of remaining work is first priority
- Both documentation and abstracts should be completed and ready
- Original deadlines should be completed
- Group will begin developing presentations
- Practice speeches for presentation

As we progressed through our milestones, we took two factors into account when determining our successes: first, how easily we were keeping up in our efforts to meet our milestones, and second, our changing perception of what aspects of the project were most important and most time consuming.

5. Implementation

5.1 Line-Wise Code

```
def file2Json(filename):
  if (len(filename) <= 8):
    return None
  if not os.path.isfile(filename):
    #logger.warning(filename + " does not exist")
    return None
  pages = {}
  pages["text_content"] = []
  with open(filename, 'r') as fp:
    lines = fp.readlines()
    #if len(lines) < 25:
    #  logger.warning("error format for file" + filepath)
    pageNum = 1
    currentPageNumString = pageNum #"{:0>5}".format(pageNum)
    nextPageNumString = pageNum+1 #"{:0>5}".format(pageNum+1)
    pageContent = u""
    match = False
    page = {}
    line_dict = {}
    curr_line_num = 1
    next_line_num = curr_line_num + 1
    page["line_content"] = []
```

Figure 1 - Line-wise code 1

In Figure 1, the set-up for the page-wise and line-wise indexing can be seen. The pages dictionary is created which is the dictionary that will contain a list of all of the page dictionaries. Then the page dictionary is also created, which is the dictionary that will hold the actual contents of the page as well as the page number. Next the line_dict is created which will hold the list of the line dictionaries. The line and page numbers were also initialized here.

```
for line in lines[0:]:
  #line1 = {}
  line = line.strip()
  line = re.sub(r'"', '', line)
  line = line.lstrip(string.punctuation)
  line = line.replace('\t', ' ')
  try:
    line.decode()
  except UnicodeDecodeError:
    continue
  res = re.match('^25', line)
  if (res):
    match = True
  if (len(line) == 0 and match) or 'pgNbr' in line :
      page["page"] = currentPageNumString
      page["content"] = pageContent
      pages["text_content"].append(copy.deepcopy(page))
      pageNum += 1
      currentPageNumString = nextPageNumString
      nextPageNumString = pageNum+1
      curr_line_num = 1
      next_line_num = curr_line_num + 1
      pageContent = ""
      page["line_content"] = []
      line_dict = {}
      match = False
```

Figure 2 - Line-wise code 2

Figure 2 shows the actual process of going through each file. Lines is a list of all of the lines in the OCR file. Each page in a deposition document has to be 25 lines long, so if the line has 25 in it and the line is empty, then we effectively know that we have reached the end of a page in the document. If this is the case then we update the page dictionary with the current page number, along with the content of the page, and then add this to the pages dictionary. We also update the page counter and reset the line number counter for the next page.

```
    else:
      try:
        if (line.isdigit()):
          continue
        pageContent += line
        pageContent += ' '
        line_dict["line_number"] = curr_line_num
        line_dict["content"] = line
        #added content
        if (len(line) > 0 and line[0].isdigit()):
              if (len(line) > 1 and line[1].isdigit()):
                line_dict["actual_number"] = int(line[0:2]
              else:
                line_dict["actual_number"] = int(line[0])
        else:
            line_dict["actual_number"] = None
        page["line_content"].append(copy.deepcopy(line_dic
        line_dict["content"] = []
        #line_dict = {}
        curr_line_num = next_line_num
        next_line_num = curr_line_num + 1


      except UnicodeDecodeError:
        continue
return pages
```

Figure 3 - Line-wise code 3

Figure 3 shows the logic for the line-wise indexing. If the line is not the last line in the document, the line_dict is updated with the contents of the current line and the line number. The next part of the code will be explained in Section 5.3. After that the line_dict is copied into the list of the line dictionaries in the larger page dictionary and reset for the next line. The line numbers are then updated and the code moves on to the next line.

5.2 Output

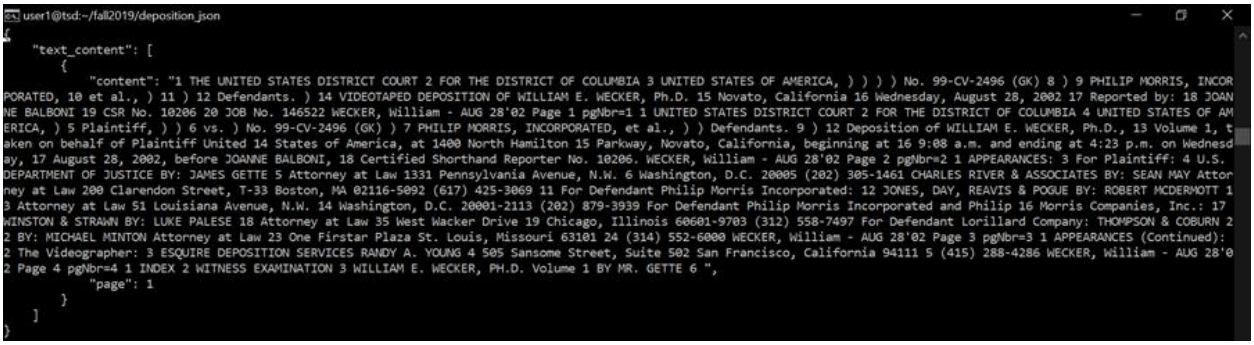

Figure 4 - Page-wise indexing output

Figure 4 shows the output of the page-wise indexing code. The file will contain a dictionary with text_content, which is a list of page dictionaries. As can be seen, each page dictionary is formatted so that the text of the page is listed along with its page number.

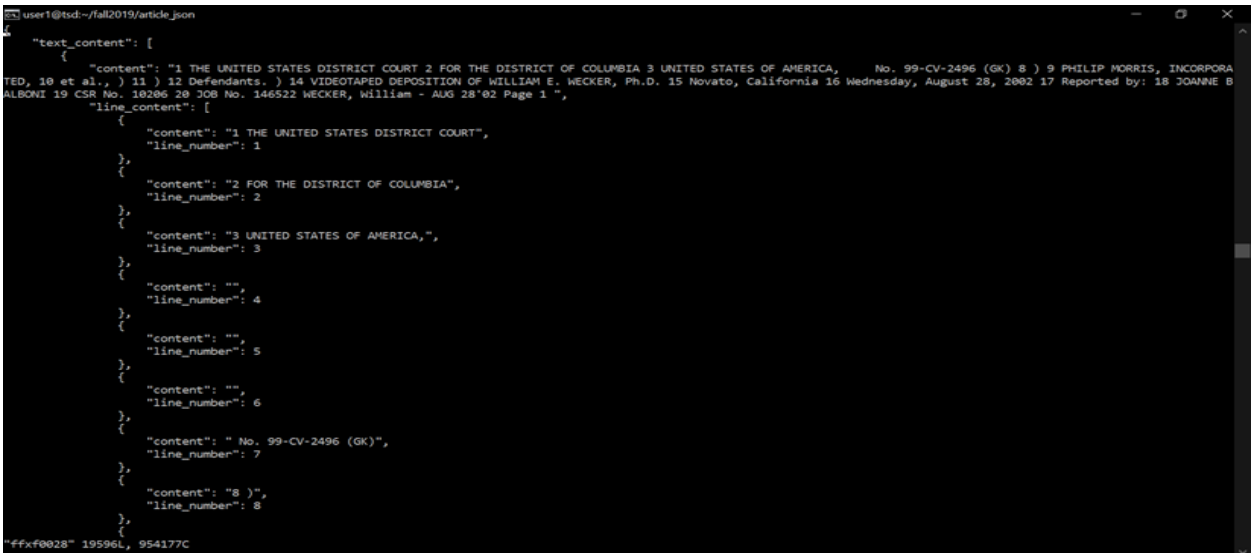

Figure 5 - Line-wise code output

Figure 5 shows the output when the OCR documents are run through our new line-wise code. It adds another key to each page dictionary, whose value is a list of the lines of each page with their corresponding line numbers.

5.3 Improvements

One important improvement that we made was handling the inconsistent line numbering of the OCR documents. In the OCR documents, there were a lot of blank lines or skipped lines that were used to space out the deposition documents. However, this caused us some problems when we ran them through our line-wise indexing code. This is because the lines that were left blank or skipped would be registered as a new line by our algorithm so it would assign these lines a line number. This became a problem because in some documents our assigned line numbers would no longer correspond to the actual line numbers that the OCR picked up. To account for this we decided to add another key to the line dictionary that would be the actual line number. This way when a user searches for a keyword they can be returned the actual line number in the document if they wanted to go to the physical document and find the line, as well as the assigned line number for where it would be if they were to search for it through the OCR file.

Figure 6 displays the new structure of our line-wise indexing output.

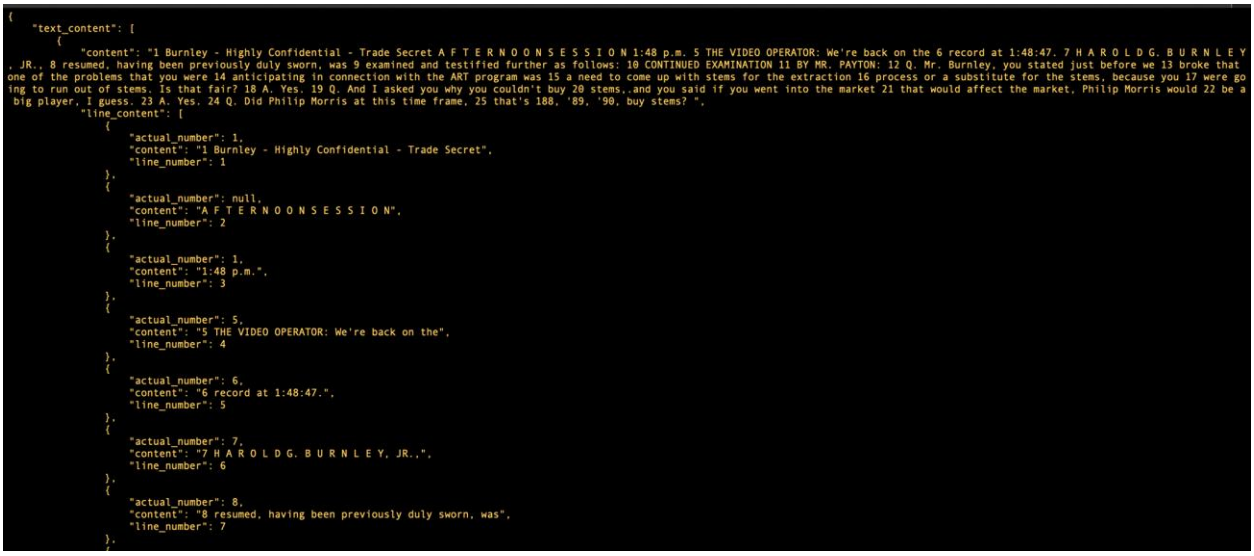

Figure 6 - New structure of line-wise indexing output

6. User Manual

6.1 UCSF Library

Our team used a specific search engine to index a collection of 14 million documents, found at https://www.industrydocuments.ucsf.edu/, that relate to the settlement between US states and the 7 large tobacco companies. Figure 7 shows the website of the UCSF library and the many forms of document selection. This link displays a source of documents found in UCSF's Industry Documents Library. In this library, the 14 million documents that our team is dealing with were produced by tobacco companies and include letters, archives, depositions, videos, and many other things.

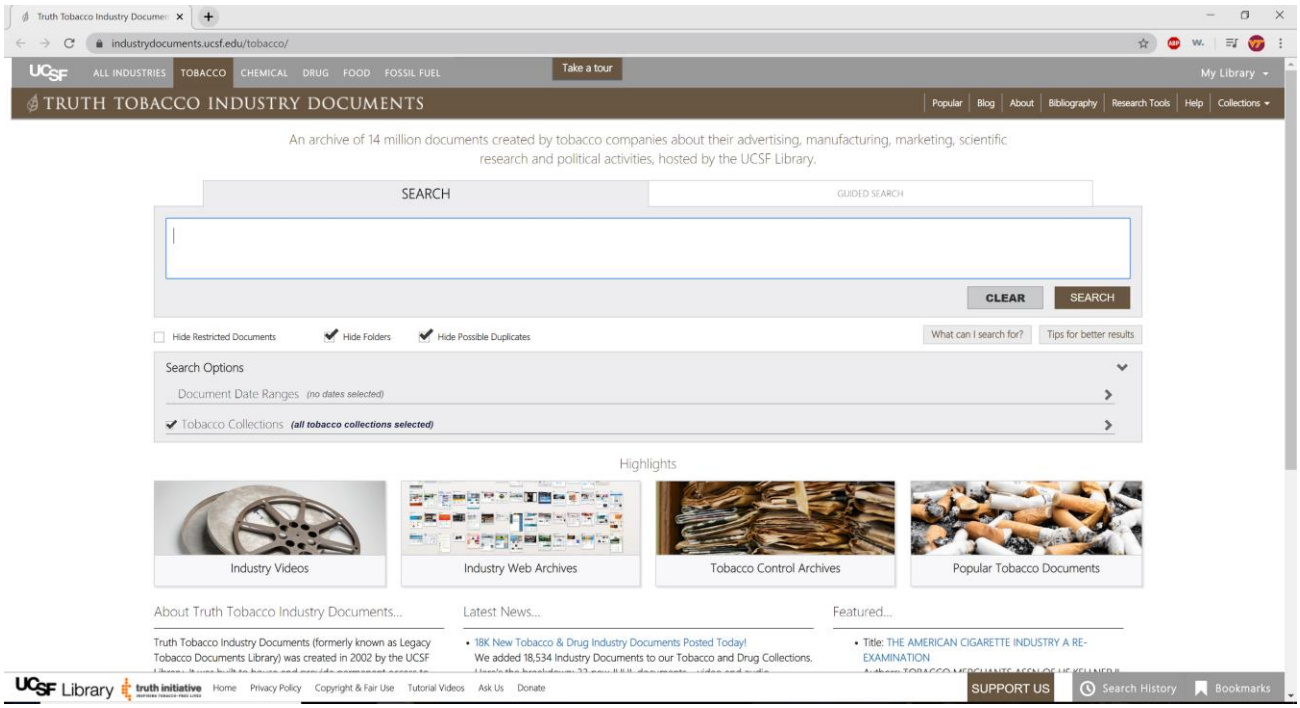

Figure 7 - UCSF Library Website

6.2 Users

In order to search the document dataset on Kibana, follow these steps

1. Navigate to the following address to go to our instance of Kibana.

http://2001.0468.0c80.6102.0001.7015.ccbe.4e22.ip6.name:5601/app/kibana#/home?_g=()

2. On the home screen, locate the **Manage and Administer the Elastic Stack** section and click on **Console**.

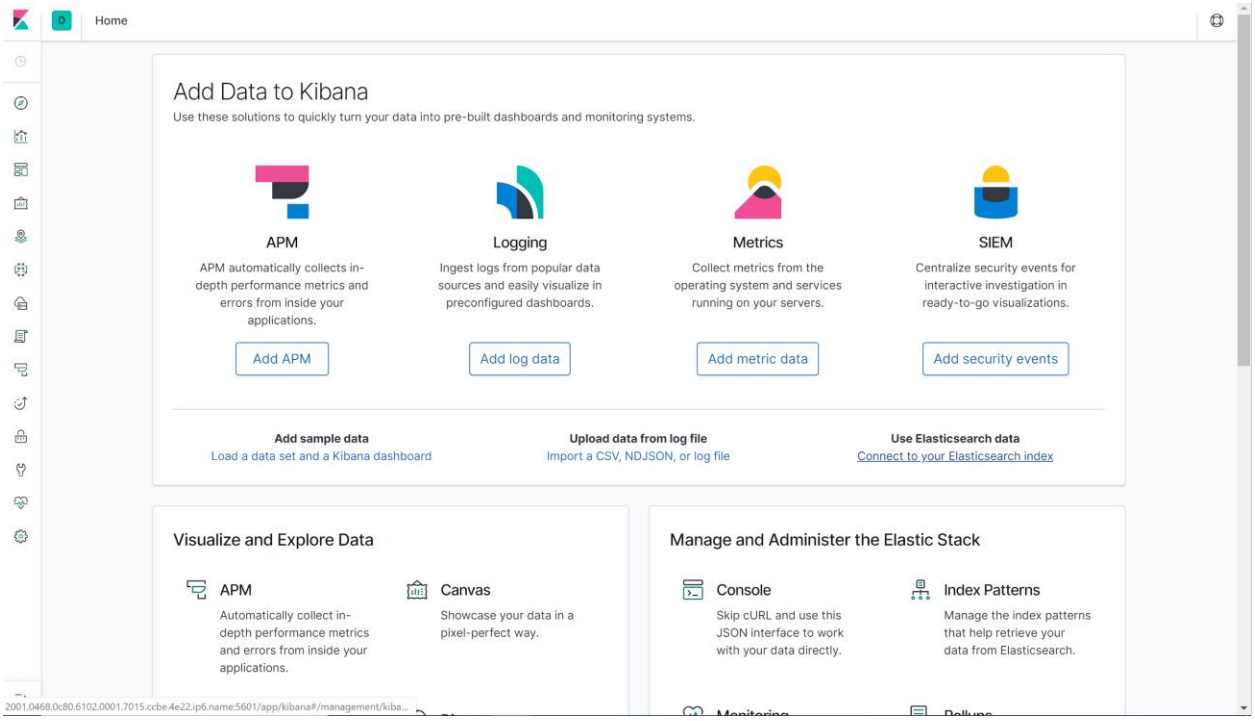

Figure 8 - Home-Screen for Kibana

3. On this screen, use some of the search queries provided below to search the database. The search query is typed in the box on the left and results appear in the box on the right.

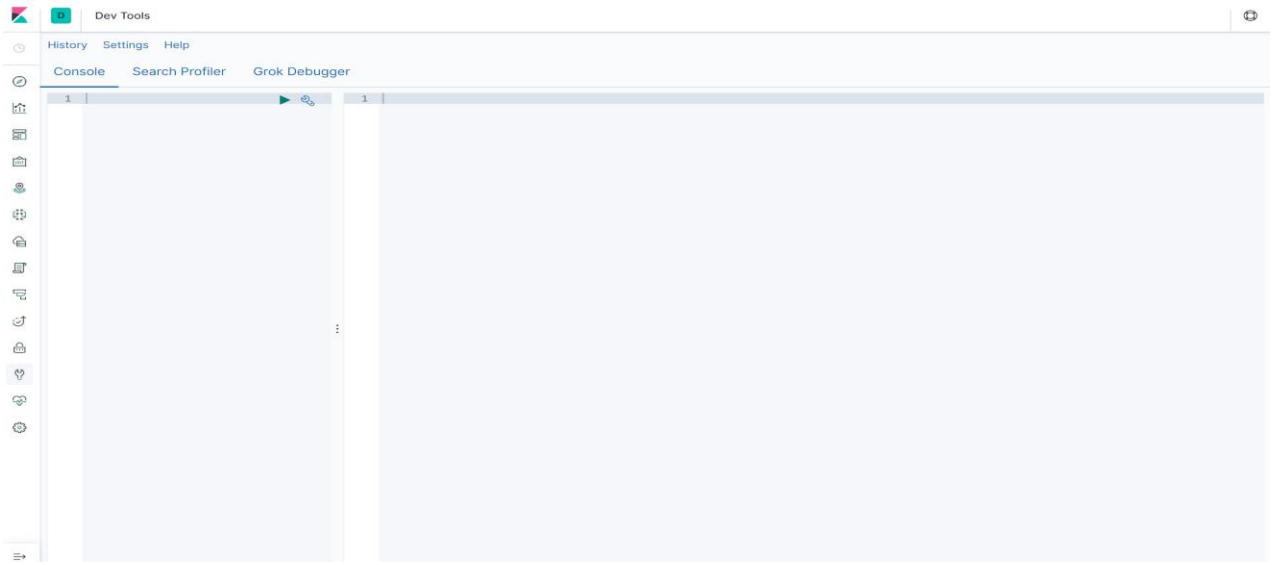

Figure 9 - Console for Queries

6.2.1 Tobacco Researchers

Users who are interested in the tobacco related aspects of this project, will have the following goals:

- Search documents by **keyword**
- Search documents by **time frame**

In order to search the database by **keyword**, the user should follow these steps:

1. Write a query and specify a path. An example of a query is provided below.

```
POST tobaccodep/_search
{
  "query": {
    "nested": {
      "path": "text_content.line_content",
      "query": {
        "match": {
          "text_content.line_content.content": "SANDRIDGE"
        }
      },
      "inner_hits": {
        "highlight": {
          "fields": {
            "text_content.line_content.line_number": {}
          }
        }
      }
    }
  }
}
```

2. Under match, specify search to be content.keyword, and type in a specific keyword to be searched. Sample code and output is shown in Figure 10.

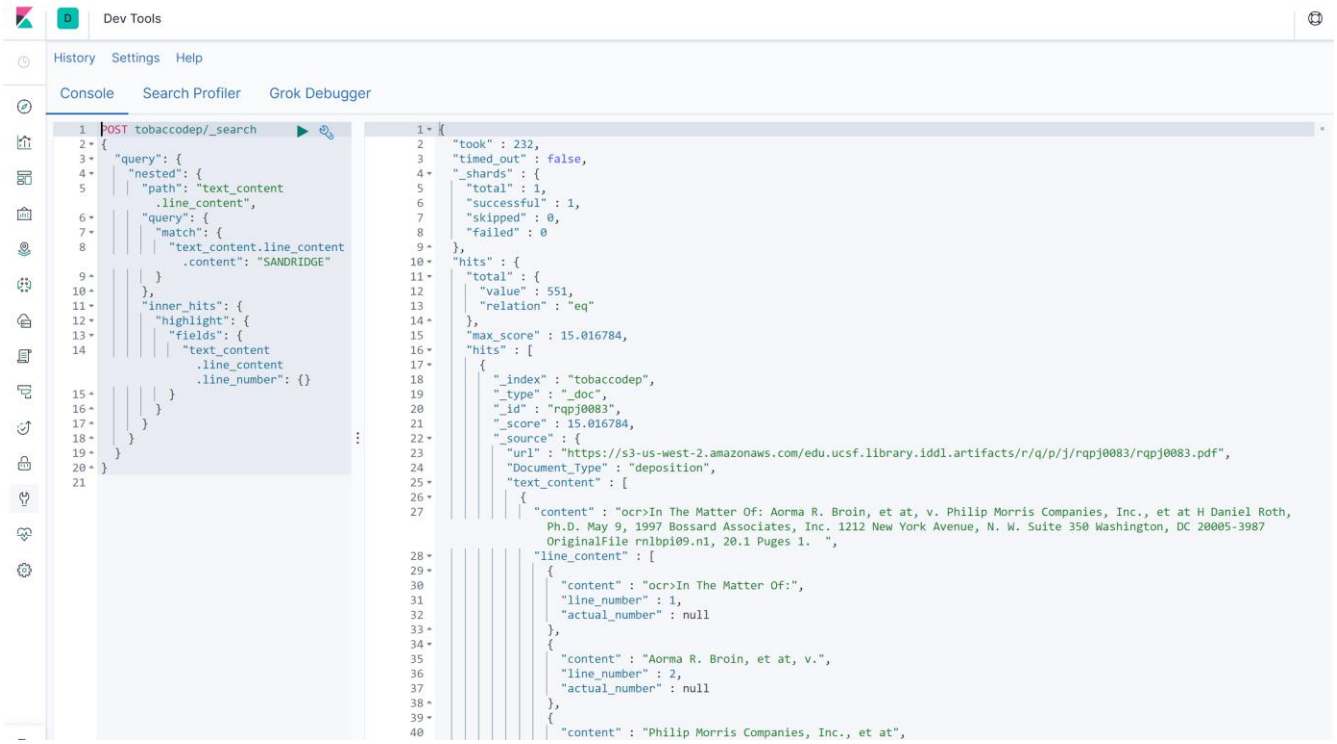

Figure 10 - Sample Query and Output for Keyword Search

In order to search the database by **time frame**, the user should follow these steps:

1. Use a search command similar to the following

```
POST tobaccodep/_search
{
    "aggs": {
     "range": {
        "date_range": {
            "field": "Document_Date",
            "format": "MM-yyyy",
            "ranges": [
                { "from": "02-2016",
                  "to" : "now/d"
                }
            ]
        }
     }
  }
}
```

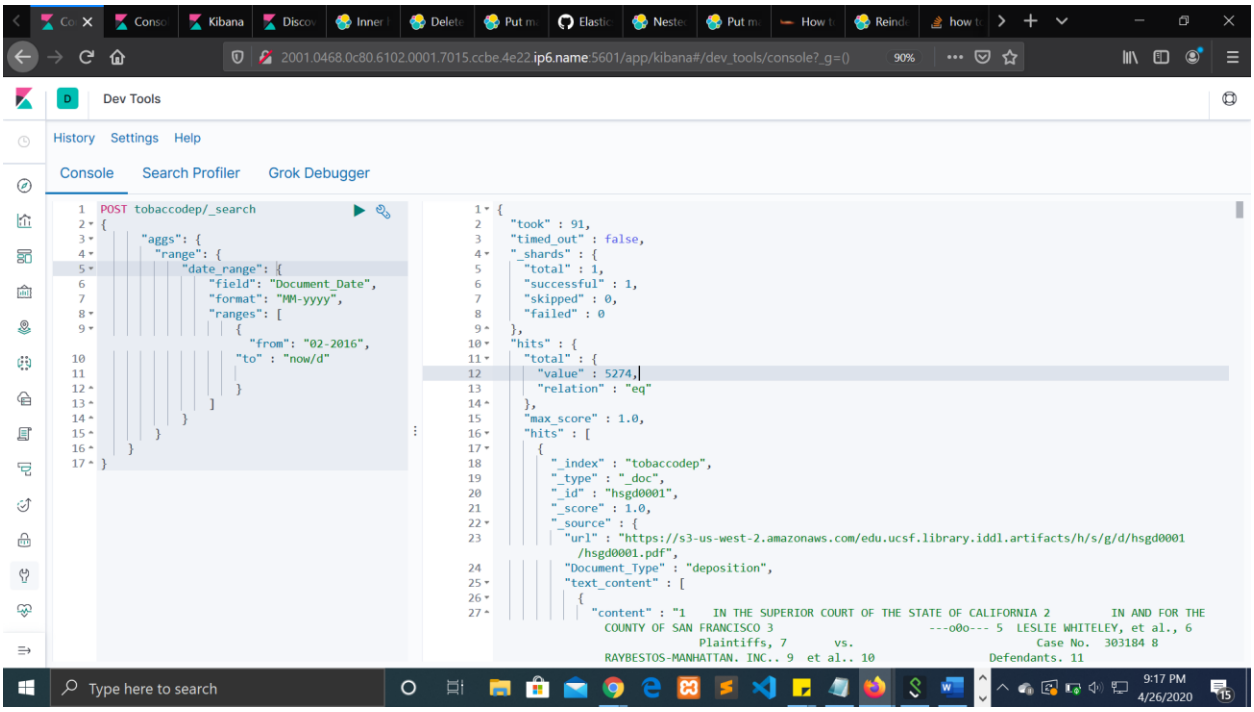

Figure 11 - Sample Query and Output for Time Frame Search

6.2.2 Computer Science Researchers

Users who are interested in the Computer Science related aspects of this project, such as ourselves, will have the following goals:

- Understanding the structure of the database and its contents.

In order to view this information, the user should follow these steps:

1. From the homepage, click on **Connect to your ElasticSearch Index** near the bottom right of the **Add Data to Kibana** section

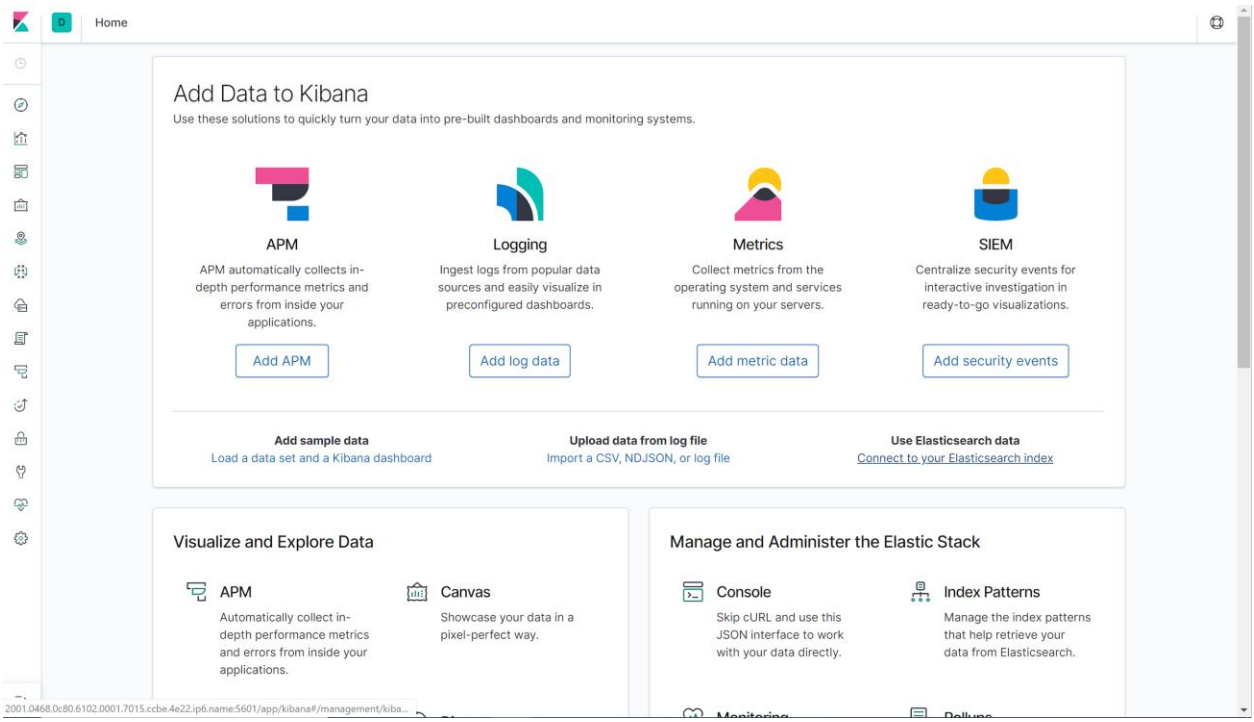

Figure 12 - Home-Screen for Kibana Data Addition

2. From the **Connect to your ElasticSearch Index** page, click on **Index Management** under **ElasticSearch.**

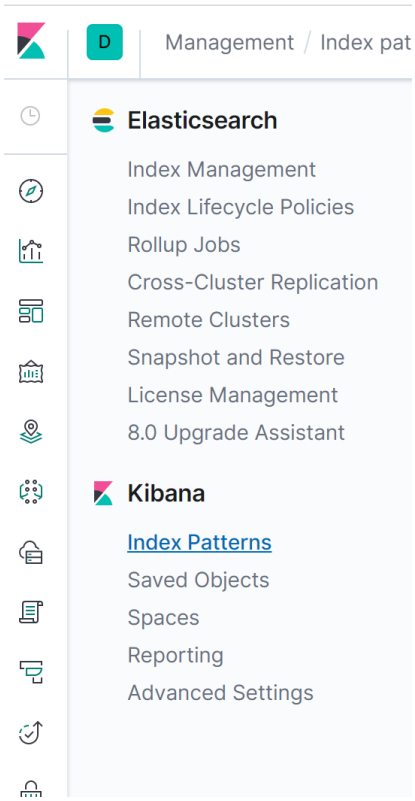

Figure 13 - Navigation to Index Management

3. On the **Index Management** page, select **tobaccodep** from the list of indices.

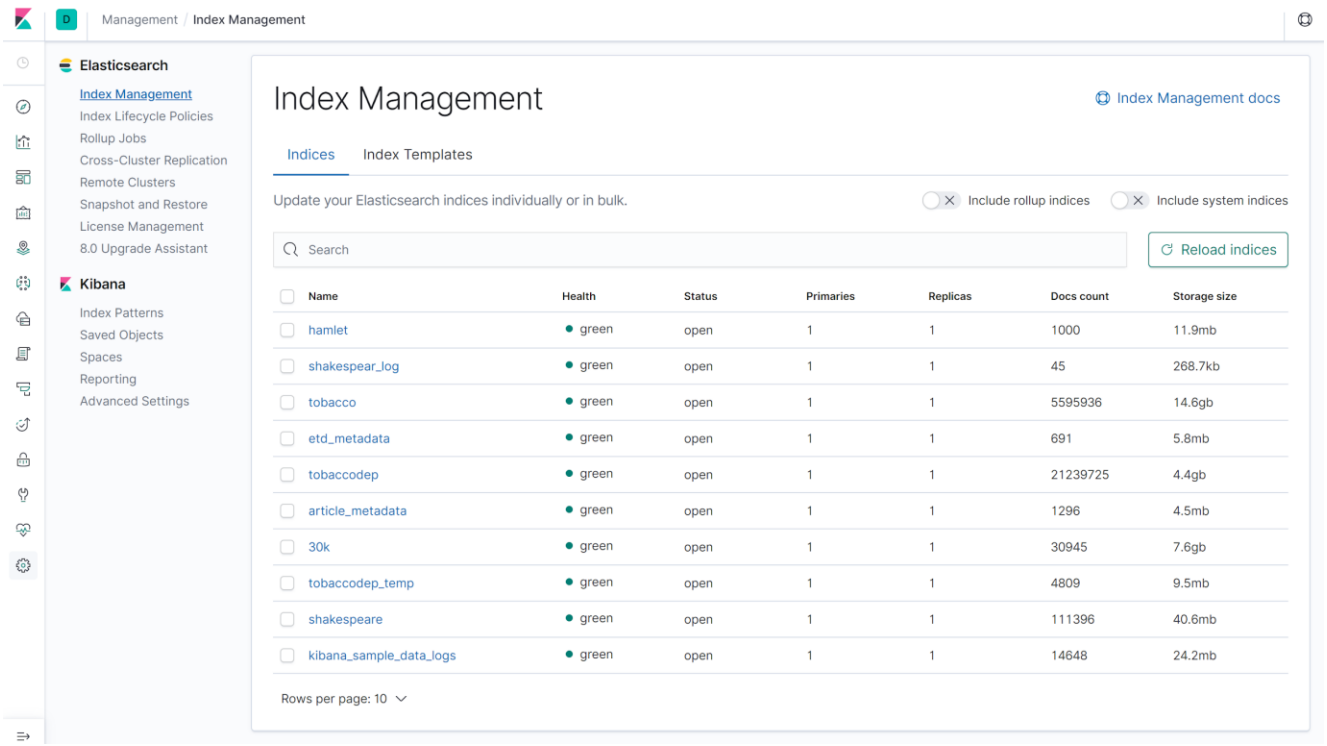

Figure 14 - Index Management Page

4. Navigate through the **Summary, Settings, Mapping,** and **Stats** tabs to view the properties and structure of the **tobaccodep** ElasticSearch index.

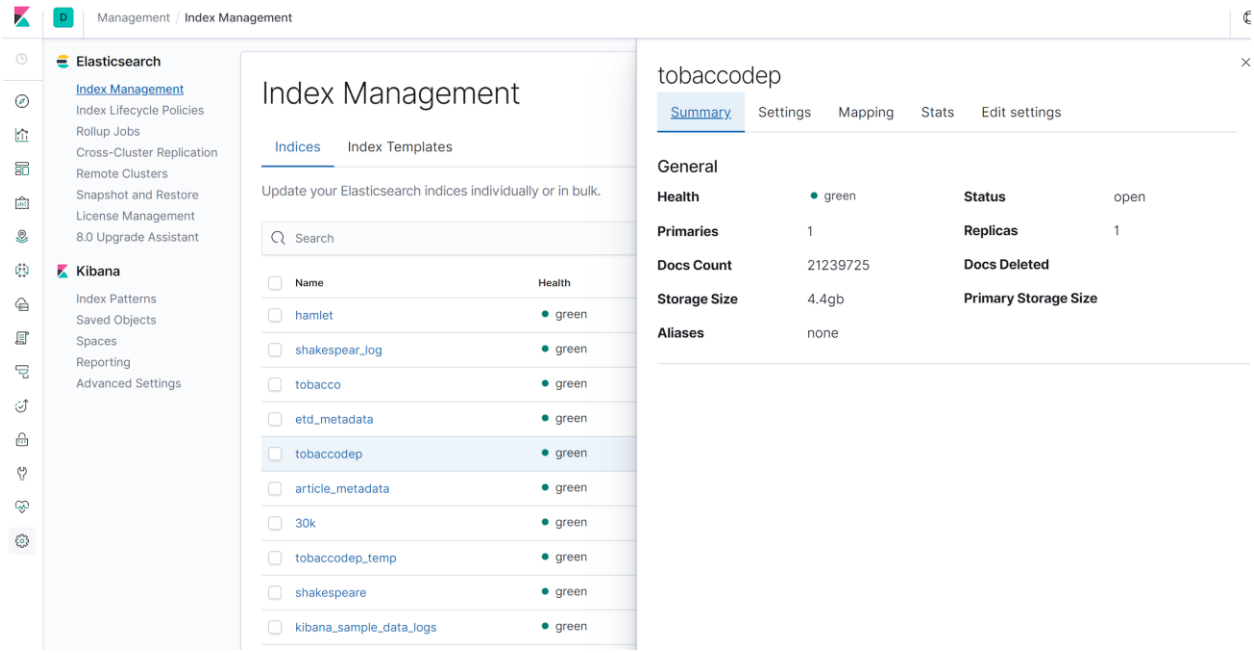

Figure 15 - Summary, Settings, Mapping, and Stats of tobaccodep

The tobacoodep index contains the 8000 deposition documents indexed line-wise. The document count says 21 million because of the nested data structure, Each nested field is regarded as a separate document. But the index actually only contains 8000 documents. The tobacco dep index can be searched line wise with the queries specified later in this report.

7. Developer's Manual

In this section, we show how to recreate and set up all aspects of the current system for ease of access. The aim is a step by step manual that will minimize the trials of future developers. Many references will be made to external sources such as the documentation provided by the software utilized such as Kibana, ElasticSearch, and MySQL database, as well as to documentation of work from the previous semester by teams in CS 5604, Information Storage and Retrieval. The work we have completed is a continuation of the work done by the previous teams and therefore the above mentioned documents illustrate the scope of work. The tobacco documents are obtained from the UCSF library and used to populate the MariaDB database. All referenced tables and keywords are obtained as such.

7.1 Accessing Remote Machine

The current system is mounted in a Virtual Machine (tsd.cs.vt.edu) that is hosted by the Department of Computer Science. The current development team has already installed MySQL and Python, but instructions on setup and deployment are posted in case of transfer of data to a new VM or any errors in the current system. To access and manipulate the Python scripts, or to access the database, the developer should login as user1. All related files are located in the directory 'fall2019'. This means the developer should execute the command below.

ssh user1@tsd.cs.vt.edu

To access and move the JSON format documents for ingestion in ElasticSearch, the user should login as root. Root access is needed to move data to **/mnt/ceph** and subdirectories which will be crucial for ingestion in ElasticSearch and any action that requires root privileges. This means the developer should execute the command below.

ssh root@tsd.cs.vt.edu

The password will be provided by either the client, or Professor Fox. If trying to access the VM, please contact the necessary individual to gain access.

To access ElasticSearch or Kibana or the above mentioned VM, the developer must either be connected to the Virginia Tech network or be using campus sponsored VPN software such as PulseSecure.

7.2 Accessing Dataset

The UCFS's IDL dataset containing a .sql file and a README file on how to set up the database is found from the link:

https://ucsf.app.box.com/v/IDL-DataSets

The files needed are named idldatabase.sql.tar.gz and README.txt.

7.3 Database Setup and Querying

7.3.1 Setting Up Database & Importing Dataset to Database

To install MariaDB on a Linux machine, start off with the command:

```
sudo yum install mariadb-server
```

Start and enable the server

```
sudo systemctl start mariadb
sudo systemctl enable mariadb
```

Check to make sure there were no errors and that mariadb is running smoothly

```
sudo systemctl status mariadb
```

Once the database is set up, you can access it with the command:

```
mysql -u root
```

If you want to use a password for security purposes, add a "-p" after this command every time to create and use it. This is strongly recommended.

Create the database and name it "data" and then exit out to the Linux shell.

```
CREATE DATABASE data CHARACTER SET utf8mb4 COLLATE
        Utf8mb4_unicode_ci;
Exit;
```

The database needs to be populated with the documents. First, untar the file

```
Idldatabase.sql.tar.gz
```

Run the following command:

```
tar -xvf idldatabase.sql.tar.gz
```

Then use this command to populate the database with all of the tobacco documents. This file contains around 15 million tobacco related metadata records for documents. It will take about 3-4 hours to finish. You will also need about 50gb of space.

```
mysql -u root data < idldatabase.sql
```

7.3.2 Explanation of Tables and Fields

```
[user1@tsd fall2019]$ mysql -u root
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 82
Server version: 5.5.64-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> SHOW databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| data               |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.05 sec)

MariaDB [(none)]> use data
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [data]> SHOW tables;
+----------------+
| Tables_in_data |
+----------------+
| idl_doc        |
| idl_doc_field  |
+----------------+
2 rows in set (0.00 sec)

MariaDB [data]> DESC idl_doc;
+-------------------+--------------+------+-----+---------+-------+
| Field             | Type         | Null | Key | Default | Extra |
+-------------------+--------------+------+-----+---------+-------+
| id                | int(11)      | NO   |     | NULL    |       |
| record_key        | varchar(8)   | NO   |     | NULL    |       |
| bates             | varchar(255) | YES  |     | NULL    |       |
| collection_id     | bigint(20)   | NO   |     | NULL    |       |
| dm                | int(11)      | NO   |     | NULL    |       |
| document_category | varchar(255) | NO   |     | NULL    |       |
| pages             | int(11)      | YES  |     | NULL    |       |
| industry_id       | bigint(20)   | NO   |     | NULL    |       |
+-------------------+--------------+------+-----+---------+-------+
8 rows in set (0.04 sec)

MariaDB [data]> DESC idl_doc_field;
+-------+----------+------+-----+---------+-------+
| Field | Type     | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| id    | int(11)  | NO   |     | NULL    |       |
| itag  | int(11)  | NO   |     | NULL    |       |
| value | longtext | NO   |     | NULL    |       |
+-------+----------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Figure 16 - MariaDB server entry and commands

Figure 16 shows how to get into the MariaDB server and commands to show the fields and tables of each database. The two tables of our database are shown to be idl_doc and idl_doc_field. Using the DESC command, all the fields of the tables are shown.

7.3.3 Queries

To select a record from idl_doc and idl_doc_field given the ID number, use the query:

```
SELECT * FROM idl_doc, idl_field WHERE id = 'id_value';
```

To find record keys of only deposition documents, use the query:

```
SELECT DISTINCT record_key from idl_doc, idl_doc_field WHERE value =
    'deposition';
```

Table 4 : Industry ID Key

ID	Name
1	Drug
2	Tobacco
3	Food
4	Chemical

According to Table 4, to receive documents of only a certain type of industry, use the query (2 is mainly used as almost all the documents are tobacco related):

```
SELECT * FROM idl_doc WHERE industry_id = 'industry_id_value';
```

7.4 Scripts

file2jsonDep.py - This script was used to do the line-wise indexing of the deposition documents. The script is located under user1/fall2019. This script is a copy of the file2json.py script except we added code to create a list of line dictionaries in the original data structure to keep track of the data in each line.

metadata_to_json_fast_line.py - This script was used to create the metadata+JSON for the line-wise indexed deposition documents. The script was adapted to fetch only the deposition documents from our database to index and format. This script would fetch the raw files from our database and run them through our line-wise code and generate a JSON file of the document completely indexed. This code would also add the index and ID to the first line of each document and add a new line at the end of the document so that it can be ingested into ElasticSearch properly. This script can be called from the command line like the following:

metadata_to_json_fast_line.py <start id> <end id> > <output_file>

ingestion_script.sh - This is a script that can be found under /mnt/ceph/AWSTobacco, and this script runs the curl command to push a JSON file to ElasticSearch on all of our JSON files. This script can loop through multiple JSON files and push them to ElasticSearch. It automates the process of running the curl command for each JSON document.

7.5 Processing Documents

One of the two key parts of this script is the query. This accesses the database and fetches all of the documents that the user wants to index, for the given range of IDs, and format for ElasticSearch.

```
# Get record keys
query = "SELECT id, record_key FROM idl_doc WHERE id>=%s AND id<%s AND industry_id = 2"
cursor.execute(query, (min_id, max_id))
results = cursor.fetchall()
id_to_record_key = {}
for id, record_key in results:
    id_to_record_key[id] = record_key

query = "SELECT id, itag, value FROM idl_doc_field WHERE id>=%s AND id<%s AND value='deposition' ORDER BY id"

cursor.execute(query, (min_id, max_id))

entries = cursor.fetchall()
assert len(entries) > 0
```

Figure 17 - Query to access database and fetch required documents

The second key part is this function in the script. This function calls the line-wise indexing script, and takes in the raw data file from the database and converts it to a JSON format while indexing the document line-wise. If the user wants to index the file differently they would use the function that loadtext() calls. If the user decides to not index the file at all they would remove the call to this function.

```
import logging

logfile="metadata_to_json_fast.log"

def loadtext(record_key):
    filename = file2jsonTest.getFileName(record_key)
    pagecontent = file2jsonTest.file2Json(filename)
    return pagecontent
```

Figure 18 - Function for line-wise indexing calls and JSON format conversion

7.6 ElasticSearch

7.6.1 Migrating Data for Ingestion

Before ingestion of the data can place, the user must first move the data into the **/mnt/ceph** directory or related subdirectories. To move any generated data to the listed directory, the developer must have root access which is discussed in Section 7.1. After gaining that, the user can move the documents by accessing the **tsd.cs.vt.edu** as the root user and using the command listed below.

mv 'source path' 'destination path'

7.6.2 Connecting to ElasticSearch

After the data has been moved to **/mnt/ceph** directory or subdirectories, the developer must login to cloud.cs.vt.edu and access the els-ceph container as shown in Figures 19 and 20.

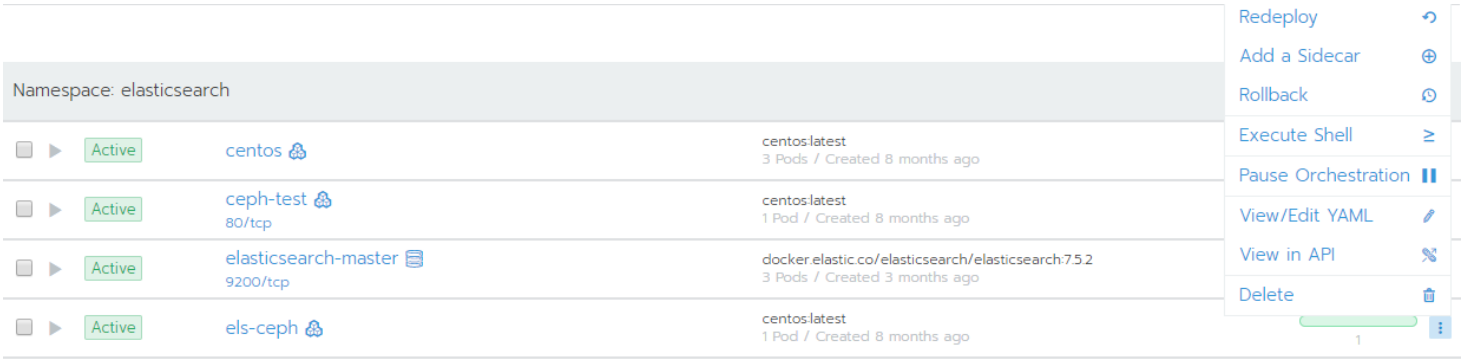

Figure 19 - els-ceph container

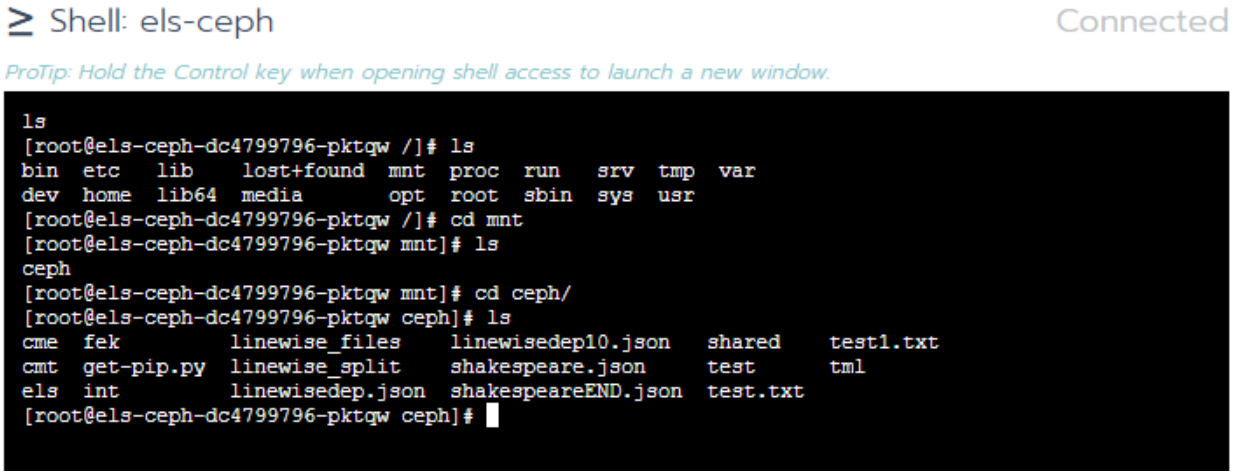

Figure 20 - els-ceph shell

After entering the els-ceph shell, the connection can be tested by sending an empty request by typing the internal server IP along with the terminal node. The internal server IP is currently 10.43.54.87:9200. Thus, the connection would be tested using a curl request as shown below.

curl 10.43.54.87:9200

If the curl request is made properly, then the server should return information validating the connection in the format shown in Figure 21

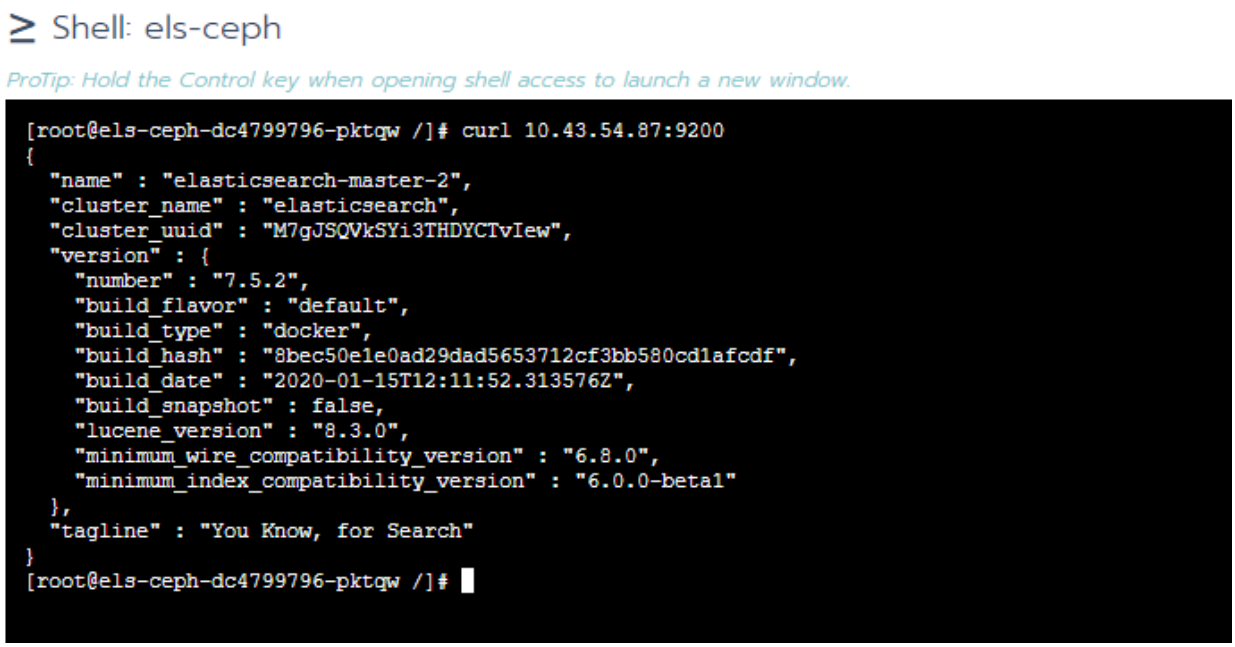

Figure 21 - Server Output upon proper curl request

7.6.3 Data formatting in ElasticSearch

For proper ingestion into ElasticSearch, the data must have the newline delimited structure as mentioned in the ElasticSearch Document API:

```
{ "index" : { "_index" : "test", "_id" : "1" } }
{ "field1" : "value1" }
```

Section 7.5 illustrates how to format the data to include the newline delimited structure through the use of the Python script metadata_2_json_fast_line.py. It is very important that the structured data also contains a newline character at the end of the data file. To accomplish this

the developer can go to the end of the data file and press the enter key to add the newline character at the end of the file.

It is important to create an index in Kibana and an associated index mapping that will be covered in-depth in the Kibana section. Before ingestion can be done, it is important to see if the index has been created. That can be done with the command below.

```
curl 10.43.54.87:9200/_cat/indices?v
```

If the index is listed, that means the developer has successfully created an index for the data with the corresponding index name. The listed index should appear in the index category after execution of the above command as shown in Figure 22

Figure 22 - Listed Index

7.6.4 Ingestion into ElasticSearch

Now that the data is in the proper newline delimited structure, the developer should create an index with the associated name, and create an index mapping that adheres to the structure of the data itself. The proper rules and guidelines are listed in Section 7.7 and should be utilized before using the curl command for ingestion. Several errors can occur if the index is not created, or if the index mapping is not committed. If the index has been created but the developer has not created an index mapping or is utilizing a false mapping, using a curl command to the API endpoint will result in no response from the server. The developer should then view if the index has ingested any data by utilizing the command below. If no documents have been ingested, examine the index mapping.

```
curl 10.43.54.87:9200/_cat/indices?v
```

If the data being ingested into ElasticSearch is too large, an error will be given in the terminal. Section 7.5 covers proper splitting of data to smaller JSON files and how the curl command can be automated to run for the smaller JSON files. Generally, files being ingested should be smaller than 250 mb. The proper splitting of data can minimize errors in ingesting documents and was a major obstacle in the current development phase. The proper command for ingestion is listed below. The developer simply needs to change the name of the file from the current linewisedep.json to their own data file.

```
curl -s -H "Content-Type: application/x-ndJSON" -XPOST
10.43.54.87:9200/_bulk              --data-binary "@linewisedep10.json"
```

7.7 Kibana

7.7.1 Setting up an index

All documents on ElasticSearch are stored by index. In order to connect Kibana to the files on ElasticSearch you need to set up a corresponding index on Kibana. To create an index run the following command:

```
PUT /<index_name>
```

If you want to preserve the settings and mappings of a pre-existing index, it might be easier to run the reindex command with a pre-existing index with just a limited number of documents. This can be with the following command:

```
POST _reindex
{
“max_docs”: 5000
  "source": {
    "Index": “ <original_index>”
  },
  "dest": {
    "index":  “<new_index>”
  }
}
```

The max_docs field can be used to restrict how many documents to reindex. This will create a new index and reindex 5000 documents to the destination index from the source index.

Then, if you do not need the documents in this index, run the delete command to delete the documents from the new index. Doing this will create a new empty index with all of the mappings and settings copied from the source index.

The delete command is:

```
POST /<index_name>/_delete_by_query?conflicts=proceed
{
  "query": {
    "match_all": {}
  }
}
```

All of the above was run in that order to create the tobaccodep index which would hold all of the 8000 deposition documents that were indexed line-wise.

7.7.2 Mappings

To edit the mapping run a command similar to the following.

```
PUT /<index_name>/_mapping
{
  "properties": {
    "email": {
      "type": "keyword"
    }
  }
}
```

The mapping needs to match the structure of your JSON files, to ensure that all fields are accessible when searching through the documents in the index.

7.7.3 Nested fields

When there are fields that are arrays of objects that need to be indexed, nested datatypes are useful. They allow the array objects to be grouped together so when they are indexed they are indexed as a single object. You can add a nested datatype in the mapping like the following:

```
PUT tobaccodep_temp/_mapping
{
  "properties": {
    "text_content": {
    "type": "nested"
    , "properties": {
        "content": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "line_content": {
          "type": "nested",
          "properties": {
            "actual_number": {
              "type": "long"
            },
            "content": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            },
            "line_number": {
              "type": "long"
            }
          }
        },
        "page": {
          "type": "long"
        }
      }
    }
  }
}
```

Figure 23 - Addition of nested datatype in mapping

This is what was used to update the mappings for the line-wise indexed files. This mapping has multiple nested datatypes. Text_content is of type nested and the line_content inside of text_content is also of type nested.

7.8 Testing and Evaluation

Each of the files created are fairly large so it makes sense to first test a sample of the data for pushing into ElasticSearch. Knowing that two lines represent one document for each of the JSON files created to be pushed into ElasticSearch, it is possible to create sample files with simple Linux commands. Use this command to cut the top part of any file:

```
head -'# of lines' "file_name.ext" > "test_sample.ext"
```

Here # of lines is how much of the beginning lines we want, file_name.ext is the large file, and test_sample.ext is the file holding only the number of lines specified.

This allows a much smaller sample to be pushed onto ElasticSearch and will make it faster to recognize any errors early.

Overall we created the tobaccodep index and ingested about 8000 files to it, all indexed line-wise. We also created another tobacco index and pushed about another 3 million tobacco files to this index. We are continuing to add to this index today and are aiming to have about 8 million documents pushed to that index. This will take a little bit of time as Kibana keeps crashing and it takes a couple hours to ingest each million. There are also times that the curl command will not finish executing so this is also adding to the wait times, but it is in progress now. We have finished the indexing of all 8 million tobacco files and are now in the process or ingesting them into ElasticSearch.

8. Future Work

Although much has been done in regards to line-wise indexing, page-wise indexing, Kibana, and ElasticSearch, numerous other improvements are possible, especially those involving Machine Learning.

There is a lot of great processing that can be done with Machine Learning techniques that might simplify the contents of longer documents for better ease of understanding. There is some work that has occurred already with these files to set the stage for work like this. The main Machine Learning concepts that we think should be considered are Sentiment Analysis, Text Summarization, and Immediate Data Extraction. Sentiment Analysis could help with the studying of the different reactions to tobacco related court cases, or other efforts of tobacco companies. Text Summarization would help people like Dr. Townsend, who read hundreds to thousands of these documents over many hours. With Immediate Data Extraction, we would be able to read virtually any document and extract text and data without manual effort. Custom code would not be required either. One way to achieve this is Amazon Textract, which accurately extracts data from documents, forms, and tables. This would be beneficial, since there are thousands of documents. Having the data extracted would greatly speed up the process of reading through the documents.

In addition, using improved OCR methods could be considered. There is a wide variety of documents that the developers are working with. These documents are created, prepared, and scanned in a variety of different ways, which presents numerous challenges for extracting information from the PDFs themselves. As with all OCR technologies, there is a trade-off between how exact the results are, and how long the process takes -- an important factor when dealing with millions of documents. Finding a way to improve these methods would reduce the noise content that sometimes needs to be filtered out of the metadata, ensuring that details are not left out when trying to provide these documents to researchers.

Additionally, there should be some way of evaluating the documents. There are a plethora of documents, each having a different level of significance. Having a usefulness rating would help users focus on important, relevant information. Articles that are viewed more might have more valuable information as well. Having a rating and view count might help Dr. Townsend, in allowing him to read the most important information first.

9. Acknowledgments

The AWS Tobacco Settlement Retrieval team wants to acknowledge the key individuals that have been crucial to the success of the project. First, and foremost, we want to thank our client Mr. Satvik Chekuri for his continued support as both a client and mentor in this project. We want to thank Dr. Edward A. Fox for his continued aid and insight in the project. We want to thank Dr. Townsend for agreeing to serve as the major client and for the project proposal and information regarding the impacts on the industry. We want to thank the Project Teams from Fall 2019 CS5604: Information Retrieval for the amazing work they have done in setting up all resources and thorough documentation. Furthermore, we want to thank our fellow classmates and peers as well as the teaching assistants Carlos Augusto Bautista Isaza and Shuai Liu, for critical feedback throughout the semester and for their continued support.

Bibliography

[1] Bendelac, A, et al. *Information Storage and Retrieval Collection Management of Tobacco Settlement Documents*. Technical report, Virginia Tech, 2019. http://hdl.handle.net/10919/96437. [Accessed: 25-Apr-2020].

[2] Li, Y, et al. *Final Report CS 5604: Information Storage and Retrieval*. Technical report, /Virginia Tech, 2019. http://hdl.handle.net/10919/96310. [Accessed: 25-Apr-2020].

[3] Powell, E, et al. *Fall 2019 CS5604 Information Retrieval Final Report Front-End and Kibana (FEK)*. Technical report, Virginia Tech, 2019 . http://hdl.handle.net/10919/96418. [Accessed: 25-Apr-2020].