Multimedia, Hypertext, and Information Access (CS 4624)

Common Crawl Mining Project

Final Report

Virginia Tech, Blacksburg, VA 24061

26 April 2017

Team:	Brian Clarke Thomas Dean Ali Pasha Casey J. Butenhoff	
Project Manager:	Don Sanderson	Eastman Chemical Company
Client:	Ken Denmark	Eastman Chemical Company
Instructor:	Edward A. Fox	Virginia Polytechnic Institute and State University

Table of Contents

Table of Figures

Table of Tables

1 Executive Summary

This report describes the Common Crawl Mining project for the Multimedia, Hypertext, and Information Access (CS 4624) class at Virginia Tech during the Spring 2017 semester. The main goal behind the Common Crawl Mining system was to improve Eastman Chemical Company's ability to use timely knowledge of public concerns to inform key business decisions. It provides information to Eastman Chemical Company that is valuable for consumer chemical product marketing and strategy development.

To meet this goal, Eastman desired a system that provides insight into the current chemical landscape. Information about trends and sentiment towards chemicals over time is beneficial to their marketing and strategy departments. They wanted to be able to drill down to a particular time period and look at what people were writing about certain keywords. The system also needed to act as a black box, i.e., function without the user needing to understand the inner-workings of the system.

This project provides such information through a search interface. The interface accepts chemical names and search term keywords as input and responds with a list of Web records that match those keywords. Included within each record returned is the probable publication date of the page, a score relating the page to the given keywords, and the body text extracted from the page. Since it was one of the stretch goals of the project, the current iteration of the Common Crawl Mining system does not provide sentiment analysis. However, it would be relatively straightforward to extend the system to perform it, given the appropriate training data.

In addition to Amazon Web Services' (AWS) tools, we used several other frameworks to complete our project. Python is the main language used for development. We used various Python packages such as mrjob for MapReduce functionality, Webhose for publication date extraction, and Beautiful Soup for HTML parsing.

The general constraints for this project included the time necessary to complete the project for the course as well as funds to run our system on AWS. Resource constraints included the number of articles maintained in Common Crawl, network bandwidth, memory, storage, and processor speed. Financial resources were a significant constraint as they prevented us from ever processing a complete monthly archive and threaten the production viability of this project.

The final Common Crawl Mining system is a search engine implemented using Elasticsearch. Relevant records are identified by first analyzing Common Crawl for Web Archive (WARC) files that have a high frequency of records from interesting domains. Records with publication dates are then ingested into the search engine. Once the records have been indexed by Elasticsearch, users are able to execute searches which return a list of relevant records. Each record contains the URl, text, and publication date of the associated webpage.

2 Introduction

This document represents the total sum of effort on the Common Crawl Mining project. It outlines our process to complete this project, the problems we encountered, and the solutions to solve these problems. We discuss the lessons we learned and recommendations for the project going forward. We include a timeline of our efforts over 3 months of work. The report includes documentation of the entire development process of this system from requirement definition through implementation and testing. It also includes all the documentation that should be necessary for users and developers to continue to work with the system. We include technical functional requirements as well as diagrams of our system. Developers and users should both feel comfortable with the system after reading this report. For that reason, this report is important. To support future expansion, we outline the files we created to develop the project.

Little content related to this project was known by our team before we began. It was necessary to have multiple conversations with the client to ensure we understood the requirements and goals of this project. Because early communication with the client was limited, we researched topics we thought might be included in the project. Some of these, natural language processing and sentiment analysis, were not used because we determined they were outside the scope of our assignment.

The specific objective of this project is to provide Eastman with a system to query the Common Crawl dataset for search terms and related chemicals. Eastman plans to take away several aspects of the results such as being able to compute weighted relevance of articles to search terms and chemicals and to also consider the date the article was published. Using Kibana, a product of Elastic, users can visualize results and even watch them change while new data is being indexed. For example, a user could create a graph that shows the number of articles returned by a given query that were published in each month of a given year. Additionally, users can also perform weighted queries through Kibana in the form of a GET search. Kibana can assign each term a different weight, based on how important they are relative to each other, and return a list of hits in decreasing order of weighted score.

We used several of Amazon’s web services to support our system. Elasticsearch and Elastic MapReduce provided capabilities to store and analyze our data. The data from the Common Crawl archives were imported into AWS (Amazon Web Services) in the form of WARC (Web Archive) files. The costs of running a portion of the Common Crawl database were much higher than the client anticipated for this project and may be unrealistic for a small project. We analyzed bottlenecks that increase the cost beyond expectations and explain those in this report.

3 Users' Manual

This section considers the deployment, and Elasticsearch searching. The deployment steps include creating accounts and configuring key authentication and launch instances. Elasticsearch searching includes multiple types of queries.

3.1 Deployment

3.1.1 Create Accounts

Navigate to https://aws.amazon.com

1. Create a new account which will act as the root account
2. Make note of account ID

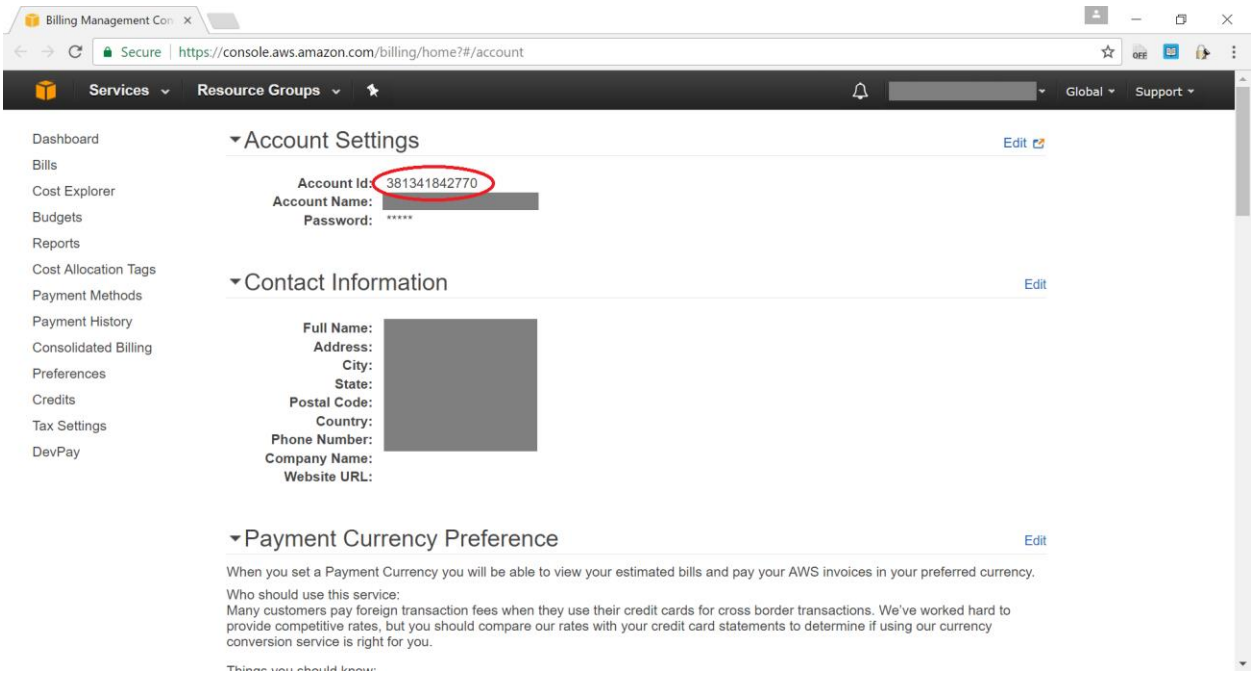


Figure 1. Account ID

Navigate to https://console.aws.amazon.com/iam

3. Create an IAM account called "console"
    a. Grant AWS Management Console access
    b. Create a user group called "console"
        i. Grant "AmazonEC2FullAccess"

ii. Grant “AmazonESFullAccess”

c. Make note of AWS Management Console Address when complete

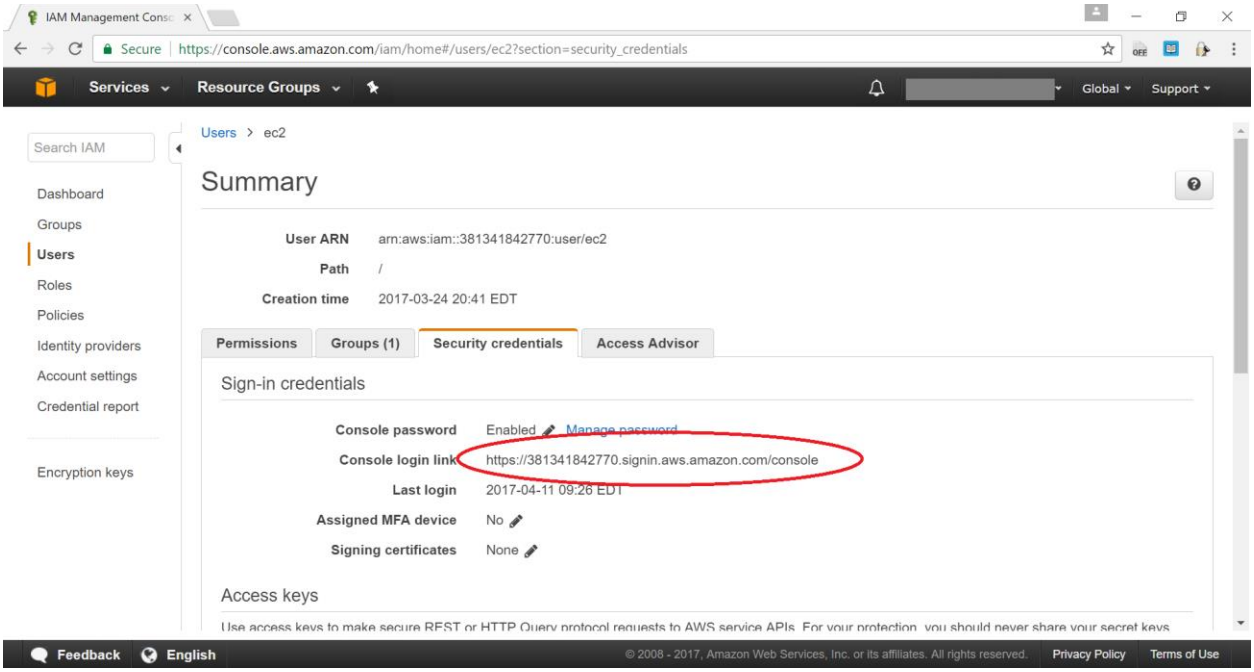


Figure 2. Console Login Link

4. Create an IAM account called “es”
    a. Grant “Programmatic access”
    b. Create a user group called “ES”
        i. Grant “AmazonESFullAccess”
    c. Make note of Access key ID and Secret access key
5. Create an IAM account called “emr”
    a. Grant “Programmatic access”
    b. Create a user group called “EMR”
        i. Grant “AmazonElasticMapReduceFullAccess”
        ii. Grant “IAMReadOnlyAccess”
        iii. Grant “AmazonEC2FullAccess”
    c. Make note of Access key ID and Secret access key

3.1.2 Configure Key Authentication and Launch Instances

1. Log out of root account and navigate to noted AWS Management Console Address
    a. Log into ec2 account
    b. Select region
    c. Import a key pair

i. Run these commands on your local computer terminal
    1. ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa -q -N "" -C "your_email@example.com"
    2. cat .ssh/id_rsa.pub
        a. Make note of public key contents
ii. Paste the public key contents noted before

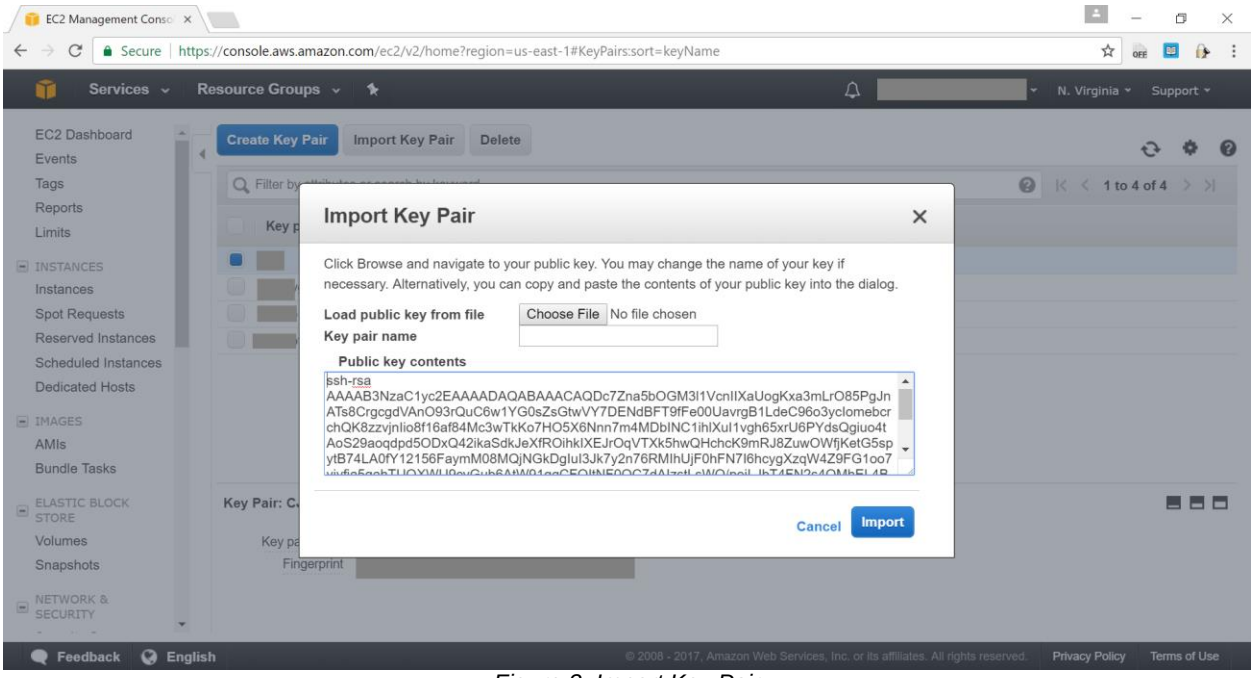
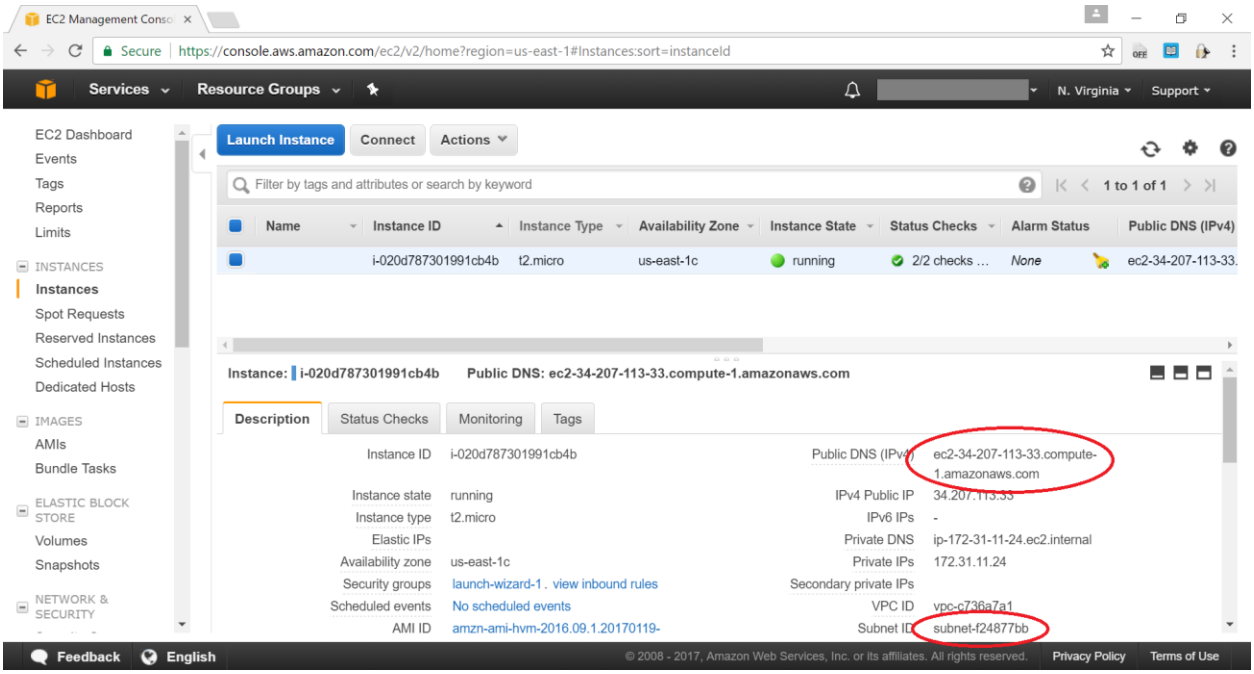


Figure 3. Import Key Pair

d. Create a key pair named “mrjob”
    i. Make note of the contents of the private key file it downloads
e. Launch an EC2 instance
    i. Amazon Linux AMI
    ii. T2.micro
    iii. Choose the previously created key pair
    iv. Edit the security group
        1. Add a custom TCP rule
            a. Port range 5000
            b. Source anywhere
    v. Wait until it initializes and note its Public DNS address and Subnet ID

Figure 4. Public DNS address and Subnet ID



f. Create an ElasticSearch domain called “es-cluster”
   i. Instance count 4
   ii. Instance type m3.medium.elasticsearch
   iii. Storage Type EBS
   iv. EBS volume type Magnetic
   v. EBS volume size 20
   vi. Set the domain access policy
      1. “Allow or deny access to one or more AWS accounts or IAM users”
      2. Enter the IAM user ARN of the “es” user
   vii. Make note of the endpoint address

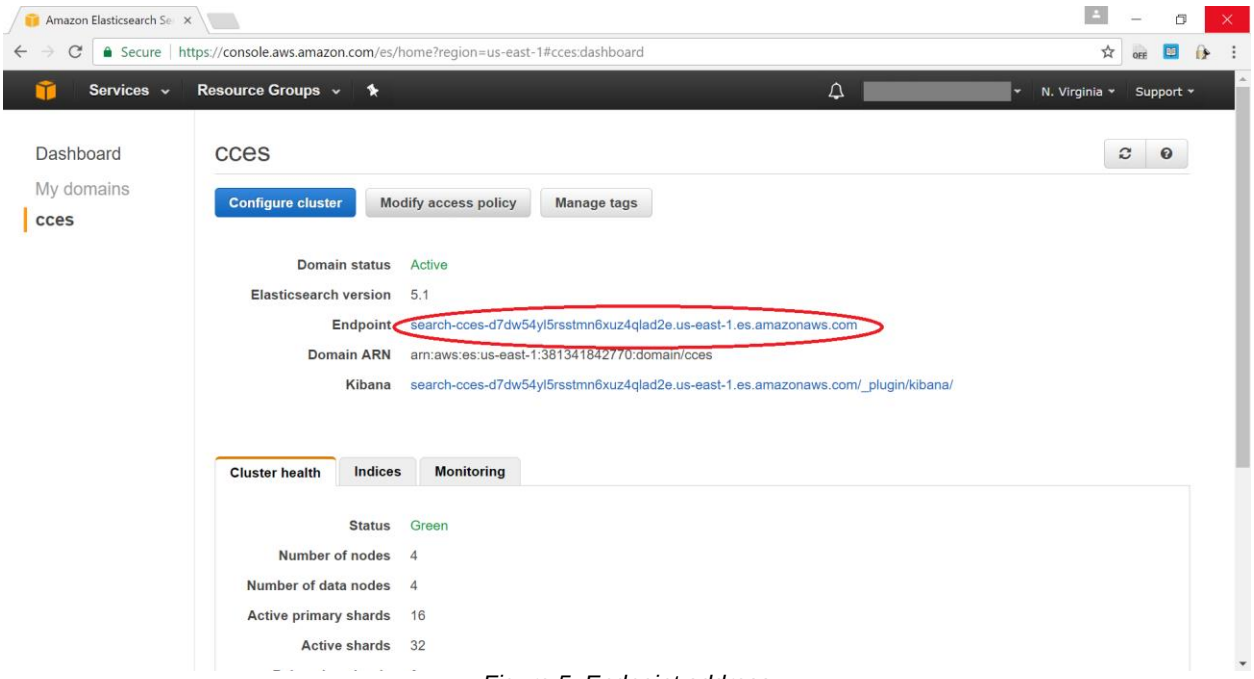


Figure 5. Endpoint address

2. Connect to noted Public DNS address with an SSH client
    a. Log in as ec2-user with private key authentication and run the following:
        i. ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa -q -N "" -C "your_email@example.com"
        ii. cat .ssh/id_rsa.pub
            1. Make note of public key contents
        iii. Navigate to https://github.com/
            1. Add a new SSH key
                a. Paste previously noted public key
        iv. sudo yum update -y
        v. sudo yum install -y docker git
        vi. sudo service docker start
        vii. sudo usermod -a -G docker ec2-user
        viii. git clone git@github.com:cjbvt/CommonCrawlMining.git
        ix. cd CommonCrawlMining/
        x. git checkout feature/elasticsearch
        xi. Replace the contents of mrjob.pem with the private key file noted before
        xii. Replace Access key ID and Secret access key in es_indexer.py with those noted before for the “es” user
        xiii. Replace the es_host string in es_indexer.py with the Elasticsearch endpoint address noted previously

xiv. Replace the input file “test-1.warc” in app.py with the name of the file containing an S3 paths for a WARC file on each line, such as “CC-MAIN-2017-09-warc.paths”
xv. Replace the upload file “test-1.warc” in mrjob.conf with the same file as in xiv.
xvi. Replace Access key ID and Secret access key in mrjob.conf with those noted before for the “emr” user
xvii. Replace Subnet ID with the value noted for the EC2 instance
xviii. Replace the path in test-1.warc with the path to the warc file to index

b. Log out and log back into SSH client and run the commands:
   i. cd CommonCrawlMining/
   ii. ./run.sh
c. Wait for the docker container to build and start

3. Connect a web browser to port 5000 of the EC2 instance’s noted Public DNS address
   a. e.g. “http://public-dns-address:5000”
4. Connect to the /start endpoint to begin an indexing operation
   a. e.g. “http://public-dns-address:5000/start”
5. On client computer or a server inside a firewall
   a. Install Node.js and run the following:
      i. npm install -g aws-es-kibana
   b. Either edit proxy.sh to contain your access keys and run it, or run the following:
      i. export AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXXXXX
      ii. export AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXX
      iii. aws-es-kibana <es-cluster-endpoint>
   c. Connect to http://127.0.0.1:9200/_plugin/kibana

3.2 Elasticsearch Searching

When we were defining the project requirements, the system was not supposed to have a front-end and was only only available as a service of a computer system. The interaction would depend on the interface or library that other system uses to interact with Elasticsearch. As such, it is not possible to cover all aspects of future use here, in any meaningful way. But because the system was implemented using Elasticsearch, a nice search front-end called Kibana is also included in case users want to interact with our system more directly.

Using Kibana, you can search Common Crawl for specific search terms or chemicals. At this point, there is only one search bar instead of one for each search term and a chemical. However, you can use various query capabilities that allow this search bar to become powerful. For example, you can search for a single string, the most basic of queries. You can also perform a search where you choose a prefix or suffix of a base word such as “infert*”. This performs a

search for all strings that begin with the prefix “infert”. Words such as infertility or infertile would be marked as a hit. Examples of queries can be found on the Elastic website.[36] The syntax for queries can be found on the Apache website.[37]

Another example of a search is a weighted query search. The development section of Kibana allows you to perform a GET search such as in Figure 6.

```
GET /_search
{
    "query": {
        "function_score": {
          "query": { "match_all": {} },
          "boost": "5",
          "functions": [
              {
                  "filter": { "match": { "html": "sputnik" } },
                  "weight": 23
              },
              {
                  "filter": { "match": { "html": "nitwit" } },
                  "weight": 42
              }
          ],
          "max_boost": 42,
          "score_mode": "max",
          "boost_mode": "multiply",
          "min_score" : 42
        }
    }
}
```

Figure 6. Weighted Query

This weighted query can be used to search for results, that are returned in decreasing order of relevance according to the search terms and the weight given to each term. The max_boost, min_score, boost, and weight fields can be modified for your preference. The “html” refers to the the field of the record from which the search term comes. For example, “html”: “sputnik” means we are looking in the HTML response of plain text for the search term “sputnik”. We could have also chosen “url” to search for specific URLs. The match query is of type boolean. It means that the text provided is analyzed and the analysis process constructs a Boolean query from the provided text.

boost_mode can be one of the following:

Table 1. boost_mode weighted query

boost_mode key term	description of mode
multiply	query score and function score is multiplied (default)
replace	only function score is used, the query score is ignored
sum	query score and function score are added
avg	average
max	max of query score and function score
min	min of query score and function score

The weight score allows you to multiply the score by the provided weight. This can sometimes be desired since the boost value set on specific queries gets normalized, while for this score function it does not. The number value is of type float. score_mode can be one of the following:

Table 2. score_mode weighted query

score_mode key term	description of mode
multiply	scores are multiplied (default)
first	the first function that has a matching filter is applied
sum	scores are summed
avg	scores are averaged
max	maximum score is used
min	minimum score is used

4 Developer's Manual

This section contains the data flow diagram for the system, the file inventory explaining all files and scripts pertinent to the system and the development software.

4.1 Flow Diagram

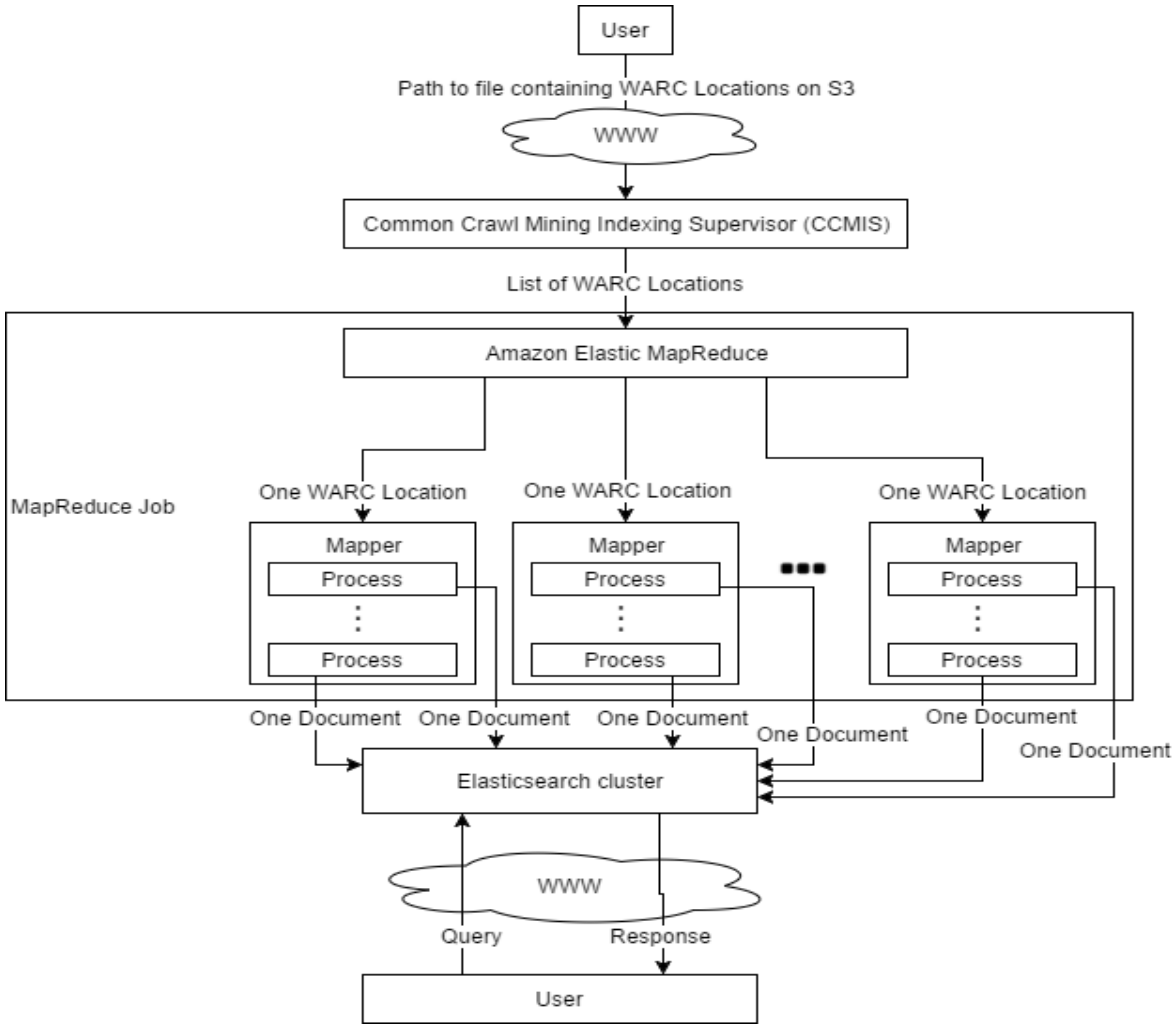


Figure 7: Data flow diagram with multiprocess mappers

Figure 7 is similar to the one in Appendix B, with the addition of record multiprocessing within the mapper. Because indexing is relatively inexpensive compared to the other operations that these processes are performing, they don't need to batch the indexing requests as we had originally planned. Each process is given one record at a time by the mapper, which it processes and indexes if all the necessary information can be extracted from it.

4.2 File Inventory

4.2.1 ChemicalListing.csv

`ChemicalListing.csv` contains a list of commercial chemicals that were paired with search terms

- Each row contains a chemical's name and CAS Registry Number

```
1  Chemical,CAS
2  Fragrance(s)/perfume(s),000000-00-1
3  Dye(s) (unspecified),000000-00-2
4  Enzyme(s) (unspecified),000000-00-3
5  Surfactant(s) (unspecified),000000-00-4
6  Lauryl polyglucose,000000-00-5
7  Solvent extracted mineral oil,000000-00-7
8  MSDS: Proprietary or trade secret ingredient(s),000000-00-8
9  Zinc compound(s) (unspecified),000000-00-9
```

Figure 8: Sample chemical search terms

4.2.2 SearchTerms.csv

`SearchTerms.csv` contains a list of search terms defined by our client.

- The "+" was a simple "and" condition
- <chemical> was a placeholder to search for any chemical in the above list

```
1  SearchSK    SearchTerm
2  0   male+infertility+<chemical>
3  1   salon+health+<chemical>
4  2   endocrine+disruptor+hormone
5  3   perfluorinated+compound
6  4   asthma+<chemical>
7  5   acrylic+paint+health+risk
8  6   targeted+ban+congressional+<chemical>
9  7   crop+hazard+<chemical>
```

Figure 9: Sample search terms from Eastman

4.2.3 TargetURLs.csv

`TargetURLs.csv` contains a list of domains that are closely related to chemicals and therefore likely to contain a high proportion of chemicals in the bodies of their pages.

```
SourceSK,URL,Domain
126,http://saferchemicals.org/,Sustainability
129,http://www.ecy.wa.gov/ecyhome.html,Sustainability
134,http://www.IHS.COM,Procurement
8,http://greenandcleanmom.org/,Sustainability
9,http://greenlivingguy.com/,Sustainability
10,http://greenpeaceblogs.org/category/toxics/,Sustainability
11,http://grist.org/,Sustainability
12,http://groovygreenlivin.com/,Sustainability
13,http://inthesetimes.com/,Sustainability
14,http://mindfulmomma.com/,Sustainability
15,http://myplasticfreelife.com/,Sustainability
16,http://naturallivingmamma.com/,Sustainability
```

Figure 10: URLs that Eastman suggested to filter on

4.2.4 Matches.csv

`Matches.csv` contains a sample of Eastman Chemical Company's search results where 'Relevance' matched URLs to 'SearchTerm's.

- Relevance is computed as 2/3 term match and 1/3 chemical name match.
- There are some false positives in here, possibly due to URL content changing, as well as some sites that are no longer active.

```
,SearchTerm,Relevance,Title,URL
1,male+infertility+<chemical>,1,COMUNICADO: Merck Launches More Than a Mother Campaign Aiming to
2,male+infertility+<chemical>,1,Merck KGaA : Launches "More than a Mother" Campaign Aiming to Red
3,male+infertility+<chemical>,1,"Perfluorooctanoic acid - Wikipedia, the free encyclopedia",http:
4,male+infertility+<chemical>,1,Publications - Health and Environment Alliance,http://www.env-hea
5,male+infertility+<chemical>,1,Publications - Health and Environment Alliance,http://www.env-hea
6,male+infertility+<chemical>,1,  Other Diseases,http://www.chemicalshealthmonitor.org/blog/other
7,male+infertility+<chemical>,1,Green Advocacy Archives - Page 2 of 16 - Groovy Green Livin,http:
8,male+infertility+<chemical>,1,foodconsumer.org - Other_Diseases,http://www.foodconsumer.org/new
9,male+infertility+<chemical>,1,HEAL Newsflash - November 2015 - Health and Environment Alliance,
```

Figure 11: Sample search results

4.2.5 run.sh

`run.sh` is a script used to remove any active Docker containers and then build and run a new container.

```
#!/bin/bash

# http://stackoverflow.com/questions/17236796/how-to-remove-old-docker-containers

# kill any old docker instances
docker kill $(docker ps -a -q)

# delete any old docker instances
docker rm $(docker ps -a -q)

# build the docker container
```

```
docker build -t commoncrawlmining .

# run the docker container
docker run -d -p 5000:5000 commoncrawlmining

# wait 10 seconds for the container to start up
sleep 10

# show status of docker containers
docker ps --all

# show the logs from all containers in case something crashed
docker logs $(docker ps -a -q)
```

Figure 12: Docker launch script

4.2.6 mrjob.conf

`mrjob.conf` is a configuration file for the mrjob library, such as in Figure 13.

```
    # https://pythonhosted.org/mrjob/guides/emr-opts.html
runners:
  emr:
    region: us-east-1
    # Set these or environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY
    aws_access_key_id: XXXXXXXXXXXXXXXXXXXX
    aws_secret_access_key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

    ec2_key_pair: mrjob
    ec2_key_pair_file: mrjob.pem

    subnet: subnet-f24877bb

    # https://aws.amazon.com/ec2/instance-types/
    num_core_instances: 1
    instance_type: m4.2xlarge
    master_instance_type: m4.large

    interpreter: python2.7

    upload_files:
    - test-1.warc

    bootstrap:
    - sudo yum install -y python27 python27-devel gcc-c++
    - sudo pip install 'requests>=2.13.0'
    - sudo pip install 'elasticsearch>=5.0.0,<6.0.0'
    - sudo pip install 'elasticsearch-dsl>=5.0.0,<6.0.0'
    - sudo pip install boto mrjob warc aws-requests-auth articleDateExtractor beautifulsoup4
    - sudo pip install https://github.com/commoncrawl/gzipstream/archive/master.zip
```

Figure 13: Mrjob configuration.

- “runners” and “emr” indicate that we are configuring mrjob to run jobs on ElasticMapReduce.
- “region” indicates that EMR clusters will be launched in the us-east-1 region, which is the same region that all our AWS resources are created in.
- “aws_access_key_id” and “aws_secret_access_key”, with their values redacted, configure the access keys that allow mrjob to launch AWS resources. These are associated with a particular IAM or root account and should be kept secret because they can result in a very large bill if misused. They can also be revoked if it’s suspected that an unauthorized user has gained access to them.
- “ec2_key_pair” and “ec2_key_pair_file” configure authentication for SSH connections to the master node, which provides things such as live job progress and fast error log fetching.
- “subnet” is set to the VPC subnet that contains our t2.micro instance because the instance types we want to use require us to use VPC.
- “num_core_instances” and “instance_type” configure the core nodes of our EMR clusters. These are the nodes that actually run the mappers and reducers, so they need to be pretty beefy. During testing, we’re only using one core node because we’re only using one WARC file and we’re not going to split a single WARC file into different mappers.
- “master_instance_type” configures the master node of our EMR cluster. This node handles things like scheduling and resource management. It doesn’t take much power to do this so we can get away with using the cheapest node we can, m4.large.
- “interpreter” makes sure that we’re using Python 2.7 rather than 2.6 or 3.x. The libraries and modules we’re using require Python 2.7.
- “upload_files” includes the file containing the paths to all the WARC files that our job will process; essentially our input file. This file is not automatically copied to the EMR cluster so we must do it explicitly here.
- “bootstrap” runs all the remaining commands we need to do to set up the execution environment in the cluster nodes. Our MapReduce job requires Python 2.7 as well as various dependencies so we make sure all of those are installed here.

4.2.7 proxy.sh

`proxy.sh` sets up a proxy that authenticates browser connections to Kibana using AWS keys.

```
#!/bin/bash

export AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXX
export AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
aws-es-kibana search-cces-d7dw54yl5rsstmn6xuz4qlad2e.us-east-1.es.amazonaws.com
```

Figure 14: Proxy script to run our system to open Kibana

4.2.8 es_indexer.py

`es_indexer.py` creates an EMR job that processes WARC records in a multiprocessing pool.

```
def process_payload(args):
    try:
        (payload, url) = args
        (headers, body) = payload.split('\r\n\r\n', 1)
        datestart = time.time()
        with stdout_redirected(to=sys.stderr):
            pd = articleDateExtractor.extractArticlePublishedDate(url,
                    body)
        dateend = time.time()
        if pd is not None:

            soupstart = time.time()
            soup = BeautifulSoup(body, 'html.parser')
            for script in soup(['script', 'style']):
                script.decompose()
            souptext = soup.get_text()
            soupend = time.time()
            indexstart = time.time()
            es.index(index='cc', doc_type='article', id=None,
                     body={'html': souptext, 'publication-date': pd,
                     'url': url})
            indexend = time.time()
            es.index(index='timers', doc_type='article', id=None,
                     body={'date': dateend - datestart, 'soup': soupend
                     - soupstart, 'index': indexend - indexstart})
            return 'daterecord'
        return 'htmlrecord'
    except:
        es.index(index='exception', doc_type='article', id=None,
                 body={'datetime': datetime.now(),
                 'traceback': traceback.format_exc()})
        raise
```

Figure 15: Function to process the payload.

4.2.9 app.py

`app.py` sets up a web service that can clear Elasticsearch indices or start indexing with mrjob.

```
@app.route('/start')
def start():
```

```
    def inner():
      #subprocess.Popen("python es_indexer.py -r emr --conf-path mrjob.conf --no-output --
output-dir s3://common-crawl-miner-data/output test-1.warc", shell=True)
      #return 'job started'
      # http://stackoverflow.com/a/15354403
      proc = subprocess.Popen("python es_indexer.py -r emr --conf-path mrjob.conf --no-
output --output-dir s3://common-crawl-miner-data/output test-1.warc", shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
      for line in iter(proc.stdout.readline,''):
        sleep(0.2)
        yield line.rstrip() + '<br>\n'

    return Response(inner(), mimetype='text/html')

@app.route('/clear')
```

Figure 16: Function to spawn new processes.

4.2.10 CC-MAIN-2017-09-warc.paths

CC-MAIN-2017-09-warc.paths contains links to all the WARC files that need to be processed. This is the February 2017 archive downloaded from commoncrawl.org which contains the paths of 65,200 WARC files. It's a simple text file containing the S3 path for a single compressed WARC file on each line. This is the input file for the MapReduce job; each Mapper is given one line at a time from this file.

```
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00000-ip-
10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00001-ip-
10-171-10-108.ec2.internal.warc.gz
```

```
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00002-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00003-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00004-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00005-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00006-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00007-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00008-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00009-ip-10-171-10-108.ec2.internal.warc.gz
crawl-data/CC-MAIN-2017-09/segments/1487501169769.33/warc/CC-MAIN-20170219104609-00010-ip-10-171-10-108.ec2.internal.warc.gz
```

Figure 17: MapReduce input file - S3 paths to WARC files

4.2.11 Dockerfile

Initializes the environment that app.py is executed in, including the installation of dependencies. This is a pretty standard Dockerfile, based on an ubuntu:16.04 image, it installs Pip and Python development libraries as well as all the Python libraries listed in requirements.txt. It finally executes app.py, whose HTTP endpoints are implicitly exposed on port 5000.

```
# https://runnable.com/docker/python/dockerize-your-flask-application

FROM ubuntu:16.04

#MAINTANER Casey Butenhoff "cjb@vt.edu"
```

```
RUN apt-get update -y && \
    apt-get install -y python-pip python-dev

# We copy just the requirements.txt first to leverage Docker cache
COPY ./requirements.txt /app/requirements.txt

WORKDIR /app

RUN pip install --upgrade pip

RUN pip install -r requirements.txt

COPY . /app

ENTRYPOINT [ "python" ]

CMD [ "app.py" ]
```

Figure 18: Configuration file for Docker container that runs app.py

4.3 Development Software

4.3.1 Setting up Python

In order to execute and compile Python source code, you must download Python 2.7 for your machine. You can do so by navigating to https://www.python.org/downloads/ and clicking the link corresponding to your OS. Be sure that the Python folder is added to your PATH so that your terminal can locate the Python executable.

4.3.2 Installing pip

For Windows users:

https://github.com/BurntSushi/nfldb/wiki/Python-&-pip-Windows-installation

For Mac uses:

Run the following command in the terminal: `sudo easy_install pip`

For Linux users:

For Ubuntu users, run this command in the terminal: `sudo apt-get install python-pip`
For CentOS users: run this command: `sudo yum install python-pip`

Alternatively, you can download the `get-pip.py` script at:
https://bootstrap.pypa.io/get-pip.py
You can execute this script using Python to automatically install pip.

4.3.3 Dependencies

A list of the dependencies necessary to run the source code can be found in `requirements.txt`:

```
Flask==0.10.1
boto
mrjob
warc
https://github.com/commoncrawl/gzipstream/archive/master.zip
elasticsearch>=5.0.0,<6.0.0
elasticsearch-dsl>=5.0.0,<6.0.0
aws-requests-auth
articleDateExtractor
beautifulsoup4
```

Figure 19: Source code dependencies.

5 Lessons Learned

5.1 Timeline and Schedule

To organize the project, we followed the timeline shown in Figure 20. A detailed description of the progress made on each task is given below.

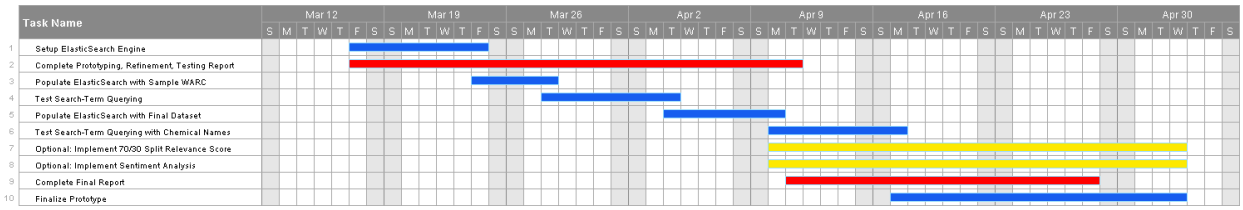


Figure 20. Project Timeline

5.1.1 Set up Elasticsearch Engine (Complete)

The Elasticsearch engine was set up through AWS to support the design decisions illustrated in Figure 1 and Figure 2. The components were provisioned based on our complexity, cost, and resource analysis. See Appendix D for specific details on how the prototype was set up.

5.1.2 Complete Prototyping, Refinement, and Testing Report (Complete)

The Prototyping, Refinement, and Testing Report was completed through regularly scheduled team meetings and individual contributions via the Google Docs collaboration tool.

5.1.3 Populate Elasticsearch with Sample WARC (Complete)

A sample WARC file was selected based on its frequency of records from interesting domains. These interesting domains were provided by the client. A MapReduce job extracted records with publication dates from the sample WARC file and indexed them into the Elasticsearch engine. See Appendix D for specific details on the MapReduce job and interesting domains.

5.1.4 Test Querying (Complete)

Basic search term testing was conducted on the records ingested from the sample dataset. Search terms were retrieved from a sample dataset provided by the client. The retrieved search terms were queried against the sample WARC in Elasticsearch to ensure that:

1. The results did contain the search term.
2. The sample WARC contains some records of interest.

See section 4.2 for specific details on the search terms. See Appendix E for more information on our testing procedures.

5.1.5 Populate Elasticsearch with Final Dataset (Complete)

The final dataset was ingested into the final prototype from the Common Crawl February 2017 archive. A parallelized MapReduce job was used on a subset of WARC files from this month that have a high frequency of records from relevant domains. From this subset, we extracted the records that have publication dates.

5.1.6 Test Querying with Chemical Names (Complete)

Additional tests were conducted on the records ingested from the final dataset using both search terms and chemical names. Search terms and chemical names were provided by the client. See the file inventory for specific details on the search terms and chemical names. See Appendix E for more information on our testing procedures.

5.1.7 Optional: Implement 70/30 Split Relevance Score (Complete)

Although search term querying with additional chemical names is conceptually the same as searching for multiple search terms, we figured out how to assign custom weights to individual terms in our query through Elasticsearch's Function Score Query[33]. This allows us to implement the split relevance scoring on an as-needed basis to assign higher relevance to search results that contain both search terms and chemical names. See Appendix F for more information on custom relevance scores using Elasticsearch's Function Score Query.

5.1.8 Optional: Implement Sentiment Analysis (Future Work)

Sentiment analysis can be performed using classification models like Logistic Regression, Naive Bayes or Support Vector Machines. These models can be used on a TF/IDF or Word Vector representations of the data. Eastman did not have training data for these models so it would have had to be generated by hand, which we did not have the resources to do, so this feature was not implemented.

5.1.9 Complete Final Report (Complete)

The Final Report was completed through regularly scheduled team meetings and individual contributions via the Google Docs collaboration tool.

5.1.10 Finalize Prototype (Complete)

The team finalized the Common Crawl Mining prototype and provided the client with the Final Report and Final Presentation.

5.2 Problems and Solutions

5.2.1 Getting Started

During the development of the Common Crawl Mining system we ran into a number of issues.

Early on in the project, the team had trouble establishing contact with the client, Eastman Chemical Company. The initial project specification was limited in details and had some vague wording. It suggested that the client simply needed assistance writing code for publication date extraction and querying pages; both of which already have open-source software packages available. As such, the team was eager to speak with the client for clarification and specification.

Being unable to contact the client, we had to make some assumptions as to what the system would look like and what the use cases would be. We initially viewed the system as a sentiment analyzer involving natural language processing along with classification algorithms. So we started doing research into tools like Stanford Parser and Scikit-Learn. We also had some idea of the size of the project and felt that we would have to use AWS at some point, so we also looked into that.

5.2.2 Scalability

A large problem that the team ran into after clarifying the system requirements was scalability. At this early stage, the prototype was a Python script that created a MapReduce job to query a single WARC file for a single chemical name. This was taking multiple hours and did not fit well into an API paradigm where responses need to be immediate. We considered a ticketing system where tickets were generated for each API request, but users would have had to continuously check their job to find out if it had completed. We considered converting this to a batch processing system where a batch of queries would be executed periodically. The results of queries would be stored in a warehouse where users could immediately retrieve their pre-processed query results. This was still infeasible, however, due to the long duration of such MapReduce jobs.

Instead, we decided on an Elasticsearch search engine that would periodically index new Common Crawl monthly archives. Due to their large size, oftentimes greater than 250 TB, it was unrealistic to try and index an entire month during the development process. In order to scale down the amount of information indexed from each batch, we narrowed the search space down to HTTP responses from interesting domains. In addition, we required responses to have HTML data with an extractable publication date. We selected CC-MAIN-20170219104611-00388-ip-10-171-10-107.ec2.internal.warc.gz as the WARC file we would use because it contains 138,257 records, 43,748 of which are HTTP responses containing HTML and a high proportion of records from domains that Eastman told us were particularly relevant. When the date extractor was applied to each of the 43,748 HTML response records in the test WARC, it returned a date for 11,031 of them (about 25.2%).

5.2.3 Configuration

After the specifications had been cleared, we started experimenting with the Common Crawl dataset. Our initial attempts to get a prototype up and running were hindered by the system crashes that were related to relative unfamiliarity with major architectural components such as Elastic MapReduce. The first problem we ran into was the EMR cluster crashing because the cluster’s master and core nodes were too small. We were initially using small and medium

nodes with the goal of limiting development costs but discovered that we had to use a large node before the cluster would initialize properly. Another problem we ran into was that the default timeout for indexing requests to Elasticsearch were too short which was causing the requests to time out and throw an exception that crashed the EMR job before it had finished processing the files. Increasing the Elasticsearch request timeout fixed this issue.

5.2.4 Date Extraction

Predicting publication date of web pages is an essential part of the Common Crawl Mining system, but no particular method was specified by the client. So, we had some leeway in terms of implementation. We researched several methods to predict this date, including:

- Assuming the date it was first indexed on Google was probably close to the publication date and getting it by querying the page URL, appending “&as_qdr=y15” to the query
- Assuming the date it was first archived by Common Crawl was probably close to the publication date and getting it by searching previous indices
- IBM Bluemix Natural Language Understanding API (formerly Alchemy API)
- CarbonDate web service
- Webhose.io article-date-extractor Python module

The problem with using Google for this is that Google doesn’t provide an API for it. We would have to perform a separate web search query for each page and scrape the date out of the result page. Google is almost certain to identify this sort of automated use as abusive and restrict our system’s access.

Searching previous Common Crawl indices is straightforward by using their CDX Server API, but the main problems with this method are again not only the large number of queries but also poor precision. At best, we can infer that a page was probably published sometime within the one-month window of the first index that contains it.

The Alchemy API (predecessor to the IBM Bluemix NLU API) was the solution used by the predecessor to our project for sentiment analysis but it does date extraction as well. We determined that it’s not feasible because the cost is multi-tiered based on the number of calls per month (down to a lower limit of $0.0002 USD/NLU Item after 5 million calls[29]) which precluded us from using it because it would cost hundreds of thousands of dollars to process the Common Crawl index for a single month.

CarbonDate is a research-backed method of extracting publication date from web pages. Unfortunately, like the Alchemy API, it’s very expensive to use. It depends on the Bing search API[30] which is priced based on the number of calls per month (up to a limit of 10M calls costing $27,000 per month[31]).

We ultimately decided to use the article-date-extractor Python module because it promised “close to 100% precision with 90% recall”[28] on articles in the Google news feed. We tested it with the CC-MAIN-20170219104611-00388-ip-10-171-10-107.ec2.internal.warc.gz WARC file

that we identified earlier and came to the conclusion that it produced minimally acceptable results in a reasonable time which seemed to be at least a good starting point.

5.2.5 Indexing

After we configured our prototype and selected our sample dataset, we needed to index the relevant information. We created the es_indexer.py Python script to process WARC records from Common Crawl and index them into our Elasticsearch engine. During the indexing process, we discovered that the body text extractor in our first prototype included a lot of things we didn't want to index, such as script tags and JavaScript code. To fix this issue we used the *decompose* method provided by Beautiful Soup. It allowed us to get rid of any style or script tags.

Another issue we ran into during the indexing process was extraneous output being printed to the STDOUT file descriptor. Whenever we tried to use the article-date-extractor module within the mapper, the reducer would crash. Lengthy debugging led us to discover that the article-date-extractor module was printing text to STDOUT each time it was called in the mapper. The mrjob Python library was assuming this output was the mapper's result and giving it as input to the reducer. The reducer was proceeding to crash because this input wasn't valid JSON. This problem was solved by redirecting the file descriptor of standard output to the file descriptor of standard error for the block of code that contains the article-date-extractor call.

5.2.6 Parallelization

We couldn't come up with any good way to give anything smaller than a single WARC file to each mapper, but even a single WARC file contains hundreds of thousands of records. To make the system run faster, we use parallelization in the mapper step. We used multiprocessing rather than multithreading since the multithreading library in Python proved to be slower than multiprocessing because CPython's global interpreter lock causes multiple threads to be executed sequentially. A process pool containing one process per CPU is given 128 records at a time. Even though multiprocessing utilized each mapper's CPU resources more completely and resulted in an improvement in the time taken to process a file, the improvement was not very significant. For instance, for one WARC file, it took 2 hours 22 minutes without multiprocessing while it took 1 hour and 46 minutes with multiprocessing.

5.2.7 Deduplication

Common Crawl performs fuzzy deduplication during the crawl process for individual monthly batches, but it is unclear if deduplication is performed across these batches. Therefore, we wanted to ensure that new versions of records replaced outdated versions in our Elasticsearch prototype. To achieve this, we index records by their URL. If a matching URL is found to exist during the indexing process, then the new record replaces the existing record. We can be certain the new record is indeed the latest version by chronologically indexing Common Crawl batches. Since proceeding batches will have versions of records that were crawled after preceding batches, then proceeding batch records must be newer and we can avoid the

expensive prospect of querying for an existing record and comparing their publication dates.

5.3 Future Work

Moving forward, one of the things that can be looked into is increasing the percentage of records for which publication date was extracted. For one of the WARC files we tested, we found 43,748 HTML response records and returned a date for 11,031 of them (about 25.2%). This might be something that might not be acceptable since a lot of valuable information might be lost. A way to fix this would be to write a date extractor using regular expressions and string matching but this comes at the cost of time required to process a record thoroughly. The date extraction step could also be skipped entirely and this would fix the issue. But if the aim is to visualize trends over time, this might not be possible.

Ideally, low-quality pages wouldn't be processed or indexed at all. This is a pretty domain-specific application so targeting all of Common Crawl might not be the best strategy. Ingesting data from the RSS feeds of relevant scientific journals or even from the Common Crawl CC-NEWS feed might focus the effort on more high-quality pages and avoid processing all the spam pages and irrelevant content of the greater web.

Another thing that might be implemented in the future is sentiment analysis. We could classify the records as either being positive or negative using algorithms like SVMs, Naive Bayes, Logistic Regression, or Neural Networks. We could also use *word2vec* representations to get a better representation of the records. This task would require training data and might need to be hand labelled.

6 Conclusions

This system is potentially able to deliver valuable insights through queries on a vast dataset that's been annotated with inferred publication dates. Given limited information about particular user needs and usage scenarios, we designed and built it to be as general-purpose and flexible as possible. Unfortunately, due to budget constraints we were unable to load a full data set into the system and had to infer a lot about its performance and usefulness at scale. The queries we performed on the development instance of our system, which was populated with the contents of a single WARC file, didn't seem to turn up much useful information, but we think trends are likely to emerge from a much larger set of data that weren't apparent in the subset we worked with. Kibana provides a flexible visualization system that's likely to be particularly useful to end-users once particular information needs are determined and visualizations are built to accommodate them.

The most significant issue that is likely to prevent this system from being deployed to users is cost. We can summarize the cost analysis we performed in Appendix B as follows: processing and indexing 3 billion pages, the size of a single monthly Common Crawl archive, with our system would have a one-time cost of about twenty-two thousand dollars and maintaining that indexed data in an Elasticsearch cluster would have an ongoing cost of about seven thousand dollars per month. This cost could potentially be reduced somewhat using strategies such as restricting indexed pages to a subset of domains or to pages that contain particular words.

Other strategies could be used to control costs and possibly deliver information that's even more useful to users. It might be advisable to revisit and more closely examine the exact nature of the information needs of the users; a different set of use cases and requirements could lead to a system that's wildly different from the one described in this document. For example, if the users are most interested in being made aware of sudden changes in public sentiment or attention as they occur, then this system is unlikely to meet their need at all as it operates on data that is at least one month old. In that case, it might be more appropriate to leverage the Common Crawl CC-NEWS feed, science journal feeds, or even social media in order to ingest more recent data. Such a system could have the capability to identify trends as they emerge, possibly even the same day, and could even have much lower operating costs as it could be targeted exclusively to news pages and particular subjects.

Clearly there are a lot of possible directions for Eastman to go from here. Picking one is a matter of distilling the most essential needs of the users and determining what meeting those needs is worth in order to establish resource constraints for both ongoing development and maintenance of this system.

7 Acknowledgments

Included in this section are acknowledgments of groups and individuals who contributed to the Common Crawl Mining project.

Eastman Chemical Company
Eastman Chemical Company graciously provided funds for this project and assigned the team two mentors, Ken Denmark and Don Sanderson, to guide them through the project. The final prototype was designed with Eastman Chemical Company in mind to suit their needs.

Ken Denmark
denmark@eastman.com
Mr. Denmark works as an Analytics Associate for Eastman Chemical Company in their Business Analytics Group; Eastman Chemical Company provided funding for this project. He acted as primary client of the Common Crawl Mining system.

Dr. Edward Fox
fox@vt.edu
Dr. Fox is the instructor for the Multimedia, Hypertext, and Information Access (CS 4624) course at Virginia Polytechnic Institute and State University. He coordinated with Eastman Chemical Company to launch the Common Crawl Mining project. He guided the team through all of the project phases and provided useful insight into project direction and specific technologies that were used to develop the Common Crawl Mining system.

Liuqing Li
liuqing@vt.edu
Mr Li. is a PhD student in the Virginia Polytechnic Institute and State University computer science department. He works with the Digital Library Research Laboratory (DLRL). He met with members of the team to discuss loading Common Crawl Data into the DLRL cluster.

Don Sanderson
donsanderson@eastman.com
Mr. Sanderson works as a Service Manager for Eastman Chemical Company in their Marketing Solutions Group; Eastman Chemical Company provided funding for this project. He served as project manager for the Common Crawl Mining project.

8 References

[1]: Common Crawl. Common Crawl, 2017 Web. http://commoncrawl.org/the-data/get-started/ 19 Feb. 2017. http://commoncrawl.org/the-data/get-started/

[2]: "What Is Amazon EC2? - Amazon Elastic Compute Cloud". *Docs.aws.amazon.com*. Amazon, 2017. Web. 13 Mar. 2017. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

[3]: "Amazon Simple Storage Service (S3) — Cloud Storage — AWS". *Amazon Web Services, Inc.*. Amazon., 2017. Web. 13 Mar. 2017. https://aws.amazon.com/s3/

[4]: "Amazon EMR – Amazon Web Services". *Amazon Web Services, Inc.*. Amazon, 2017. Web. 13 Mar. 2017. https://aws.amazon.com/emr/

[5]: "Download — Python 2.7.13 Documentation". *Docs.python.org*. Python, 2017. Web. 13 Mar. 2017. https://docs.python.org/2/download.html

[6]: "Mrjob — Mrjob V0.5.8 Documentation". *Pythonhosted.org*. Mrjob, 2017. Web. 13 Mar. 2017. https://pythonhosted.org/mrjob/

[7]: "Welcome | Flask (A Python Microframework)". *Flask.pocoo.org*. Flask, 2017. Web. 13 Mar. 2017. http://flask.pocoo.org/

[8]: "Webhose.Io". *GitHub*. Webhose, 2017. Web. 13 Mar. 2017. https://github.com/Webhose

[9]: "Webhose/Article-Date-Extractor". *GitHub*. Webhose, 2017. Web. 13 Mar. 2017. https://github.com/Webhose/article-date-extractor

[10]: "What Is Docker? – Amazon Web Services (AWS)". *Amazon Web Services, Inc.*. Amazon, 2017. Web. 13 Mar. 2017. https://aws.amazon.com/docker/

[11]: "Mongodb Documentation". *Docs.mongodb.com*. Mongodb, 2017. Web. 13 Mar. 2017. https://docs.mongodb.com/?_ga=1.53726184.1583110366.1489444878

[12]: "Scikit-Learn: Machine Learning In Python — Scikit-Learn 0.17.1 Documentation". *Scikit-learn.org*. Scikit-Learn, 2017. Web. 13 Mar. 2017. http://scikit-learn.org/stable/

[13]: "Build Software Better, Together". GitHub.com. Github, 2017. Web. http://www.github.com 17 Mar. 2017.

[14]: C, Ben. "Yelp." Analyzing the Web for the Price of a Sandwich. CommonCrawl, 20 Mar. 2015. Web. https://engineeringblog.yelp.com/2015/03/analyzing-the-web-for-the-price-of-a-sandwich.html 19 Feb. 2017.

[15]: "Dockerize Your Flask Application." Runnable Docker Guides. runnable, 19 July 2016. Web. https://runnable.com/docker/python/dockerize-your-flask-application 19 Feb. 2017.

[16]: "Elastic MapReduce Using Python and MRJob." Elastic MapReduce Using Python and MRJob - Cs4980s15. MoinMoin, 2015-02-12 Web. https://weblog.cs.uiowa.edu/cs4980s15/Elastic%20MapReduce%20using%20Python%20and%20MRJob 19 Feb. 2017.

[17]: ESA Software Engineering Standards (ESA PSS-05-0 Issue 2), ESA Board for Software Standardization and Control (BSSC), Feb 1991 http://wwwis.win.tue.nl/2R690/doc/PSS050.pdf

[18]: Jrs026. "CommonCrawlMiner." GitHub. Jrs026, 2012 Web. https://github.com/jrs026/CommonCrawlMiner 19 Feb. 2017.

[19]: Kleijkers, Tom. Software Engineering Project (2IP40) User Requirements Document. Rep. Vol. 1.0.0. Eindhoven: Technische Informatica, Eindhoven U of Technology, Eindhoven, 2006. Print. http://www.academia.edu/9021460/Software_Engineering_Project_2IP40_Software_Requirements_Document

[20]: Merity, Stephen "Common Crawl: Making Python Work for Big Web Data by Smerity." Slides. Merity, Stephen, n.d. Web. http://slides.com/smerity/a-python-the-spiders-web#/ 19 Feb. 2017.

[21]: Merity, Stephen. "Navigating the WARC File Format." Common Crawl. CommonCrawl, 2 Apr. 2014. Web. http://commoncrawl.org/2014/04/navigating-the-warc-file-format/ 19 Feb. 2017.

[22]: Nagel, Sebastian. Crawling the Web for Common Crawl. Powerpoint. Common Crawl, 2016. Print. http://events.linuxfoundation.org/sites/events/files/slides/aceu2016-snagel-crawling-web-commoncrawl.pdf

[23]: Namin. "Linkrev." GitHub. Namin, 15 Aug. 2012. Web. https://github.com/namin/linkrev/tree/master/project 19 Feb. 2017.

[24]: Robert Meusel, Researcher, Data Scientist Follow. "Mining a Large Web Corpus." Share and Discover Knowledge on LinkedIn SlideShare. Linkedin, 23 May 2014. Web. https://www.slideshare.net/RobertMeusel/mining-a-large-web-corpus 19 Feb. 2017.

[25]: Ssalevan. "MapReduce for the Masses: Zero to Hadoop in Five Minutes with CommonCrawl." YouTube. YouTube, 16 Dec. 2011. Web. https://www.youtube.com/watch?v=y4GZ0Ey9DVw 19 Feb. 2017.

[26]: "What Is Docker? – Amazon Web Services (AWS)." Amazon Web Services, Inc. Amazon, n.d. Web. https://aws.amazon.com/docker/ 19 Feb. 2017.

[27]: "Cheaper In Bulk | Elasticsearch: The Definitive Guide [2.X] | Elastic". *Elastic.co*. 2017. Web. https://www.elastic.co/guide/en/elasticsearch/guide/current/bulk.html. 17 Mar. 2017.

[28]: “Article’s publication date extractor – an overview”. Webhose.io. 13 Dec 2015. Web. http://blog.webhose.io/2015/12/13/articles-publication-date-extractor-an-overview/. 9 Apr. 2017.

[29]: “Natural Language Understanding”. IBM. 7 Apr. 2017. Web. https://console.ng.bluemix.net/catalog/services/natural-language-understanding. 9 Apr. 2017.

[30]: “oduwsdl/CarbonDate: Estimating the age of web resources”. GitHub. 5 Oct. 2016. Web. https://github.com/oduwsdl/CarbonDate. 9 Apr. 2017.

[31]: “Microsoft Cognitive Services - Bing Web Search API”. Microsoft. https://www.microsoft.com/cognitive-services/en-us/bing-web-search-api. 9 Apr. 2017.

[32]: “FAQ - aol/moloch Wiki”. GitHub. 7 Apr. 2017. Web. https://github.com/aol/moloch/wiki/FAQ#How_many_elasticsearch_nodes_or_machines_do_I_need. 9 Apr. 2017.

[33] “Function Score Query”. Elastic. 10 Apr. 2017. Web. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html. 10 Apr. 2017.

[34] “CommonCrawl Index Server”. CommonCrawl. 10 Apr. 2017. Web. http://index.commoncrawl.org/. 10 Apr. 2017.

[35] “Kibana: Explore, Visualize, Discover Data”. Elastic. 10 Apr. 2017. Web. https://www.elastic.co/products/kibana. 10 Apr. 2017.

[36] Elastic Co. "Queries." Queries | Kibana [3.0] | Elastic. Elastic Co., 2017. Web. 06 May 2017.

[37] Lucene. "Apache Lucene - Query Parser Syntax." Apache.org. The Apache Software Foundation., 21 June 2013. Web. 06 May 2017.

9 Appendices

A Requirements

A.1 General Information

A.1.1 General Constraints

The quality of the Common Crawl Mining system is primarily constrained by the amount of time the project team has available as well as the presence of various deadlines. It is also constrained by a limited budget for commercial software components and AWS resources.

A.1.2 Assumptions

It is assumed that various Amazon AWS services will be available for the duration of the development and testing phases of this project, and for the foreseeable future to support the deployed release instance of the software.

A.1.3 Dependencies

The Common Crawl Mining system depends on the continued operation and availability of Common Crawl. It also depends on the sufficiency of the data provided by Eastman Chemical Company for the purposes of developing this system. Any data -- including search terms of interest, chemical names, and previously matched data -- that is required for developing this system should be provided to the development team.

A.2 Users

Users of the Common Crawl Mining system are split into two classes, marketers and strategy developers. Both users will engage in the same use cases, but for different reasons that are discussed in the description of the respective user class. Note that a user is not necessarily human; computer systems may also request the system to execute some behavior.

A.2.1 Marketer

Marketers are interested in knowing what Eastman’s current customers’ concerns are and making sure the company is attending to them. Marketers must be made aware of upcoming public concerns so Eastman is not blindsided by them. Marketers must be able to proactively tell people when their concerns do not apply to Eastman’s products (for example, that Eastman products do not contain harmful bisphenol A (BPA) chemicals). Marketers must also be able to recognize when there is a legitimate concern about an Eastman product so that the company can react to the concern and mitigate it. Marketers also need to manually review web pages related to these insights in order to understand their context.

A.2.2 Strategy Developer

This user has long-term and big-picture concerns; they need to explore and assess which markets Eastman might want to enter or grow into. They must be aware of negative public perceptions and major issues that already exist in that market, as well as their historical trends. Current and former concerns to the public are important to be aware of, whether to understand the risks involved for Eastman products or to leverage Eastman's superior product characteristics. Strategy developers are also interested in maintaining market share, so they must be aware of changes in overall interest in particular chemicals, and hence the company. Like marketers, strategy developers need to be able to review the content of relevant web pages in order to have a more complete understanding of the context of any insights.

A.2.3 Use Cases

A use case is a piece of functionality in the system that represents a user doing something and the system responding in some way, typically by returning a value to the user. Note that a user is not necessarily human; computer systems may also request the system to execute some behavior.

A.2.3.1 UC_1010

Scenario: *The user requests information about a particular search term.*
User: *Marketer or Strategy Developer*
Actions:

1. The user provides the Common Crawl Mining system with a search term.
2. The system matches the given search term against the records in the existing data set.

3a. There is at least one page containing the search term.

   a. The system returns historical metadata for each match including but not limited to URL, text, and publication date. A sentiment score may be returned as well. Matches are given 100% relevance.
   b. The use case ends successfully

3b. There are no pages containing the search term.

   a. The system returns an error code.
   b. The use case ends with a failure condition.

A.2.3.2 UC_1020

Scenario: *The user requests information about a particular search term and a relevant chemical name.*
User: *Marketer or Strategy Developer*
Actions:

1. The user provides the Common Crawl Mining system with a search term and a relevant chemical name.
2. The system matches the given search term and chemical name against the records in the existing data set.

3a. There is at least one page containing the search term and/or chemical name.

   a. The system returns historical metadata for each match including but not limited to

URL, text, publication date, and a relevance score. A sentiment score may be returned as well.

b. The use case ends successfully

3b. There are no pages containing the search term.

a. The system returns an error code.

b. The use case ends with a failure condition.

A.3 User Requirements

User requirements are prefixed with a “UR” label to distinguish them from software requirements. A priority level is assigned to each of these requirements as a number between 1 and 5, with 1 representing the highest priority. User requirements with priority level 1 and 2 are required to be implemented. Other requirements will be implemented, in priority order, if time allows.

A.3.1 Functional Requirements

Functional requirements define what the system is supposed to do. They will inform the design of the system. These requirements are generally in the form of *“the system shall do …”* which describe the functions of the system as if they were black boxes, with no regard for the details of how that requirement might be satisfied. The functional requirements for the Common Crawl Mining system are listed in Table 3.

Table 3. Functional User Requirements

ID	Description	Priority
UR_0010	*The Common Crawl Mining system shall deliver analysis results to users in response to search queries.*	1
UR_0011	*When a query is made on the system, it shall respond with information that the user can use to monitor the state of their query.*	1
UR_0012	*The system shall respond immediately to requests with information about whether the request can be satisfied successfully or not.*	1
UR_0013	*The system shall always provide some sort of immediate response to requests.*	1
UR_0014	*In addition to an immediate response in reaction to search queries, the system shall deliver query results asynchronously (potentially at a much later time).*	1
UR_0020	*The system shall accept a search term as an input and return the URL, text, publication date, and relevance score of the pages that contain it.*	1
UR_0030	*The system shall accept a search term and chemical name as an input and return the URL, text, publication date, and relevance score of the pages that contain one or both of them.*	2
UR_0040	*The system shall compute a sentiment score between -1 and 1 for each page returned to the user.*	5
UR_0050	*If only a search term is provided as part of a query, the system shall compute the relevance score of each matching page.*	1
UR_0060	*If a search term and chemical are provided as part of a query, the system shall compute the relevance score of each matching page.*	1
UR_0070	*The system shall support up to three main users.*	1

A.3.2 Non-functional Requirements

Non-functional requirements define how the system is supposed to be. They are architecturally significant and will inform the design of the system architecture. These requirements are

generally in the form of *“the system shall be…”* which describe overall properties of the system rather than specific functions. This includes characteristics of the system that are observable at run time, such as security and usability. The non-functional requirements for the Common Crawl Mining system are listed in Table 4.

Table 4. Non-functional User Requirements

ID	Description	Priority
UR_8010	*The system shall be easily extensible for other use cases.*	1
UR_8015	*The maximum amount of time that may elapse before the system acknowledges a request is 1 second.*	1
UR_8020	*The system shall be reliable in that it will provide correct results if they exist at all.*	1
UR_8030	*The system shall have proper and sufficient documentation for end users and developers.*	1
UR_8040	*Resource constraints are those of number of articles maintained in Common Crawl, network bandwidth, memory, and processor speed.*	1
UR_8050	*Users shall not need to know anything about the inner workings of the system in order to use it.*	1
UR_8060	*The system shall properly handle failures by terminating the query and providing the user with the cause of failure. Failures may be triggered by invalid queries, insufficient data, or some broken component.*	1

A.4 Software Requirements

Software requirements are prefixed with an “SR” label to distinguish them from user requirements. As with user requirements, a priority level is assigned to each software requirement as a number between 1 and 5, with 1 representing the highest priority. The software requirements for the Common Crawl Mining system are listed in Table 5.

Table 5. Software Requirements

ID	Associated User Requirement	Description	Priority
SR_0010	N/A	*The Common Crawl Mining system shall analyze web crawl data retrieved from Common Crawl.*	1
SR_0020	N/A	*The system shall store analysis results in a NoSQL database.*	2
SR_0030	UR_0050	*If only a search term is provided as part of a query, the system shall compute relevance scores of each matching page in the range from 0 to 1 as the ratio of search term elements that are present in the page. Search term elements include a multi-token search term and zero or more associated multi-token chemical names.*	1
SR_0040	UR_0060	*If a search term and chemical are provided as part of a query, the system shall compute the relevance score of each matching page in the range from 0 to 1 as the sum of 0.3 (if the queried chemical is present in the page) and 0.7 (scaled by the ratio of search term elements that are present in the page).*	1
SR_0050	UR_0010	*The system shall expose data via a REST API.*	1
SR_0060	UR_0070	*The system shall support up to two queries per month per user.*	1

B Design

B.1 Components & Tools

B.1.1 Common Crawl[1]

Common Crawl is an organisation that crawls the web and provides its archives and datasets without any cost. It crawls the entire web four times a year. Common Crawl was founded by Gil Elbaz and is advised by Peter Norvig.

The Common Crawl corpus contains data from the past 7 years including raw web page data, metadata extracts and text extracts. The corpus is stored on Amazon Web Services. The data is stored in WARC format.

WARC[1], WAT[1], and WET[1] file formats are stored in the Common Crawl repository and are the file types with which we will directly interact to gather information. WARC files store the raw data from the source. WAT files store metadata about those articles. WET files store the literal plaintext from the articles. The team considered using WET files for uploading plaintext articles into Elasticsearch, but it would create unnecessary overhead during the indexing process; the plaintext is already parsed from WARC records after extracting publication dates. Utilizing WET files does not provide any benefit to the Common Crawl Mining system at this time.

A WARC file contains several WARC records. At the beginning of a WARC file, you can find a ‘warcinfo’ record type that contains header information regarding the WARC file. Following that record are other record types, specifically of note are the ‘request’, ‘response’ and ‘metadata’ record types. The ‘request’ record stores information regarding how the URL was requested. The ‘response’ record stores the HTTP response information. The ‘metadata’ record type stores the actual metadata from the crawl itself.

B.1.2 Amazon Elastic Compute Cloud (EC2)[2]

Amazon EC2 is the system we will use for our server to host the jobs we run to search Common Crawl.

B.1.3 Amazon Simple Storage Service (S3)[3]

Amazon S3 providers large-scale cloud-based storage. Common Crawl stores its web-crawl data in S3.

B.1.4 Amazon Elastic MapReduce (EMR)[4]

Amazon EMR provides Map Reduce (MR) functionality. EMR will provision a Hadoop cluster, run and terminate jobs, and transfer associated data between EC2 and S3.

B.1.5 Python[5]

We will be using Python 2.7 to develop the system that searches Common Crawl and provides results to users.

B.1.5.1 mrjob[6]

mrjob is a Python package that supports MR functionality. It allows multi-step MR jobs to execute on Hadoop clusters running in EMR. It is used by the Common Crawl Mining System to ingest WARC files from Common Crawl and perform search term queries on those files in distributed environments.

B.1.5.2 Flask[7]

Flask is an open source WSGI container microframework for Python that will be providing the REST API. The container will be hosted by gunicorn and nginx.

B.1.5.3 Webhose[8]

Webhose is an open source Python module to help extract publication dates from online articles. This Article Date Extractor relies on Beautifulsoup and assures over 90% success rate[9].

B.1.6 Docker[10]

Docker is an open-source technology that allows you to build, run, and deploy applications inside software containers. It allows you to package a piece of software in a standardized unit for software development, containing everything the software needs to run: code, runtime, system tools, system libraries, etc. Docker enables you to quickly, reliably, and consistently deploy applications regardless of environment.

B.1.7 MongoDB[11]

MongoDB is an open-source document-oriented database program. It is classified as a NoSQL database and uses JSON-like documents with schemas.

B.1.8 Scikit Learn[12]

Scikit learn is an open-source tool for data mining and data analysis. It is written in Python and built on NumPy, SciPy and matplotlib. It will possibly be used as part of the implementation of sentiment analysis.

B.1.9 GitHub[13]

GitHub is a web-based version control repository. It is being used as a collaboration tool. We use it to create branches in which we store our code.

B.1.9 Common Crawl Index (CDX)[34]

CDX files are returned by Common Crawl's custom indexing system. The system provides a query API designed specifically for the Common Crawl archive. Queries can be executed on monthly batches of Common Crawl data. The results of queries are CDX files. The first line of a CDX file is a legend that is used to interpret the remaining data in the file. The remaining lines correspond to matching records from the queried batch.

B.1.9 Kibana[35]

Kibana is a user interface and visualization tool for Elasticsearch. It is used to provide an interface to the Common Crawl Mining system. Access policies are used to control access to the user interface. It works as a whitelist to provide only certain IP addresses with access.

B.2 System Architecture

There are four major architectural components to the Common Crawl Mining system illustrated in Figure 21:

1. An EC2 instance which hosts the Common Crawl Mining Indexing Supervisor (CCMIS). The CCMIS is a Python program deployed in a Docker container on EC2; Docker greatly simplifies deployment of this component and makes most of the deployment process self-documenting.
2. The Common Crawl S3 bucket which contains the Web archive data we are processing,
3. An Elasticsearch cluster which will provide full-text search and user interface of web page data and metadata, and
4. An EMR cluster which will analyze web archive data from the S3 bucket and send the analysis results to Elasticsearch for storage and indexing. The EMR cluster is a transient component that will be created and destroyed as needed through the use of the mrjob framework.

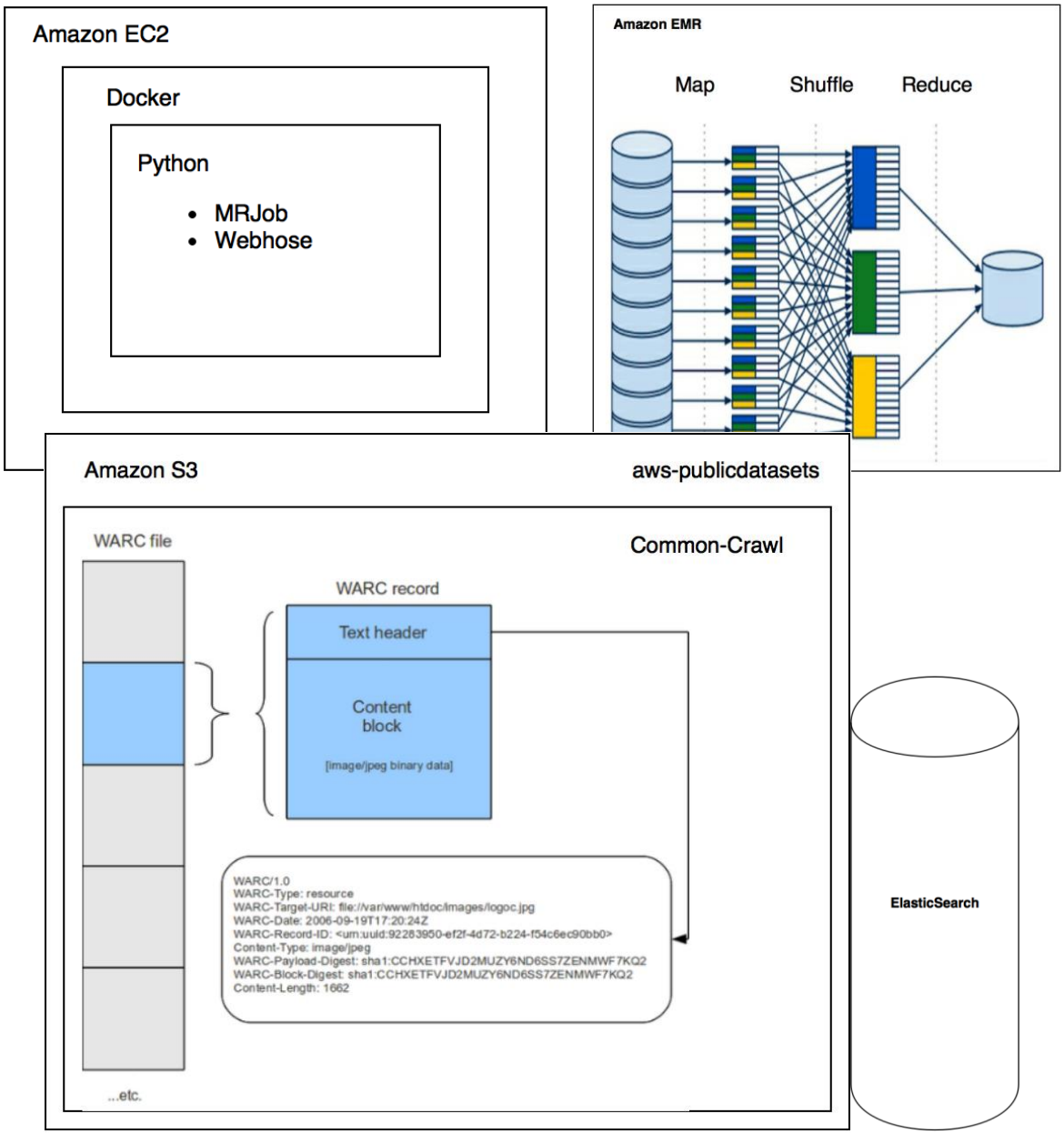


Figure 21. Architecture Diagram

B.3 Data Flow

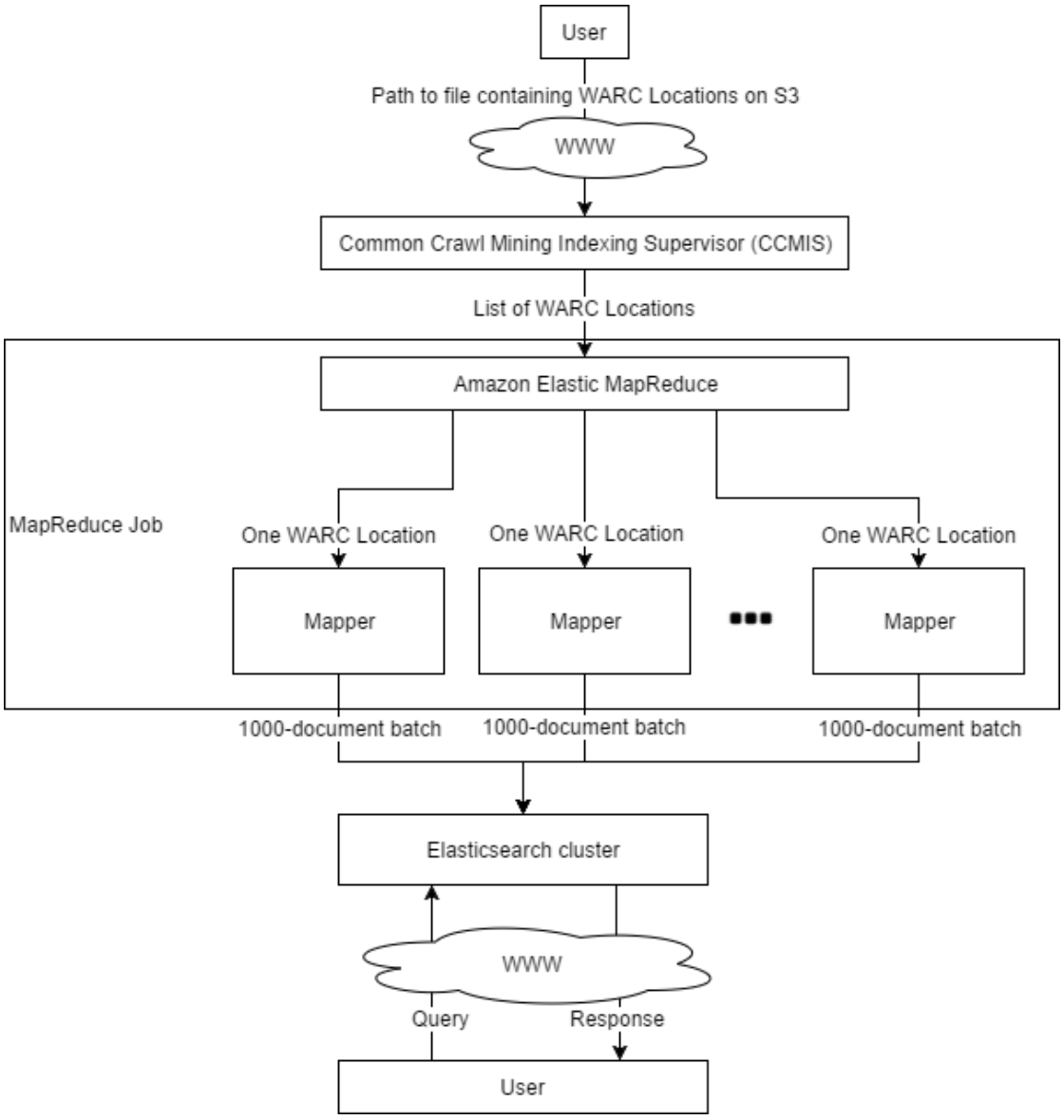


Figure 22. Data Flow Diagram

Before queries can be serviced, the system must pre-process and index Common Crawl data. In order to pre-process and index Common Crawl data, the system, specifically the Common Crawl Mining Indexing Supervisor (CCMIS), must be provided the path to a text file as input that contains the locations of WARC files on S3, with one location on each line of the file. This input will be provided to the CCMIS through a web-facing REST interface. As Figure 22 shows, the CCMIS will spawn a single-stage MapReduce job and the WARC locations will be given as the input to the mapper. Hadoop normally designates many lines of input to a single map task but we send each line to a separate mapper because processing a single WARC file is relatively expensive.

Each instance of the mapper will retrieve a WARC file from S3 using the path it was given and iterate over the records that the WARC file contains. The mapper will skip any records that aren't HTTP responses containing HTML data because we are only interested in the text content of web pages. It will also skip any records that don't have associated request URIs because URI is used to uniquely identify a page in the Common Crawl Mining system.

The mapper will then pass the page's HTML data to the Webhose article date extractor which will return the probable publication date of the page, if it can be deduced. If no publication date can be found, then the record will be skipped because undated pages are not useful for the temporal and trend analysis that this system is intended to inform. Records that have not been skipped at this point will be passed into Beautiful Soup which will return the probable body text of the record. The body text, publication date, crawl date, and URI of this record will then be passed in batches, of approximately 1000 documents, to Elasticsearch for indexing.

We have selected this batch size based on recommendations made by Elasticsearch that "a good place to start is with batches of 1,000 to 5,000 documents or, if your documents are very large, with even smaller batches. It is often useful to keep an eye on the physical size of your bulk requests. One thousand 1KB documents is very different from one thousand 1MB documents. A good bulk size to start playing with is around 5-15MB in size."[27]

For each index request in a batch, Elasticsearch will look for an existing document with the given URL; if one is found, it will be overwritten. Since indexing jobs should be run on monthly batches that chronologically proceed previously indexed batches, it can be assumed that all new matching documents are the most recent versions. Otherwise, a new document will be added to the index containing the given data. This process will happen in parallel with multiple mappers sending index requests to be processed simultaneously by multiple Elasticsearch shards.

Once the data has been stored in Elasticsearch, the user can give queries that will be searched for in Elasticsearch. These queries will be given to the web-facing interface provided by Elasticsearch in the form of standard Elasticsearch Query DSL JSON. Queries may include chemical names and other search terms. Elasticsearch's default relevance score computed by Lucene's Practical Scoring Function (which involves Boolean model, TF/IDF, and vector space model components) will be considered a satisfactory relevance metric for the web pages found. Scores that are calculated using Lucene's Practical Scoring Function will be positive values where larger values correspond to higher relevance. The output provided to the user by Elasticsearch will include the URL, publication date, crawl date, HTML text and relevance score for each web page it deems relevant.

B.4 Schema & Interface

The schema for the input to Elasticsearch will contain at a minimum the URL of the record, publication date and HTML text. This input will be provided to Elasticsearch in the JSON format expected for bulk indexing, with documents separated by newline characters.

Queries will be provided in JSON format appropriate for the standard Elasticsearch Query DSL interface. The response to a particular query will not be stored in this system but may be cached by the user. The output will contain at a minimum the URL for the record, publication date, HTML body text and relevance score. Queries will be made as GET requests to the _search endpoint and will look something like this for the search term “endocrine disruption”:

```
{
  "query": {
    "match_phrase" : {
      "html" : "endocrine disruption"
    }
  }
}
```

B.5 Complexity, Cost, and Resource Analysis

The following tables illustrate monthly rates for various AWS resources and services that we use.

Table 6: Storage Costs

Storage Type	Cost
m4.large.elasticsearch	$0.151 per Hour
General Purpose (SSD) Storage	$0.135 per GB / month

Table 7: Computation Costs

EC2 and EMR	EC2	EMR
m4.large	$0.120 per Hour	$0.030 per Hour
m4.xlarge	$0.239 per Hour	$0.060 per Hour
m4.2xlarge	$0.479 per Hour	$0.120 per Hour

Table 8: Instance types and their specs

Model	vCPU	Mem (GiB)	SSD Storage (GB)	EBS Bandwidth (Mbps)
m4.large	2	8	EBS-only	450

m4.xlarge	4	16	EBS-only	750
m4.2xlarge	8	32	EBS-only	1000

The count of web pages in the February 2017 Index from the domains provided by Eastman Chemical was 698,697. Based on 165.54MB being the indexed size of 11,031 dated HTML records that we extracted from a February 2017 WARC file, we can infer that the average indexed size of a page is something like 165.54MB / 11031 = 15KB. Therefore, the Elasticsearch storage required for just the pages on the domain list is likely to be about 698,697 pages x 15KB per record = 10.5GB. In terms of Elasticsearch storage, this would cost 10.5GB x $0.135 GB/month = $1.42/month on SSDs and 10.5GB x $0.067 = $0.70/month on magnetic storage.

If we assume that all WARCs have similar content to the one we analyzed, we can probably predict the publication date of about 25.2% of the HTML responses in them. Given that the Common Crawl February 2017 Index contains about 3.08 billion web pages and assuming we're able to predict the publication date of 25.2% of them, this works out to about 3.08 billion x 0.252 x 15KB = 11.7TB of indexed data. In terms of Elasticsearch storage, this would cost 11.7TB x $0.135 GB/month = $1,580/month on SSDs and 11.7TB x $0.067/month = $784/month on magnetic storage.

A good rule of thumb for deploying Elasticsearch is to have between 1% and 3% of disk space available in memory[32]. Given 11.7TB of indexed data, this means between 11.7TB x 0.01 = 117 GB and 11.7TB x 0.03 = 351 GB of memory. This can be achieved with a cluster of three to seven m4.4xlarge.elasticsearch instances at a cost of $1.207/hour each, or between $1.207 x 24 x 30 x 3 = $2,607 and $1.207 x 24 x 30 x 7 = $6,083 per month.

In three test trials, it took our multiprocessing-pool-based MapReduce job, using a m4.2xlarge core node, 43, 45, and 49 minutes to execute the step that processed the WARC file we pulled from the Common Crawl February 2017 Index. This time is overwhelmingly dominated by the processing performed on HTML responses in particular; the processing step only takes about 3 minutes if the HTML response processing is excluded. The WARC contained 43,748 HTML responses, so we can assume that it takes on average about 43 minutes / 43,748 responses = 59ms per response. Given that there are 3.08 billion pages in the February 2017 Index, we can compute that processing the entire month would take about 3.08 billion x 59ms per response = 50,478 hours of m4.2xlarge compute time. Given that m4.2xlarge instances cost $0.431/hour, this means it will cost about 50,478 hours x $0.431/hour = $21,756 to process the entire Common Crawl February 2017 Index.

C Implementation

C.1 Team Roles

Each team member is assigned to a main role in the project. Due to the size of the team in relation to the complexity of the project, team members often collaborate with one another within their assigned roles. Each project role has certain tasks associated with it.

Casey Jordan Butenhoff: DevOps Lead
- Maintain project on AWS
- Support project on GitHub

Brian Clarke: Scribe
- Take notes during meetings
- Proofread reports

Tommy Dean: Communications Lead
- Maintain communication with client and professor
- Lead client meetings

Ali Pasha: Data Analysis Lead
- Perform cost and resource analysis
- Formulate implementation solutions for data analysis

C.2 Project Timeline

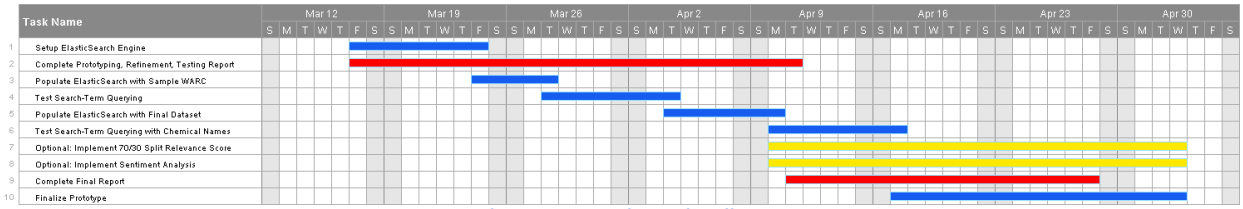


Figure 23. Project Timeline

C.2.1 Set up Elasticsearch Engine (Done)

The Elasticsearch engine was set up through AWS to support the design decisions illustrated in Figure 21 and Figure 22. The components were provisioned based on our complexity, cost, and resource analysis. See Appendix D for specific details on how the prototype was set up.

C.2.2 Complete Prototyping, Refinement, and Testing Report (In-Progress)

The Prototyping, Refinement, and Testing Report will be completed through regularly scheduled team meetings and individual contributions via the Google Docs collaboration tool.

C.2.3 Populate Elasticsearch with Sample WARC (Done)

A sample WARC file was selected based on its frequency of records from interesting domains. These interesting domains were provided by the client. A MapReduce job extracted records with publication dates from the sample WARC file and indexed them into the Elasticsearch engine. See Appendix D for specific details on the MapReduce job and interesting domains.

C.2.4 Test Search-Term Querying (Done)

Basic search term testing was conducted on the data ingested from the sample WARC. Search terms were retrieved from a sample dataset provided by the client. The retrieved search terms were queried against the sample WARC in Elasticsearch to ensure that:

1. The results did contain the search term.
2. The sample WARC contains some records of interest.

See Appendix D for specific details on the search terms. See Appendix E for more information on our testing procedures.

C.2.5 Populate Elasticsearch with Final Dataset

The final dataset will be ingested into the final prototype from a month of Common Crawl data. A parallelized MapReduce job will be used on a subset of WARC files from the month that has a high frequency of relevant domains. From this subset, it will extract those records that have publication dates.

C.2.6 Test Search-Term Querying with Search Term & Chemical Names

Additional tests on the sample WARC file will be conducted using both search terms and chemical names. This will be conceptually the same as searching for multiple search terms in that chemical names can be treated as additional search terms. The search terms will be the same as those used in previous testing with a list of chemical names provided by the client. See Appendix D for specific details on the search terms and chemical names.

C.2.7 Optional: Implement 70/30 Split Relevance Score

Although search term querying with chemical names is conceptually the same as searching for multiple search terms in that chemical names can be treated as additional search terms, we would like to implement custom scoring into the engine which will provide higher weight to search terms and and lower weight to chemical names. A solution that we are investigating is custom score queries which allow a custom scoring function to be used[33]. See Appendix F for more information on custom relevance score refinement.

C.2.8 Optional: Implement Sentiment Analysis

Sentiment analysis can be performed using classification models like Logistic Regression, Naive Bayes or Support Vector Machines. These models can be used on a TF/IDF or Word Vector representations of the data. Training data will have to be generated by hand.

C.2.9 Complete Final Report

The Final Report will be completed through regularly scheduled team meetings and individual contributions via the Google Docs collaboration tool.

C.2.10 Finalize Prototype

The team will finalize the Common Crawl Mining prototype and provide the client with test results and a demo of the system’s capabilities.

D Prototyping

We began this project by meeting with Don Sanderson and Ken Denmark, representing Eastman Chemical Company, to discuss project requirements and stretch goals. As a result of this meeting, Eastman provided us with four files from their earlier attempt to implement a system that crawls the web and analyzes public sentiment towards chemicals. These files included:

1. ChemicalListing.csv – list of commercial chemicals that were paired with search terms
    a. Each row contains a chemical's name and CAS Registry Number

```
Chemical,CAS
Fragrance(s)/perfume(s),000000-00-1
Dye(s) (unspecified),000000-00-2
Enzyme(s) (unspecified),000000-00-3
Surfactant(s) (unspecified),000000-00-4
Lauryl polyglucose,000000-00-5
Solvent extracted mineral oil,000000-00-7
MSDS: Proprietary or trade secret ingredient(s),000000-00-8
Zinc compound(s) (unspecified),000000-00-9
```

Figure 24: Sample chemical search terms

2. SearchTerms.csv – list of search terms defined by our client.
    a. The "+" was a simple "and" condition
    b. <chemical> was a placeholder to search for any chemical in the above list

```
SearchSK    SearchTerm
0   male+infertility+<chemical>
1   salon+health+<chemical>
2   endocrine+disruptor+hormone
3   perfluorinated+compound
4   asthma+<chemical>
5   acrylic+paint+health+risk
6   targeted+ban+congressional+<chemical>
7   crop+hazard+<chemical>
```

Figure 25: Sample search terms from Eastman

3. TargetURLs.csv – list of domains that are closely related to chemicals and therefore likely to contain a high proportion of chemicals in the bodies of their pages.

```
SourceSK,URL,Domain
126,http://saferchemicals.org/,Sustainability
129,http://www.ecy.wa.gov/ecyhome.html,Sustainability
134,http://www.IHS.COM,Procurement
8,http://greenandcleanmom.org/,Sustainability
9,http://greenlivingguy.com/,Sustainability
10,http://greenpeaceblogs.org/category/toxics/,Sustainability
11,http://grist.org/,Sustainability
12,http://groovygreenlivin.com/,Sustainability
13,http://inthesetimes.com/,Sustainability
14,http://mindfulmomma.com/,Sustainability
15,http://myplasticfreelife.com/,Sustainability
16,http://naturallivingmamma.com/,Sustainability
```

Figure 26: URLs that Eastman suggested to filter on

4. Matches.csv – A sample of our search results where 'Relevance' matched URLs to 'SearchTerm's.
    a. Relevance is computed as 2/3 term match and 1/3 Chemical name match.
    b. There are some false positives in here, possibly due to URL content changing, as well as some sites that are no longer active.

```
,SearchTerm,Relevance,Title,URL
1,male+infertility+<chemical>,1,COMUNICADO: Merck Launches More Than a Mother Campaign Aiming to
2,male+infertility+<chemical>,1,Merck KGaA : Launches “More than a Mother” Campaign Aiming to Red
3,male+infertility+<chemical>,1,"Perfluorooctanoic acid - Wikipedia, the free encyclopedia",http:
4,male+infertility+<chemical>,1,Publications - Health and Environment Alliance,http://www.env-hea
5,male+infertility+<chemical>,1,Publications - Health and Environment Alliance,http://www.env-hea
6,male+infertility+<chemical>,1,  Other Diseases,http://www.chemicalshealthmonitor.org/blog/other
7,male+infertility+<chemical>,1,Green Advocacy Archives - Page 2 of 16 - Groovy Green Livin,http:
8,male+infertility+<chemical>,1,foodconsumer.org - Other_Diseases,http://www.foodconsumer.org/new
9,male+infertility+<chemical>,1,HEAL Newsflash - November 2015 - Health and Environment Alliance,
```

Figure 27: Sample search results

Since each Common Crawl monthly archive includes approximately 3 billion web pages, we wanted to limit the size of this data set as much as possible so that it would be easier to work with. One thought was to include only pages from the domains listed in Matches.csv, so we wanted to estimate how many pages that would include. We loaded the list of domains into a Python script and used them to query the Common Crawl February 2017 Index with their CDX Server API. The number of pages given by the API responses were counted and we found that 698,697 pages in total were crawled on those domains. An earlier version of this script misidentified this number at 300 million because it wasn’t tokenizing the API responses correctly for counting pages.

Next, we researched the structure of the data provided by Common Crawl, since using it is an essential project requirement. This data is stored and returned in the format of WARC, WAT, and WET files. Only the WARC files contain the entire HTTP response, which we later determined to be necessary to effectively predict the publication date of web pages. We’re only interested in HTTP responses containing HTML so we can ignore other record types.

We wanted to identify a WARC file we could use for testing that was most representative of the domain list we were given, so we wrote a Python script and translated all the URLs from TargetURLs.csv into a list of wildcard domains. The script again uses the CDX Server API to identify the WARC file in the Common Crawl February 2017 Index that contains each of the pages from those domains. This let us count the total number of pages from those domains in each WARC file and identifying the WARC file that contained the most was simply a matter of getting the top few counts. Here are the top three results of that script:

```
[
(u'crawl-data/CC-MAIN-2017-09/segments/1487501174215.11/crawldiagnostics/
  CC-MAIN-20170219104614-00307-ip-10-171-10-108.ec2.internal.warc.gz', 24),
(u'crawl-data/CC-MAIN-2017-09/segments/1487501171971.82/warc/
  CC-MAIN-20170219104611-00388-ip-10-171-10-108.ec2.internal.warc.gz', 24),
(u'crawl-data/CC-MAIN-2017-09/segments/1487501172404.62/warc/
  CC-MAIN-20170219104612-00155-ip-10-171-10-108.ec2.internal.warc.gz', 21)
]
```

Figure 28: Script results.

We can see that the first file with the highest count is a diagnostic file, so we selected the second file in the list, CC-MAIN-20170219104611-00388-ip-10-171-10-107.ec2.internal.warc.gz, as the WARC we would use for testing going forward. We later discovered that, including the 24 pages from the domain list we were given, this file contains 138,257 records, 43,748 of which are HTTP responses containing HTML.

Considering that our dataset is very large and deployment on Amazon AWS is an essential project requirement, we decided to use Amazon EMR for distributed web page processing. In order to start doing MapReduce development, we created an AWS root account to handle all the costs associated with using AWS during the development phase of this project. To this account, we added an IAM user for each developer and uploaded SSH keys for them to authenticate with. Users were granted FullAccess permission to EC2, S3, ElasticMapReduce, and IAM SSH Keys. They were also granted ReadOnlyAccess to IAM. We deployed a t2.micro Amazon EC2 instance to this account for developer use that has been configured to authenticate IAM users. We have also created an S3 bucket for the project.

In order to run jobs on Amazon ElasticMapReduce, we built a Docker container that contains all the dependencies necessary for running the example scripts in the cc-mrjob GitHub repository. It was configured to run a simple Flask app to provide an interface for starting MapReduce jobs and fulfilling various debugging roles as needed. This was initially hooked up to a modification of the tag counter script provided by cc-mrjob so that we could make sure everything was configured correctly and functional. This initial version of the Docker container and all associated files were added to the feature/docker branch of the GitHub repository that was created for this project. This repository was cloned onto the t2.micro development instance where it could be launched in order to trigger MapReduce jobs.

Included in the GitHub repository is a script that deploys this Docker container in one command, which contains sufficient internal documentation for its workings to be self-evident:

```
#!/bin/bash

# http://stackoverflow.com/questions/17236796/how-to-remove-old-docker-containers

# kill any old docker instances
docker kill $(docker ps -a -q)

# delete any old docker instances
docker rm $(docker ps -a -q)

# build the docker container
docker build -t commoncrawlmining .

# run the docker container
docker run -d -p 5000:5000 commoncrawlmining

# wait 10 seconds for the container to start up
sleep 10

# show status of docker containers
docker ps --all

# show the logs from all containers in case something crashed
docker logs $(docker ps -a -q)
```

Figure 29: Docker script

The container also includes an mrjob configuration file like this:

```
    # https://pythonhosted.org/mrjob/guides/emr-opts.html
runners:
  emr:
    region: us-east-1
    # Set these or environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY
    aws_access_key_id: XXXXXXXXXXXXXXXXXXXX
    aws_secret_access_key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

    ec2_key_pair: mrjob
    ec2_key_pair_file: mrjob.pem

    subnet: subnet-f24877bb

    # https://aws.amazon.com/ec2/instance-types/
    num_core_instances: 1
    instance_type: m4.2xlarge
    master_instance_type: m4.large

    interpreter: python2.7

    upload_files:
    - test-1.warc

    bootstrap:
    - sudo yum install -y python27 python27-devel gcc-c++
    - sudo pip install 'requests>=2.13.0'
    - sudo pip install 'elasticsearch>=5.0.0,<6.0.0'
    - sudo pip install 'elasticsearch-dsl>=5.0.0,<6.0.0'
    - sudo pip install boto mrjob warc aws-requests-auth articleDateExtractor beautifulsoup4
    - sudo pip install https://github.com/commoncrawl/gzipstream/archive/master.zip
```

Figure 30: Mrjob configuration.

1. “runners” and “emr” indicate that we are configuring mrjob to run jobs on ElasticMapReduce.
2. “region” indicates that EMR clusters will be launched in the us-east-1 region, which is the same region that all our AWS resources are created in.
3. “aws_access_key_id” and “aws_secret_access_key”, with their values redacted, configure the access keys that allow mrjob to launch AWS resources. These are associated with a particular IAM or root account and should be kept secret because they can result in a very large bill if misused. They can also be revoked if it’s suspected that an unauthorized user has gained access to them.

4. “ec2_key_pair” and “ec2_key_pair_file” configure authentication for SSH connections to the master node, which provides things such as live job progress and fast error log fetching.
5. “subnet” is set to the VPC subnet that contains our t2.micro instance because the instance types we want to use require us to use VPC.
6. “num_core_instances” and “instance_type” configure the core nodes of our EMR clusters. These are the nodes that actually run the mappers and reducers, so they need to be pretty beefy. During testing, we’re only using one core node because we’re only using one WARC file and we’re not going to split a single WARC file into different mappers.
7. “master_instance_type” configures the master node of our EMR cluster. This node handles things like scheduling and resource management. It doesn’t take much power to do this so we can get away with using the cheapest node we can, m4.large.
8. “interpreter” makes sure that we’re using Python 2.7 rather than 2.6 or 3.x. The libraries and modules we’re using require Python 2.7.
9. “upload_files” includes the file containing the paths to all the WARC files that our job will process; essentially our input file. This file is not automatically copied to the EMR cluster so we must do it explicitly here.
10. “bootstrap” runs all the remaining commands we need to set up the execution environment in the cluster nodes. Our MapReduce job requires Python 2.7 as well as various dependencies so we make sure all of those are installed here.

The first mrjob script we wrote ingests the list of chemicals directly from the CSV file we were given and counts their frequency in the HTML response records contained in a WARC file. The first major issue we encountered was that special characters in chemical names seemed to be causing the reducer to crash. We never figured out exactly what was causing this issue, but it was worked around by excluding those characters from the chemical names. This code was added to the feature/chem_counter branch of the GitHub repository. The approach taken by this script was later abandoned when, at the recommendation of Dr. Fox, we decided to use Elasticsearch for the full-text search part of the system in order to avoid having to reinvent a search engine from scratch. With Elasticsearch handling full-text search, it was no longer necessary for the MapReduce job to know anything about particular chemicals, search terms, or domains.

Our first basic prototype extracted the body text of every HTML response in the WARC file it was given using Beautiful Soup and sent it to Elasticsearch for indexing. The Elasticsearch cluster was configured with four m3.medium.elasticsearch instances and an access policy that included IP addresses for search clients and IAM accounts for indexing clients. The output of this prototype in response to various queries can be seen in Figure 31, Figure 32, and Figure 33. As you can see, the HTML data included a lot of JavaScript that we’re not interested in and we didn’t have publication dates estimated for any of the pages. The timestamp included in the results the time that document was indexed into Elasticsearch which isn’t useful to users.

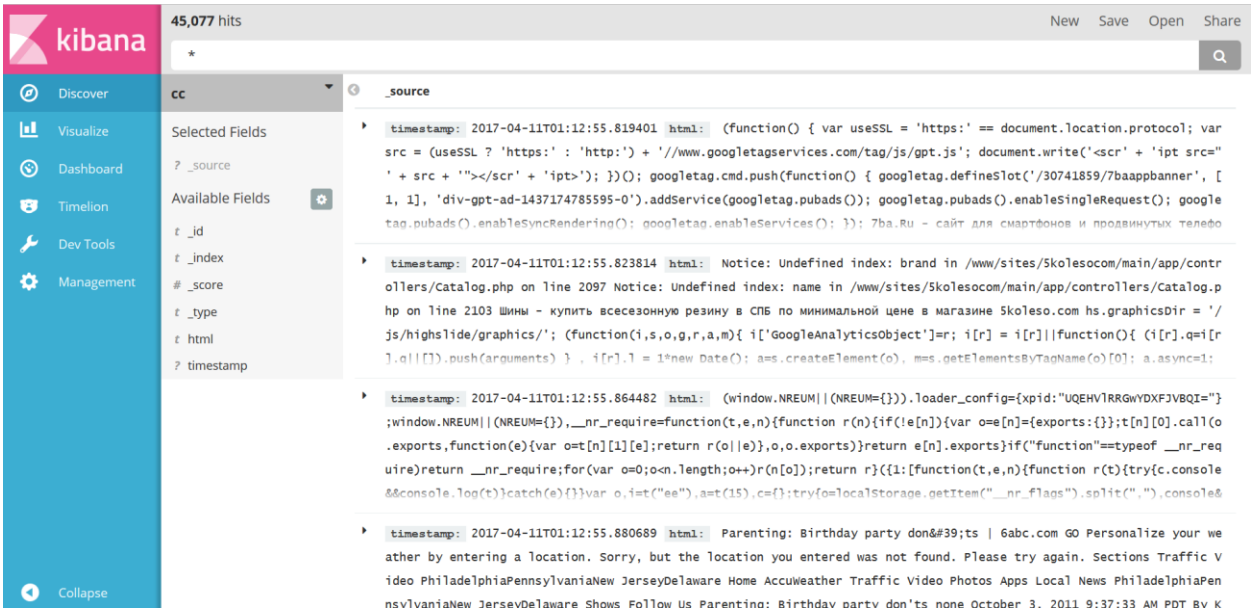


Figure 31: Kibana on early prototype with a wildcard query.

Figure 32: Kibana on early prototype with a “fertility” query.

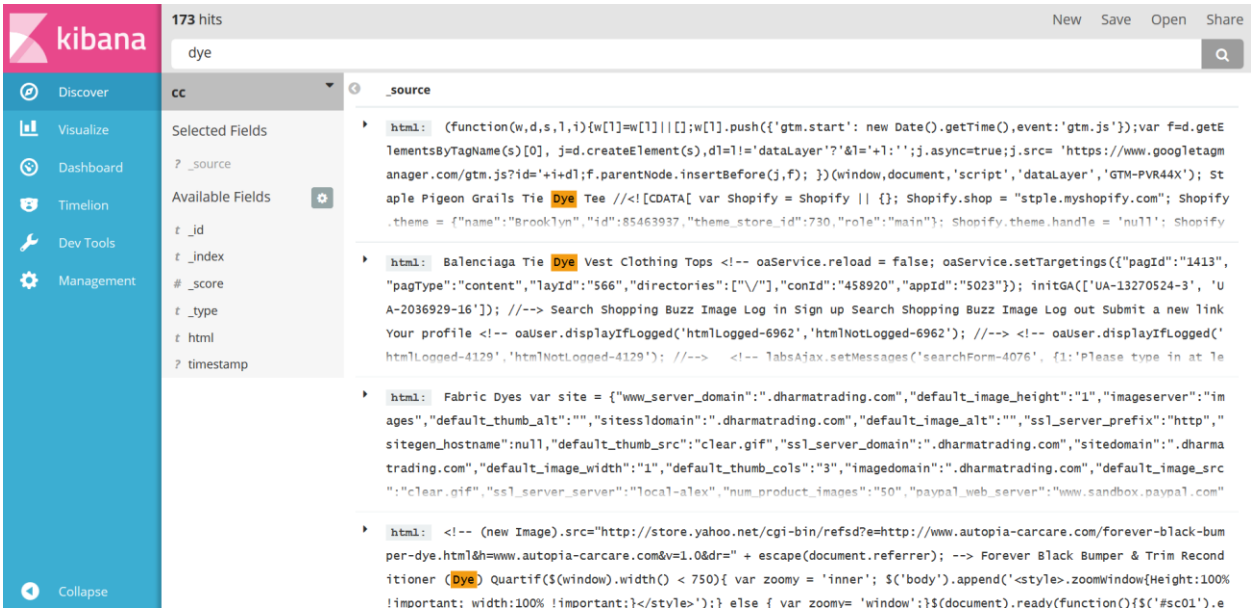


Figure 33: Kibana on early prototype with a “dye” query.

E Testing and Refinement

E.1 Tests

In order to get an idea about where our mapper was spending most of its time, we measured the time spent on extracting publication date, extracting body text, and sending the record to Elasticsearch for indexing. This test was constructed by boxing each operation in the Python code by two time.time() calls. The earlier time was then subtracted from the later time and the result was indexed into a new index in Elasticsearch. We then constructed a metric visualization on that index that averaged each of the three time fields. The result showed that the average time spent performing these operations for each record in the WARC we tested with was 0.332 seconds, 0.215 seconds, and 0.05 seconds respectively. This means extracting the publication dates represents about 56% of the work done by the multiprocessing pool, extracting body text 36%, and making indexing requests 8%. A second trial produced similar results; 0.378, 0.238, and 0.047 representing 57%, 36%, and 7%.

We used Kibana, a product of Elastic, to help us visualize our Elasticsearch results. Our initial use for Kibana was to search the returned documents for keywords. We used this as a sanity check to determine if we could find documents in the returned data set for search terms that Eastman would expect to use. Kibana offers other capabilities such as histograms, line graphs, pie charts, sunbursts, and other advanced analysis of data. As compared to Figure 31, Figure 32, and Figure 33 showing the result of the initial prototype in Kibana, Figure 34, Figure 35, and Figure 36 reflect the removal of JavaScript from the page body and the addition of both a URL and an estimated publication date for each page.

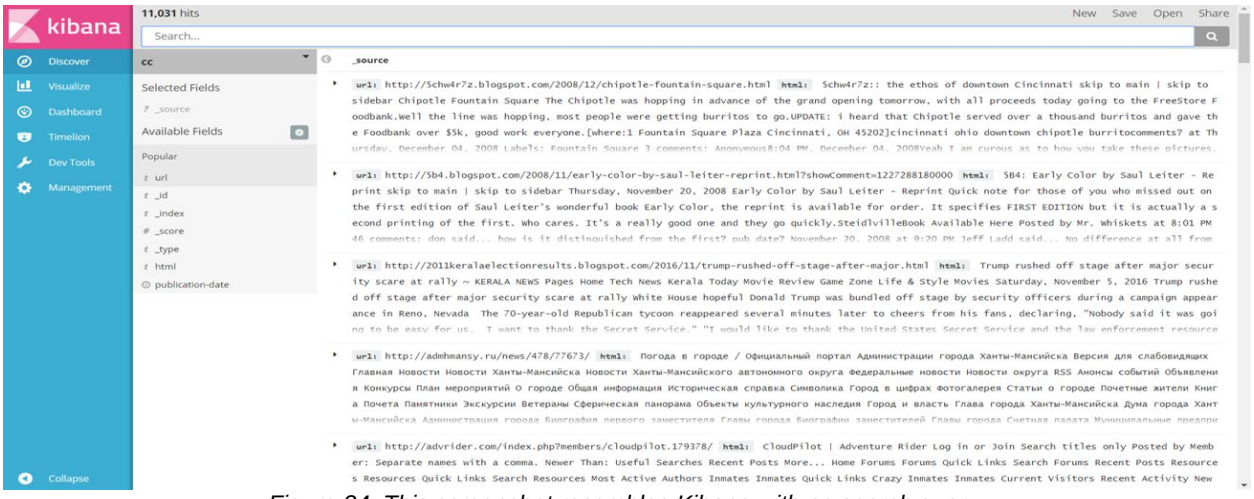


Figure 34: This screenshot resembles Kibana with no search query.

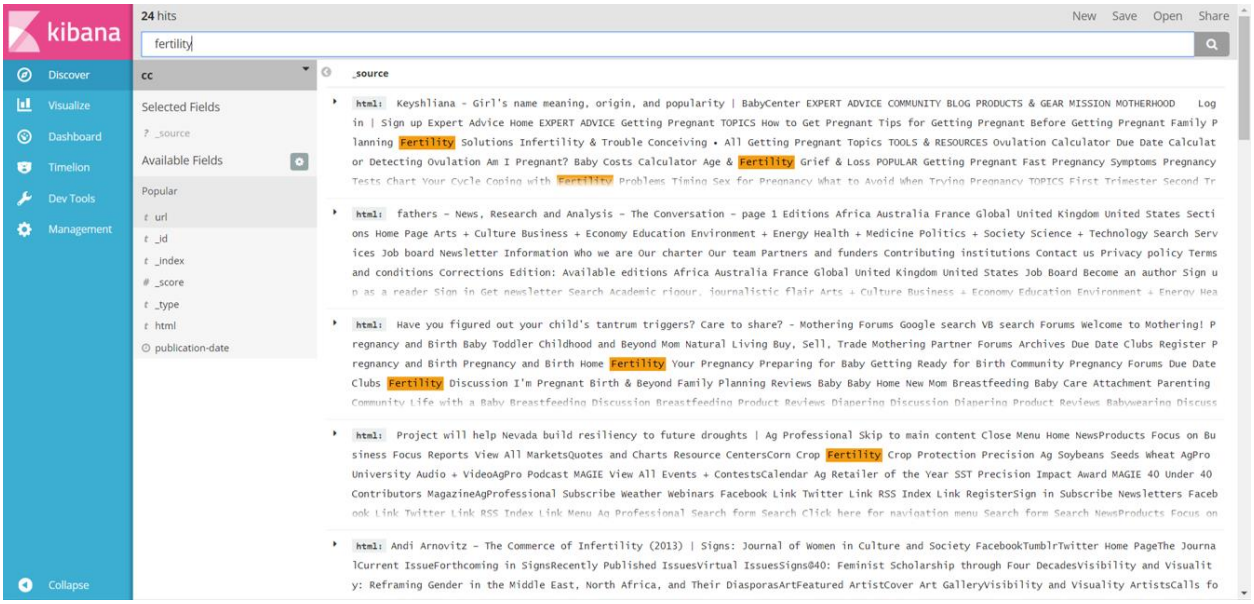


Figure 35: 'fertility' is used as a search query in this screenshot of Kibana

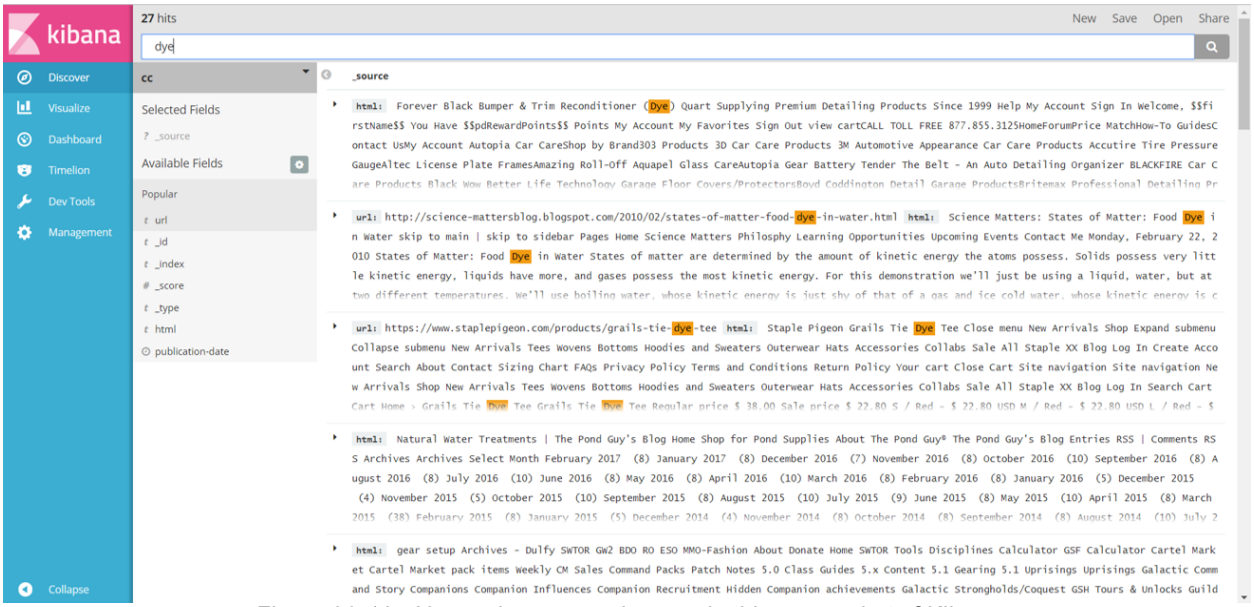


Figure 36: 'dye' is used as a search query in this screenshot of Kibana

E.2 Problems and Refinement

We discovered that the body text extractor in our first prototype included a lot of things we didn't want to index, such as script tags and JavaScript code. We solved this issue by decomposing the script and style tags from the Beautiful Soup object before extracting the body text, as seen in the following code snippet.

```
soupstart = time.time()
      soup = BeautifulSoup(body, 'html.parser')
      for script in soup(["script","style"]):
        script.decompose()
      souptext = soup.get_text()
      soupend = time.time()
      indexstart = time.time()
      # id has a limit of 512 bytes of UTF-8.
      # In this encoding, each character may require up to 4 bytes
      es.index(index='cc', doc_type='article', id=url[0:128], body={'html': souptext,
'publication-date': pd, 'url': url})
```

Figure 37: Use of Beautiful Soup in our script

Predicting publication date of web pages is another essential requirement, but no particular method was specified by the client so we had some leeway in terms of implementation. We researched several methods to predict this date, including:

- Assuming the date it was first indexed on Google was probably close to the publication date and getting it by querying the page URL, appending “&as_qdr=y15” to the query
- Assuming the date it was first archived by Common Crawl was probably close to the publication date and getting it by searching previous indices
- IBM Bluemix Natural Language Understanding API (formerly Alchemy API)
- CarbonDate web service
- Webhose.io article-date-extractor Python module

The problem with using Google for this is that Google doesn’t provide an API for it. We would have to perform a separate web search query for each page and scrape the date out of the result page. Google is almost certain to identify this sort of automated use as abusive and restrict our system’s access.

Searching previous Common Crawl indices is straightforward by using their CDX Server API, but the main problems with this method are again not only the large number of queries but also poor precision. At best, we can infer that a page was probably published sometime within the one-month window of the first index that contains it.

The Alchemy API (predecessor to the IBM Bluemix NLU API) was the solution used by the predecessor to our project for sentiment analysis but it does date extraction as well. We determined that it’s not feasible because the cost is multi-tiered based on the number of calls per month (down to a lower limit of $0.0002 USD/NLU Item after 5 million calls[29]) which precluded us from using it because it would cost hundreds of thousands of dollars to process the Common Crawl index for a single month.

CarbonDate is a research-backed method of extracting publication date from web pages. Unfortunately, like the Alchemy API, it's very expensive to use. It depends on the Bing search API[30] which is priced based on the number of calls per month (up to a limit of 10M calls costing $27,000 per month[31]).

We ultimately decided to use the article-date-extractor Python module because it promised "close to 100% precision with 90% recall"[28] on articles in the Google news feed. We tested it with the CC-MAIN-20170219104611-00388-ip-10-171-10-107.ec2.internal.warc.gz WARC file that we identified earlier. Unfortunately, whenever we tried to use the article-date-extractor module within the mapper, the reducer would crash. Lengthy debugging led us to discover that the article-date-extractor module was printing text to standard output each time it was called in the mapper. The mrjob Python library was assuming this output was the mapper's result and giving it as input to the reducer. The reducer was proceeding to crash because this input wasn't valid JSON. This problem was solved by redirecting the file descriptor of standard output to the file descriptor of standard error for the block of code that contains the article-date-extractor call. Once this reducer crash issue was resolved, the date extractor was applied to each of the 43,748 HTML response records in the test WARC and returned a date for 11,031 of them (about 25.2%).

This script led to the development of the first fully-functional prototype. It includes a MapReduce job which processes every record in each WARC it's given. It ignores records that aren't HTTP responses, don't contain HTML data, or aren't associated with a URL. The publication dates of the remaining records are predicted. For all records for whom a publication date can be inferred, body text is extracted using Beautiful Soup and the triplet of body text, publication date, and URL is given to Elasticsearch for indexing.

This prototype was extended to log exception stack traces to Elasticsearch in order to ease debugging. Amazon ElasticMapReduce saves logs to S3 but these logs are not convenient to search through, because separate logs are stored for each container that was created and it's difficult to identify which one contained the crash without looking through all of them. Having the full stack trace in Elasticsearch makes debugging issues with the Python MapReduce job much faster, though it doesn't help much with debugging other types of issues.

An AWS Elasticsearch Kibana proxy was added to the system to provide IAM authentication for Kibana web clients. The addition of this proxy means web client IPs no longer need to be whitelisted so the Elasticsearch access policy can be exclusively restricted to IAM users. This provides significant benefits for clients with dynamic IP addresses as well as modest security benefits as the Elasticsearch cluster is no longer vulnerable to misuse originating from whitelisted IPs. The proxy is provided by the aws-es-kibana npm module and we included a convenience shell script that initializes environment variables with appropriate AWS access keys for the proxy to use for IAM authentication before launching the proxy with the appropriate Elasticsearch target address. The contents of this simple proxy-launching shell script is included here, with our access keys redacted:

```
#!/bin/bash

export AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXX
export AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 38: Proxy script to run our system to open Kibana

The prototype was further improved by leveraging a multiprocessing pool to do publication date extraction, body text extraction, and Elasticsearch indexing in parallel. While implementing multiprocessing, the MapReduce containers were being killed because they were consuming more memory than they were allocated. This was fixed by increasing the maximum Java heap size and memory allocated to the map and reduce operations. The increased memory requirement of this implementation precluded us from using compute-optimized instances, so as a result we recommend the use of m4 instances. The container memory configuration was performed as seen here, which is roughly appropriate for a m4.2xlarge core instance type:

```
def steps(self):
    # https://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/
    # http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-
core/mapred-default.xml
    # http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-common/yarn-default.xml
    # memory.mb values set to maximum-allocation-mb for instance type from here:
    # http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-task-config.html
    # java.opts set to 80% of memory.mb
    return [MRStep(mapper=self.mapper, reducer=self.reducer, jobconf=
      {
        'mapreduce.map.cpu.vcores': 8,
        'mapreduce.map.java.opts': '-Xmx13107m',
        'mapreduce.map.maxattempts': 1,
        'mapreduce.map.memory.mb': 16384,
        'mapreduce.reduce.cpu.vcores': 8,
        'mapreduce.reduce.java.opts': '-Xmx13107m',
        'mapreduce.reduce.maxattempts': 1,
        'mapreduce.reduce.memory.mb': 16384,
        'mapreduce.job.maxtaskfailures.per.tracker': 1,
        'yarn.nodemanager.resource.cpu-vcores': 8
        #'yarn.nodemanager.resource.memory-mb': 24576
      })]
```

Figure 39: Memory configuration

E.3 Code Base

Note: Multiprocessing not implemented.

When we try to run a long-running Python script from within a SSH session (AWS in this case) and exit from the SSH session, if the program tries to write to STDOUT, the program will throw an IOexception and crash. In order to get around that issue, we need to make sure that any output is written to an external file. But using something like:

sys.stdout = open('file', 'w')

is not the best solution since any external modules would still try to write to the original STDOUT. To fix that, we can change the STDOUT at the file descriptor level using dup2.

This function gets the file number of either a file or a file descriptor.

```
def fileno(file_or_fd):
    fd = getattr(file_or_fd, 'fileno', lambda : file_or_fd)()
    if not isinstance(fd, int):
        raise ValueError('Expected a file (`.fileno()`) or a file descriptor'
                         )
    return fd
```

```
@contextmanager
def stdout_redirected(to=os.devnull, stdout=None):
    if stdout is None:
        stdout = sys.stdout
    stdout_fd = fileno(stdout)

  # copy stdout_fd before it is overwritten
  # NOTE: `copied` is inheritable on Windows when duplicating a standard stream

    with os.fdopen(os.dup(stdout_fd), 'wb') as copied:
        stdout.flush()  # flush library buffers that dup2 knows nothing about
        try:
            os.dup2(fileno(to), stdout_fd)  # $ exec >&to
        except ValueError: # filename
            with open(to, 'wb') as to_file:
                os.dup2(to_file.fileno(), stdout_fd)  # $ exec > to
        try:
            yield stdout  # allow code to be run with the redirected stdout
        finally:
     # restore stdout to its previous value
     # NOTE: dup2 makes stdout_fd inheritable unconditionally
            stdout.flush()
            os.dup2(copied.fileno(), stdout_fd)  # $ exec > &copied
```

Figure 40: Functions to pipe STDOUT to external file.

Moving on, the first thing that we need to do is connect to AWS Elasticsearch. We can use the AWSRequestsAuth API provided by AWS along with the AWS access key. The AWS service that we are using is Elasticsearch and so we specify that as a parameter. After setting up the authorization, we can connect to Elasticsearch.

```
es_host = \
    'search-cces-d7dw54yl5rsstmn6xuz4qlad2e.us-east-1.es.amazonaws.com'
auth = AWSRequestsAuth(aws_access_key='AWS ACCESS KEY HERE',
                       aws_secret_access_key='AWS SECRET ACCESS KEY HERE'
                       , aws_host=es_host, aws_region='us-east-1',
                       aws_service='es')
es = Elasticsearch(host=es_host, port=80,
                   connection_class=RequestsHttpConnection,
```

```
                    http_auth=auth, timeout=30)
```

Figure 41: Requesting authorization and connecting to ElasticSearch.

Our main class, ESIndexer, inherits from the base class MRJob (MRJob is the Mapreduce API that we are using). This allows us to implement the Map, Combine, and Reduce step and have MRJob take care of the execution.

```
class ESIndexer(MRJob):

    def configure_options(self):
        super(ESIndexer, self).configure_options()
        self.pass_through_option('--runner')
        self.pass_through_option('-r')
```

Figure 42: ESIndexer class.

We define this helper function that processes each record. Each record in a WARC file has three types. It can be of type warc_fields, HTTP request, or HTTP response. The exact types are:

"application/warc-fields", "application/http; msgtype=request" and "application/http; msgtype=response".

We are only interested in the HTTP response, so we read them into the payload. Every HTTP response consists of the headers (metadata) followed by two CRLFs. The new lines are followed by the body of the HTTP response so we split the payload into its components (headers and body).

We then check if the content of the HTTP response is of type *text* and if it is we proceed with processing that record. We extract the URI of the record using the *WARC-Target-URI* field, which along with the payload is used to extract the published date. The presence of the date is used as a filter, i.e, we discard any records that we were not able to find a published date for.

The next step is to clean the body of the response which contains HTML tags and Javascript code. In order to do that we make use of the decompose function provided by Beautiful Soup. The decompose function removes the script and style tags from the body and destroys it. The only thing left to do is index each record into the Elasticsearch. The function then returns a tuple of htmlrecords and 1. The 1 is used to count the total number of HTML records in the reduce step.

```
def process_record(self, record):
       if record['Content-Type'] \
```

```
            == 'application/http; msgtype=response':
            payload = record.payload.read()

            (headers, body) = payload.split('\r\n\r\n', 1)
            if 'Content-Type: text/html' in headers:
                url = record.header.get('WARC-Target-URI')
                if url is not None:
                    with stdout_redirected(to=sys.stderr):
                        pd = \
                            articleDateExtractor.extractArticlePublishedDate(url,
                                payload)
                        if pd is not None:

              # index core text of page with ES

                            soup = BeautifulSoup(body, 'html.parser')
                            for script in soup(['script', 'style']):
                                script.decompose()
                            es.index(index='cc', doc_type='article',
                                    id=None,
                                    body={'html': soup.get_text(),
                                    'publication-date': pd, 'url': url})
                    return ('htmlrecords', 1)
        return ('totalrecords', 1)
```

Figure 43: Function to process records.

Since ESIndexer inherits from MRJob, we can define the mapper, combiner, and reducer functions.

In the mapper function, the first step is to connect to the S3 bucket that contains Common Crawl data. We are using *boto*, an interface for AWS, for establishing a connection. When we run the script from the command line, we pass in the location of the WARC file that we want to analyze as an argument. The *line* parameter for the mapper function is this location. Using the *Key* function provided by *boto*, we stream the WARC file. For each record in the WARC file, we yield the output we get after processing the record.

```
def mapper(self, _, line):
        try:

      # connect to S3 anonymously

            conn = connect_s3(anon=True, host='s3.amazonaws.com')
            pds = conn.get_bucket('commoncrawl')

      # line passed to mapper contains S3 path of WARC file
```

```
            k = Key(pds, line)
            f = WARCFile(fileobj=GzipStreamFile(k))

        # TODO: n-thread/process pool

            for (i, record) in enumerate(f):
                yield self.process_record(record)

        except:
            es.index(index='exception', doc_type='article', id=None,
                     body={'datetime': datetime.now(),
                     'traceback': traceback.format_exc()})
            raise
```

Figure 44: Mapper function for MRJob.

The combiner acts as an intermediary between the mapper and the reducer. In this case, it performs the same function as the reducer, which is used by default.

```
def combiner(self, key, value):
        try:
            for key_val in self.reducer(key, value):
                yield key_val
        except:
            es.index(index='exception', doc_type='article', id=None,
                     body={'datetime': datetime.now(),
                     'traceback': traceback.format_exc()})
            raise
```

Figure 45: Combiner function for MRJob.

The reducer takes in a key (*htmlrecords or totalrecords*) and a value and yields a tuple consisting of the key and the number of times the value has been seen.

It is important to note that in this case, the only thing that reducer does in this case is count the number of HTML and total records. The indexing and processing takes place inside the mapper. Also note that the map and reduce steps are surrounded in *try-catch* clauses. This allows for any exceptions thrown to be indexed into Elasticsearch which makes it easier to log any errors.

```
def reducer(self, key, value):
        try:
            yield (key, sum(value))
        except:
            es.index(index='exception', doc_type='article', id=None,
                     body={'datetime': datetime.now(),
```

```
                    'traceback': traceback.format_exc()})
            raise
```

Figure 46: Reducer function for MRJob..

MRJob takes care of the execution and since ESIndexer inherits from it, we can just call the *run* function on that.

```
if __name__ == '__main__':
    ESIndexer.run()
```

Figure 47: Main function.

Note: Multiprocessing implemented. Functions that are the same as before are not described here.

The *process_payload* function is the same as before with the only thing added is the ability to index the processing times for cleaning the text using Beautiful Soup and storing the record in Elasticsearch. Also the arguments to this function only consist of the payload and the URL since they are going to be checked for in the *mapper* function.

```
def process_payload(args):
    try:
        (payload, url) = args
        (headers, body) = payload.split('\r\n\r\n', 1)
        datestart = time.time()
        with stdout_redirected(to=sys.stderr):
            pd = articleDateExtractor.extractArticlePublishedDate(url,
                    body)
        dateend = time.time()
        if pd is not None:
```

```
            soupstart = time.time()
            soup = BeautifulSoup(body, 'html.parser')
            for script in soup(['script', 'style']):
                script.decompose()
            souptext = soup.get_text()
            soupend = time.time()
            indexstart = time.time()
            es.index(index='cc', doc_type='article', id=None,
                     body={'html': souptext, 'publication-date': pd,
                     'url': url})
            indexend = time.time()
            es.index(index='timers', doc_type='article', id=None,
                     body={'date': dateend - datestart, 'soup': soupend
                     - soupstart, 'index': indexend - indexstart})
            return 'daterecord'
        return 'htmlrecord'
    except:
        es.index(index='exception', doc_type='article', id=None,
                 body={'datetime': datetime.now(),
                 'traceback': traceback.format_exc()})
        raise
```

Figure 48: Function to process the payload.

The biggest change that is made to the mapper is to include multiprocessing of the record processing step. We start out the same as before by connecting to the Common Crawl bucket on S3 and streaming it. We then create a process pool, which is a function included in the multiprocessing package provided with Python.

We first check if the record is an HTTP response and then split it into the header and the body. If the content was of type html text, we add it to the processing queue. We keep adding payloads to the queue until there are 128 of them. We then process them using the Pool's map function and the *process_payload* that we wrote earlier. At the end, we also process any leftovers in the queue if they are less than 127. Also, we are keeping track of the number of responses that are non-HTTP, non-HTML and non-URL.

```
def process_payload(args):
    try:
        (payload, url) = args
        (headers, body) = payload.split('\r\n\r\n', 1)
        datestart = time.time()
        with stdout_redirected(to=sys.stderr):
            pd = articleDateExtractor.extractArticlePublishedDate(url,
                    body)
        dateend = time.time()
```

```
        if pd is not None:

            soupstart = time.time()
            soup = BeautifulSoup(body, 'html.parser')
            for script in soup(['script', 'style']):
                script.decompose()
            souptext = soup.get_text()
            soupend = time.time()
            indexstart = time.time()
            es.index(index='cc', doc_type='article', id=None,
                     body={'html': souptext, 'publication-date': pd,
                     'url': url})
            indexend = time.time()
            es.index(index='timers', doc_type='article', id=None,
                     body={'date': dateend - datestart, 'soup': soupend
                     - soupstart, 'index': indexend - indexstart})
            return 'daterecord'
        return 'htmlrecord'
    except:
        es.index(index='exception', doc_type='article', id=None,
                 body={'datetime': datetime.now(),
                 'traceback': traceback.format_exc()})
        raise
```

Figure 49: Mapper function with multiprocessing implemented.

The combiner and reducer steps remain the same as before. We do change some of the system parameters like the number of cores and memory. They can be set using the *steps* method provided by MRJobs.

```
def steps(self):

    return [MRStep(mapper=self.mapper, reducer=self.reducer, jobconf={
        'mapreduce.map.cpu.vcores': 8,
        'mapreduce.map.java.opts': '-Xmx13107m',
        'mapreduce.map.maxattempts': 1,
        'mapreduce.map.memory.mb': 16384,
        'mapreduce.reduce.cpu.vcores': 8,
        'mapreduce.reduce.java.opts': '-Xmx13107m',
        'mapreduce.reduce.maxattempts': 1,
        'mapreduce.reduce.memory.mb': 16384,
        'mapreduce.job.maxtaskfailures.per.tracker': 1,
        'yarn.nodemanager.resource.cpu-vcores': 8,
```

```
        })]
```

Figure 50: Setting system parameters.

One difficulty encountered while implementing multiprocessing was that the multiprocessing pool map method, unlike the Python built-in map, only accepts a single iterable as input. We first tried to use a collection of WARC records but complex objects aren't supported by the multiprocessing serializer. This problem was solved by generating an iterable of tuples containing both the record payload and the URL from the record header, both of which are needed by the parallelized code. Those tuples were then unpacked immediately after entering the function that the multiprocessing pool applies to the iterable.

```
chunk.extend([(payload, url)])
                continue
              for key in p.map(process_payload, chunk):
```

Figure 51: Generating a list of payload tuples

```
def process_payload(args):
  try:
    (payload, url) = args
```

Figure 52: Unpacking a payload tuple

E.4 Further Testing Planned

We want to evaluate the impact that various multiprocessing parameters have on processing time. These parameters might include things like the size of the process pool and the size of the data chunk submitted to the pool. We also want to try to get CPU utilization of our core nodes up to near 100% so we can be confident that we're hitting hardware limits.

F Refinement

F.1 Planned Refinement of Prototype

In addition to the testing and subsequent refinements outlined in Appendix E, the following sections discuss plans for future refinements to the Common Crawl Mining system.

F.1.1 Custom Relevance

The entire search query for the system consists of both a search term and a chemical name. Presence of the chemical in a document will boost the relevance of the document. One way in which we can supply relevance is the use of TF/IDF to boost the relevance of such documents containing the chemical name. This would be a less complex option that would allow us to create this effect. Another option would be to create a custom relevance boost through Elasticsearch's Function Score Query[33] capabilities. The function_score function allows you to modify the score of documents that are retrieved by a query. This can be useful if, for example, a score function is computationally expensive and it is sufficient to compute the score on a filtered set of documents. To use function_score, the user has to define a query and one or more functions, that compute a new score for each document returned by the query. Here is one example:

```
GET /_search {
        "query": {
                "function_score": {
                        "query": { "match_all": {} },
                        "boost": "5",
                        "functions": [
                                {
                                        "filter": { "match": {"test": "fertility" } },
                                        "random_score": {}.
                                        "weight": 30
                                },
                                {
                                        "filter": { "match": {"test": "dye" } },
                                        "weight": 70
                                }
                        ],
                        "max_boost": 100,
                        "score_mode": "max",
                        "boost_mode": "multiply",
                        "min_score": 0
                }
        }
}
```

Figure 53: Function boost query for Elasticsearch

This function computes the relevance score of the document depending on the presences of two search terms “fertility” and “dye”. The presence of the string “dye” will boost the relevance score of the document by $\frac{70}{30}$ more than a document containing only “fertility”. Given time constraints, we will implement this functionality into our system if we have time.

F.1.2 Data Deduplication

Common Crawl performs fuzzy de-duping during the crawl process for individual monthly batches, but it is unclear if de-dupe is performed across these batches. Therefore, it may be prudent to implement deduplication logic into the indexing process to remove old copies of records from the search engine and replace them with updated versions. This, however, depends on the needs of the system. If it is desirable to perform deduplication, then it may be done by indexing on the records’ URLs. If a matching URL is found in the search engine, then a comparison of the existing values with the new values can be done on the publication date or plain text fields to decide whether to drop or update. Alternatively, the system could default to a simple replacement policy if indexed data is always guaranteed to have been crawled after the existing records in the search engine.

F.1.3 Date Extraction

Due to the high percentage of processing time dedicated to extracting publication dates, as identified in Appendix E, it would be beneficial to improve the efficiency of the extraction process. In addition to its undesirable processing time, Webhose, the current date extraction tool, is not achieving a desirable ratio of extracted records to total HTML records. During our testing, a set of 43,748 HTML response records was identified in the sample WARC file. Of that set, 11,031 records were extracted with a publication date. This is about 25.2% of the total number of HTML records in the sample. Additionally, on a smaller set of HTML records from a different WARC file, all of the records that Webhose extracted publication dates for had the date in their URL. This might indicate that Webhose’s HTML parsing is not sufficient for identifying publication dates in the body of a record. As mentioned in the previous section, the only other options for extracting articles’ publication dates (Google, CarbonDate, IBM Bluemix Natural Language Understanding API) are not feasible for this system, so our only option is to attempt to refine the Webhose package.