OutbreakSum

Automatic Summarization of Texts Relating to Disease Outbreaks

CS 4984 Computational Linguistics

Virginia Polytechnic Institute and State University
Blacksburg, Virginia

December 2014

Rich Gruss
Daniel Morgado
Nate Craun
Colin Shea-Blymyer

Table of Contents:

Executive Summary

The goal of the fall 2014 Disease Outbreak Project (OutbreakSum) was to develop software for automatically analyzing and summarizing large collections of texts pertaining to disease outbreaks. Although our code was tested on collections about specific diseases--a small one about Encephalitis and a large one about Ebola--most of our tools would work on texts about any infectious disease, where the key information relates to locations, dates, number of cases, symptoms, prognosis, and government and healthcare organization interventions. In the course of the project, we developed a code base that performs several key Natural Language Processing functions. Some of the tools that could potentially be useful for other NLG projects include:

1. A framework for developing MapReduce programs in Python that allows for local running and debugging.
2. Tools for document collection cleanup procedures such as small-file removal, duplicate-file removal (based on content hashes), sentence and paragraph tokenization, irrelevant file removal, and encoding translation.
3. Utilities to simplify and speed up Named Entity Recognition with Stanford NER by using the Java API directly.
4. Utilities to leverage the full extent of the Stanford CoreNLP library, which include tools for parsing and coreference resolution.
5. Utilities to simplify using the OpenNLP Java library for text processing. By configuring and running a single Java class, you can use OpenNLP to perform part-of-speech tagging and named entity recognition on your entire collection in minutes.

We've classified the tools available in OutbreakSum into four major modules:

1. Collection Processing
2. Local Language Processing
3. MapReduce with Apache Hadoop
4. Summarization.

The User's Manual and Developer's manual will treat each module in detail.

User's Manual

This section is meant to be used as a guide to anyone who wishes to use this suite of code in their own project. If you wish to extend this code base then please also refer to the Developer's Manual section.

Prerequisite Knowledge

To use this manual, you should have an intermediate-level understanding of Python, an introductory level knowledge of Natural Language Processing, and some familiarity with MapReduce and Hadoop. Python is a thoroughly documented programming language and learning materials can be found at www.python.org. Knowledge of Natural Language Processing, and, more specifically, of the Natural Language Tool Kit (which was used extensively) can be found at www.nltk.org. Information on Hadoop can be found at hadoop.apache.org.

Collection processing

(analyze_filesizes.py, file_cleanup.py, solr.py, sentence_collector.py, paragraph_collector.py)

File Cleanup

Since size is an important limiting factor when dealing with large text collections, we created a script for finding basic statistics on collections (**analyze_filesizes.py**). See header for usage. The scripts revealed the following distributions of file size in our collections:

	Small Collection	**Large Collection**
Topic	Encephalitis	Ebola
Number of Files	365	15,344
Mean size (bytes)	6952	8775
Min size(bytes)	1	0
25%	1579	1740
50%	5241	5148
75%	8955	9598
Max size	71429	11100598
St. Deviation	8303	91182

Table 1: File Size Distributions

It became apparent through browsing that both collections contained duplicate files and numerous extremely small files, so we created a tool for cleaning up (**file_cleanup.py**). To run the script, set the DIR and MIN_SIZE_IN_BYTES variables. The code will create a hash of every file in the directory and copy non-duplicates in a '/clean' subdirectory. The script will also create a log in the /clean directory detailing which files were removed and why.

Our collection consisted mostly of small files, so we created a set of tools for dealing with small files in Hadoop (see MapReduce module).

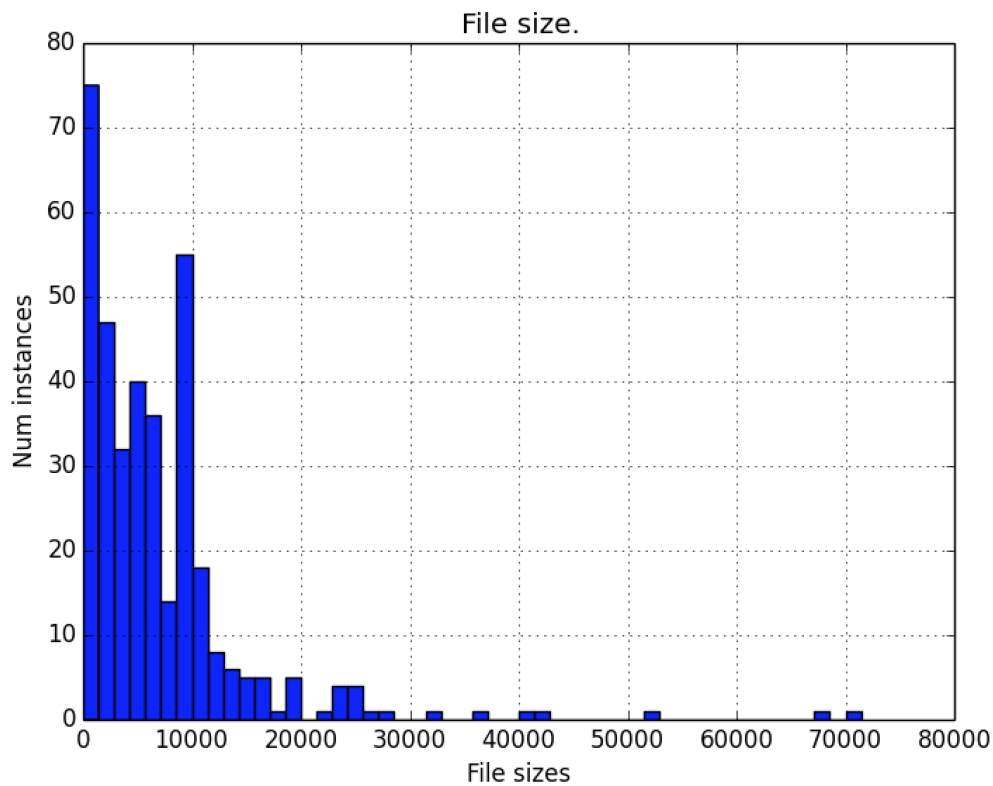

Figure 1: Histogram of File Sizes

Our collections were stored in Solr, and we periodically wanted to assess the strength of certain features for characterizing our corpus, so we created a script (**solr.py**) with a method query(q) that accepts a text query and returns a string containing precision and recall metrics. In the following sample, we're testing our named entities against our large Solr collection:

```
import solr

def getResults():

    file = open('selected_entities.txt', 'r')
    for line in file.readlines():
        q = line.strip()
        if not q.startswith('=='):
            print solr.query(q)

if __name__=='__main__':
    from timeit import Timer
    t = Timer("getResults()", "from __main__ import getResults")
    print t.timeit(1)
```

Here is some sample output from the solr.py script:

```
{'query': 'Bihar', 'relevant': 56, 'num_results': 2259, 'precision': 0.02478972996901284, 'recall': 0.007142857142857143}
{'query': 'Kenya', 'relevant': 621, 'num_results': 46794, 'precision': 0.013270932170791127, 'recall': 0.0792091836734694}
{'query': 'Chennai', 'relevant': 86, 'num_results': 1802, 'precision': 0.04772475027746948, 'recall': 0.01096938775510204}
{'query': 'Gorakhpur', 'relevant': 25, 'num_results': 65, 'precision': 0.38461538461538464, 'recall': 0.0031887755102040817}
{'query': 'Mumbai', 'relevant': 92, 'num_results': 5112, 'precision': 0.017996870109546165, 'recall': 0.011734693877551
```

Listing 1: Solr.py output

A collection that consists of thousands of small documents is easier to work with if you can Aconsolidate the text into a single large file. Our project includes two scripts for consolidation on ifferent granularities: **sentence_collector.py** and **paragraph_collector.py**. To run each, set the DIR variable to the directory containing your collection. The script creates a line for each sentence/paragraph, and gives each line an identifier so you can trace it back to the original file (filename-sequence number). The following is an excerpt of the output from running the script against our small collection.

```
1-1:The Municipal Corporation of Delhi (MCD) has suddenly woken up to the health and hygiene of pigs.
1-2:Not because Delhiites have taken a huge liking for pork, but because cases of Japanese encephalitis ( JE) have surfaced in the city.
1-3:Now one might ask what pigs have got to do with Japanese encephalitis.
1-4:The answer is that domestic pigs play host to the virus.
```

Listing 2: sentence_collector.py output

Classification of Relevant Documents

There are many sources of noise within a dataset that should be removed before processing. This includes documents in non-English languages, off topic articles, and other irrelevant documents, such as JavaScript source code. One method of removing this noise from the collection is by using machine learning to build a classifier that identifies which documents are relevant, and which ones are noise. This can be done using Scikit-Learn.

Scikit-Learn is a powerful and flexible Machine Learning software package for Python. We used Scikit-Learn as part of an effort to filter out noise from our dataset. We divided our dataset into two categories, one for noise that was irrelevant to the subject matter of diseases (mostly blank files, JavaScript sources, and completely irrelevant documents), and one for documents that were representative of the type of data that would be good for information extraction. From this training data, a Naive Bayesian Classifier was built, and used to filter out the noise from the data set.

To work on other datasets, the classifier may have to be retrained. The Naive Bayesian Classifier is a supervised learning method, which means it relies on a labeled dataset. From the dataset, take each of the documents that should be classified as noise, and list all of the filenames in a

file, like noise.txt, with one filename on each line. All of the relevant documents should have their filenames listed in a similar manner in a different file, for example relevant.txt. Modify **bayes_classify.py** and change the parameters of the calls to load_data to match your data set. By default bayes_classify.py uses a Naive Bayesian Classifier with tfidf and document length as features. To use other classifier backends or different feature extractors, simply alter the **clf** Scikit Pipeline object and add the feature extractor or classifier to the Pipeline. Details on writing custom feature extractors, and other preset feature extractors and classifiers available can be found in the Scikit-Learn documentation.

Local Language Processing

(OpenNLPRunner.java, StanfordCoreNLPRunner.java)

Operation (large collection)	ClouderaVM	DLRL Cluster	Local
POS Tagging	10m44.211s	0m51s	2m11s *
Named Entity Recognition	(killed after 4 hours)	61m31s	6m41s

Table 1: Selected Timings

It became apparent while doing Part-of-Speech tagging and Named Entity Recognition that our approach of processing each document individually on the cluster was slowing us down. For example, our POS tagger on the cluster, which collected statistics on the most frequently occurring nouns, took 51 seconds to complete. By consolidating our text and operating locally and using a Java API, in 2 minutes we were able to not only gather the same statistics, but produce a completely tagged collection that we could reuse later. Here is an example of the output from the posTag():

```
9997-65:This_DT material_NN may_MD not_RB be_VB published,_JJ broadcast,_JJ rewritten_NN or_CC redistributed._NN
9997-66:Sunday_JJ Commentary_NNP Ezekiel_NNP Emanuel:_NNP Why_WRB I_PRP want_VBP to_TO die_VB at_IN
75_CD Point_NNP Person:_NNP Former_NNP Sen._NNP Bill_NNP Bradley_NNP on_IN making_VBG America_NNP
better_RB Mitch_NNP Albom:_NNP Should_MD you_PRP be_VB able_JJ to_TO custom-order_NN a_DT baby_NN
from_IN a_DT sperm_NN bank?_NN
9998-1:Ebola_UH threatening_VBG West_NNP Africa_NNP food_NN supplies_NNS |_VBP Circa_NNP NewsEbola_NNP
threatening_VBG West_NNP Africa_NNP food_NN supplies_NNS |_JJ Circa_NNP News_NNP Save_NNP time._.
9998-2:Download_JJ Circa_NNP News_NNP for_IN free._VBG
9998-3:Get_JJ Circa_NNP News_NNP for_IN iOS_NNS or_CC Android_NNP Log_NNP in_IN Username_NNP Log_NNP
Out_IN Copyright_NNP 2014_CD Reuters_NNP Health_NNP Ebola_NNP threatening_VBG West_NNP Africa_NNP
food_NN supplies_NNS Ebola_NNP threatening_VBG West_NNP Africa_NNP food_NN supplies_NNS September_NNP
2,_CD 2014_CD 4:31AM_. Warning!_.
9998-4:Circa_NNP will_MD not_RB function_VB properly_RB without_IN javascript._VBG
9998-5:Please_. enable_VB it._.
9998-6:FOLLOW_JJ STORY_NNP OPEN_NNP STORY_NNP IN_IN APP_NNP Share_NN on:_NNS
```

Listing 3: OpenNLP Part of Speech Tagging

An even starker contrast appeared in comparing the Stanford NER with Hadoop Streaming on the cluster, which took an hour to complete, with the local version, which took only 6 minutes.

We decided therefore to build a set of tools for operating on text that has been either sentence-tokenized or paragraph-tokenized and consolidated into a single file. With the sentences.txt and paragraphs.txt files created from the scripts above, you can run several Java-based language processors to perform part-of-speech tagging, named entity recognition, parsing, coreference resolution, knowledge base population, and summarization.

Part of Speech Tagging

In OpenNLPRunner.java, set the "dir" variable to the path of your sentences.txt file and call the posTag() method. It writes a file called sentences_pos_tag.txt, with output as shown above.

Named Entity Recognition:

In OpenNLPRunner.java, set the "dir" variable to the path of your sentences.txt file and specify the type of entities you want to find in the entityTypesToFind collection.

```
private static final List<String> entityTypesToFind = Arrays.asList(
        "person", "location", "organization", "date", "money");
```

The program will produce a file for each entity type called sentences_<entity_type>.txt that simply lists each entity. To create a frequency distribution, run Counter.java, setting the dir directory and entityTypesToCount collection. This will create a file for each entity (sentences_top_<entity_type >.txt that orders the frequencies in descending order. The following listing shows the output for the sentences_top_locations.txt:

```
India  59
West Africa  34
Canada  25
Africa  18
Guinea  18
New  16
Sierra Leone  16
Asia  16
London  16
```

Listing 5: Top Location, Small Collection

Parsing and Coreference Resolution:

In trying to extract information from texts automatically, it is easy to misunderstand the intent of a statement unless you perform deep parsing. For example, the following sentence could easily trick a regex processor into thinking that there are 4000 cases of Ebola in the United States: "News in the United States of 4000 cases in West Africa has caused concern." A parse tree would remove the ambiguity about which prepositions go with which noun phrase (that is,

"news" is "in the United States", which "cases" are "in West Africa." The Stanford Core NLP library provides tools for creating parse tree like the following:

```
(NP
  (NP (NNP News))
  (PP (IN in)
    (NP
      (NP (DT the) (NNP United) (NNPS States))
      (PP (IN of)
        (NP
          (NP (CD 4000) (NNS cases))
          (PP (IN in)
            (NP (NNP West) (NNP Africa))))))))
(VP (VBZ has)
  (VP (VBN caused)
    (NP (NN concern)))
```

Listing 6: Stanford Core NLP Sentence Parsing

Likewise, ambiguity could result from pronoun usage. The following sentences, which would not present a problem to human readers, would confuse a computer: "The Centers for Disease Control has established a program for providing treatment for those with Ebola. It has been praised around the globe." The correct antecedent of "it" in the second sentence, if we follow strict grammar rules, is "Ebola," but no human reader would draw that conclusion. To correctly understand text, we need a way to intelligently tie pronouns back to their corresponding nouns. This is called "coreference resolution."

The Stanford Core Natural Language Processing library provides tools for deep parsing and coreference resolution, and OutbreakSum provides utilities for simplifying their usage.

Stanford Core NLP can downloaded at http://nlp.stanford.edu/software/corenlp.shtml#Download. Many methods in the library depend on trained models, which need to be downloaded separately from http://nlp.stanford.edu/software/corenlp.shtml#caseless. The models are large (over 280MB), so they need to be handled with care.

The class's default configuration runs several sequential annotators on the Stanford Core NLP pipeline: *tokenize, ssplit, pos, lemma, ner, parse, dcoref* . To use this runner, create an instance and invoke the analyze() method, passing the string you'd like to parse. For example this code

```java
StanfordCoreNLPRunner instance = new StanfordCoreNLPRunner();
List<String> res = instance.analyze("The spread of the deadly ebola virus throughout West Africa " +
        "is reaching epic proportions with infected people in the hardest hit areas unable to find " +
        "available hospital beds and new estimates from the World Health Organization (WHO) saying 20,000 " +
        "could be infected before it's all over. According to the WHO, the demands of the ebola outbreak in " +
        "the hardest hit country of Liberia have now "completely outstripped the government's and partners' " +
        "capacity to respond.");
for (String str : res) {
    System.out.println(str);
}
```

produces the following output, which has been abridged:

```
ORD: World
NAMED ENTITY: ORGANIZATION
WORD: Health
NAMED ENTITY: ORGANIZATION
WORD: Organization
NAMED ENTITY: ORGANIZATION
WORD: 20,000
NAMED ENTITY: NUMBER

(ROOT (S (S (NP (NP (DT The) (NN spread)) (PP (IN of) (NP (NP (DT the) (JJ deadly) (NN ebola) (NN virus))
(PP (IN throughout) (NP (NNP West) (NNP Africa)))))) (VP (VBZ is) (VP (VBG reaching) (S (NP (NP (NN
epic) (NNS proportions)) (PP (IN with) (NP (NP (JJ infected) (NNS people)) (PP (IN in) (NP (DT the) (ADJP
(RBS hardest) (VBN hit)) (NNS areas)))))) (ADJP (JJ unable) (S (VP (TO to) (VP (VB find) (NP (JJ available)
(NN hospital) (NNS beds))))))))) (CC and) (S (NP (NP (JJ new) (NNS estimates)) (PP (IN from) (NP (NP (NP
(DT the) (NNP World) (NNP Health) (NNP Organization)) (PRN (-LRB- -LRB-) (NP (WP WHO)) (-RRB- -
RRB-))) (VP (VBG saying) (NP (CD 20,000)))))) (VP (MD could) (VP (VB be) (VP (VBN infected) (SBAR (IN
before) (S (NP (PRP it)) (VP (VBZ 's) (ADVP (DT all) (IN over)))))))))) (. .)))

[reaching nsubj:[spread det:The prep_of:[virus det:the amod:deadly nn:ebola prep_throughout:[Africa nn:West]]]
aux:is xcomp:[unable nsubj:[proportions nn:epic prep_with:[people amod:infected prep_in:[areas det:the
amod:[hit advmod:hardest]]]] xcomp:[find aux:to dobj:[beds amod:available nn:hospital]]] conj_and:[infected
nsubjpass:[estimates amod:new prep_from:[Organization det:the nn:World nn:Health dep:WHO vmod:[saying
dobj:20,000]]] aux:could auxpass:be advcl:['s mark:before nsubj:it advmod:[over dep:all]]]]
```

Listing 7: Stanford Core NLP Output

The first paragraph of the output lists any named entities, and the second paragraph is the parse tree. The last paragraph is the coreference resolution. It identifies words that are semantically dependent and describes their relationship. For example, "[reaching nsubj:[spread det:The" means that the word "reaching" depends on a noun subject "spread", which is dependent on the determiner "The."

MapReduce with Apache Hadoop

Apache Hadoop is an open source implementation of the MapReduce programming paradigm. MapReduce is designed to efficiently process larger than memory datasets by exploiting parallel program design on clusters with distributed disk storage systems. We utilize the Apache Hadoop

implementation because of its large community and widespread support in both private and commercial settings.

On the most elementary level, the Hadoop MapReduce paradigm consists of a map, sort, and reduce phase. Each file in the input directory is fed to the map function which is executed in parallel across multiple nodes, which in turn produces (key, value) pairs as output. These (key, value) pairs are sorted by their key and then input to the reduce function (which may run on a single or multiple nodes) which produces output (key, value) pairs as the final output. By default, it is guaranteed that a single key is not split up between multiple reducer nodes. The input source and output destination reside in the Hadoop Distributed Filesystem (HDFS) which is a fault-tolerant, scalable, and distributed across each of the nodes.
Apache Hadoop is a Java framework and thus provides default support for Java implementation of the MapReduce functions. In order to implement MapReduce in Python we utilized Apache Hadoop's streaming interface which allows any executable file to be specified as the mapper and/or reducer. Hadoop streaming uses stdin/stdout to direct input/output to both the mapper and reducer scripts which respectively implement the desired map and reduce functions.

Hadoop Streaming with Python

Working within the bounds of the Hadoop Streaming utility we encountered various challenges and produced methodology and solutions worth noting, some of which are specific to Python.

Debugging MapReduce Functions

There are variously challenges with developing for the Hadoop ecosystem which can be limiting in most workflows, especially to newcomers.

One of the primary challenges in working with Hadoop is the developer's ability to debug MapReduce programs as it cannot be done in a traditional IDE sense as your functions are being executed on multiple computation nodes simultaneously. We quickly discovered that viewing individual node logs for errors/failures post-execution was a slow, cumbersome and inefficient way to develop MapReduce implementations. This especially applies to newcomers of Hadoop who would quickly benefit from a fail-fast development cycle.
To solve this problem we developed a Hadoop MapReduce simulation Python script which exactly replicated the distributed process of Hadoop. The script *mapred_simulate.py* can be run locally on the developer's machine such that a conventional IDE and debugger can be used. This vastly improves debugging for issues that are specific to MapReduce logical implementation details and not dependent on the particular Hadoop ecosystem or environment used.

MapReduce Framework

Working under the streaming interface can make code reuse and integration difficult as it promotes creating a new mapper and reducer file for every job.
In order to facilitate code reusability we developed a framework that allowed us to reuse MapReduce implementations across all of our jobs, both locally with *mapred_simulate.py* and on the cluster through the Hadoop streaming interface. We created a mapper and reducer script *mapred_mapper.py* and *mapred_reduce.py* respectively which can be used to call any of our MapReduce jobs defined in *mapred_computation.py*. Both *mapred_mapper.py* and

mapred_reduce.py share a utilities module named *mapred_utils.py* which provides various reading, writing, and logging functions.

Overcoming Limitations of Key, Value Pair Message Passing

Key, value pair input/output can seem limiting but there are a number of simple approaches that can be taken to enhance its potential.

One of the primary requirements of any project is the need to send rich data representations or objects in and out of your map and reduce functions. An easy way to do this in Python is to use JSON objects as your key values. This method is simple, can be read by external programs, and works best if your message can be easily encapsulated in a standard data structure. This can be extremely useful if you plan to use different programming languages for your mapper and reducer.
If your MapReduce design requires you to send more complex data representations like entire Python objects, you can use Python's serialization protocol, pickle. Pickle allows you to encode entire Python objects into ASCII representations which in turn can be used as your key values. This allows you to bypass the laborious process of manually converting your data representations into and out of a standard representation format like JSON.

Performance

Hadoop is primarily designed for accessing a small number of large files, and performs well in this scenario. When working with a large number of small files Apache Hadoop can run into performance issues.

This known issue is referred to as the Hadoop small file problem, and solutions do exist. The problem resides in the fact that a large number of files will result in many mapper tasks, each of which has their own overhead. These mappers are additionally reading in typically 64mb to 128mb blocks of input at a time. If the majority of each block read is not filled with actual data from the file, a lot of I/O and overhead operations are being performed for very little return.
A viable solution to this problem is to compress the collection of small files into larger files which can be parsed out after the file has been read in. In order to do this we created a utility script *corpus_squeeze* which will take the target directory and produce a collection of equal sized CSV files, containing one of the original files per line. It then becomes the responsibility of the mapper to take note of the new file format and execute its function on a line by line basis versus a file by file basis. Using the preceding methods we were able to achieve a minimum of x10 speedup on all MapReduce jobs.

MapReduce in Java

Annotator.java

Since Java allows more flexibility and control over distributed processing than Hadoop Streaming, we also provide a MapReduce implementation in Java. The map() method in the Annotator class runs the Standford Core NLP annotator pipeline on text passed to it by the job

controller and returns a large string that contains an extensive analysis of the text, including tokens, POS tags, named entities, parse trees, and coreference chains. The code expects paragraphs and can be run locally, but because the analysis is so slow, the following usage is highly recommended: 1) Run paragraph-collector.py to create the paragraphs.txt file. 2) Split the paragraphs.txt file into however many nodes you have in your cluster, using the **split_large_file.py** script. 3) Package your classes into a Jar file called mr.jar, including the Stanford NER Jar file and the Stanford NER models Jar. 4) import your split paragraph files into an HDFS subdirectory called "paragraphs" 5) Run the MapReduce job: *hadoop jar ./mr.jar paragraphs parseoutput*. The parsed results will be in the HDFS *parseoutput* directory.

Summarization

cost_extract.py, date_extract.py, symptom_extract.py, TemplateCreator.java, create_database.py, Summarizer.java

Template Filling

OutbreakSum includes three Python scripts and two Java classes for creating text summaries. The Python scripts and TemplateCreator.java work in tandem to fill in the slots in the following template:

```
An outbreak of 1.[DISEASE] has struck 2. [LOCATION]. As of 3. [DATE], there have been 4.[NUMBER]
number of cases either killed or hospitalized. Authorities are 5. [AUTHORITIES MEASURES]. 6. [HEALTH
ORGANIZATION] is 7. [WHAT HEALTH ORG IS DOING]. 8. [DISEASE] include 9. [SYMPTOMS].
Treatments include 10. [TREATMENTS]. The total cost of the outbreak is estimated at 11. [MONEY
AMOUNT]. The disease may spread to 12. [OTHER LOCATIONS].
```

Listing 8: Template for Slot-Summarization

Data to fill the slots is extracted from the document collection through a combination of Python scripts and Java classes, depending on the slot. To fill slots 1, 2, 4, 5, 6, 7, and 8, configure the directory containing your sentences.txt files and run TemplateCreator.java. The slots are filled using a combination of n-grams, top entities, and string filtering. The output is not meant to directly generate a template, but rather to be trimmed and combined with the output from the Python scripts:

```
DISEASE:
ebola
LOCATIONS:
west africa
canada
africa
guinea
GOVERNMENT ACTION:
HEALTH ORGANIZATION ACTION:
world health organization is concerned that the scale of the outbreak has been underestimated especially in liberia and sierra
leone, and that many people with ebola symptoms have not been identified
world health organization has declared the outbreak in west africa an international emergency, killing more than 900 people
and spreading.that's scary and serious
```

```
world health organization has estimated 2,615 people in west africa have been infected with ebola since march
world health organization has not recommended them
NUMBER OF CASES:
[0-9][0-9]* cases
an estimated 50,000 cases _ more than 10,000 of th
could exceed 20,000 cases, more than six times as
ually exceed 20,000 cases, more than six times as
```

Listing 9: Template Slot-filling Output

The remaining slots (3, 8, 10, 11, and 12) are filled with the corresponding scripts:

Sample output of symptom_extract.py:
Symptoms of [Ebola] include [high fever, fever, bleeding, vomiting, sore throat, muscle pain, diarrhoea, and headache].

Sample output of cost_extract.py:
$600,000,000

Sample output of date_extract.py
2014

Natural Language Summary Based on a Grammar

The most important elements of a summary about a disease outbreak are those that provide information about the spread and severity of the outbreak: dates, locations, and number of hospitalizations and deaths. Secondarily, a summary should provide some general information about the disease itself, such as causes, symptoms, treatments, and prognosis. The Summarizer.java class generates a natural language summary of the time-sensitive, place-sensitive, collection-specific data. The non-ephemeral data regarding Ebola can easily be obtained from a medical database, or pulled from a Wikipedia article using their MediaWiki API. Our summarizer uses the first two paragraphs of the Wikipedia article on Ebola.

To create the summary, run create_database.py to make a database of dates, locations, and cases, like the following:

```
{'date': '1989-12-02', 'source': '10041-4', 'cases': '779', 'location': 'Sierra Leone', 'deaths': '962'}
{'date': '2014-08-02', 'source': '10046-4', 'cases': 0, 'location': 'Guinea', 'deaths': 0}
{'date': '2014-08-02', 'source': '10046-4', 'cases': 0, 'location': 'Liberia', 'deaths': 0}
{'date': '2014-12-03', 'source': '10143-11', 'cases': 0, 'location': 'Guinea', 'deaths': 0}
{'date': '2014-03-02', 'source': '10152-16', 'cases': '1200', 'location': 'West Africa', 'deaths': '670'}
{'date': '2014-03-02', 'source': '10152-16', 'cases': '1200', 'location': 'Nigeria', 'deaths': '670'}
{'date': '2014-03-02', 'source': '10152-16', 'cases': '1200', 'location': 'Guinea', 'deaths': '670'}
{'date': '2014-03-02', 'source': '10152-16', 'cases': '1200', 'location': 'Liberia', 'deaths': '670'}
{'date': '2014-03-02', 'source': '10152-16', 'cases': '1200', 'location': 'Sierra Leone', 'deaths': '670'}
{'date': '2014-01-02', 'source': '10155-3', 'cases': '685', 'location': 'West Africa', 'deaths': 0}
```

Listing 10: Database for Generating Summary

This script expects locations configured for the paragraphs.txt, top_location.txt, and pargraphs_date.txt files created from OpenNLPRunner.java. Put the resulting records.txt file in the working directory of Summarizer.java and run it.

The most complicated part of the generation was selecting the data from the records. Most recent dates were obviously most important, but the most recent record usually didn't contain counts for cases or deaths. To create a reasonable best guess, we used the following approach to selecting records:

```
for each location
        sort the records in descending order of date
        select the first date
        iterate down the list until we have two numbers for "cases" and two numbers for "deaths"
        sort the two numbers and express as a range "from _ to_"
```

Listing 11: Algorithm for Choosing Number of Cases to Report

By reporting the numbers as ranges, we handled the problem conflicting numbers that littered our collection, although the results still contain some conflicting number ranges.

The code will use the following grammar to create the summary below.

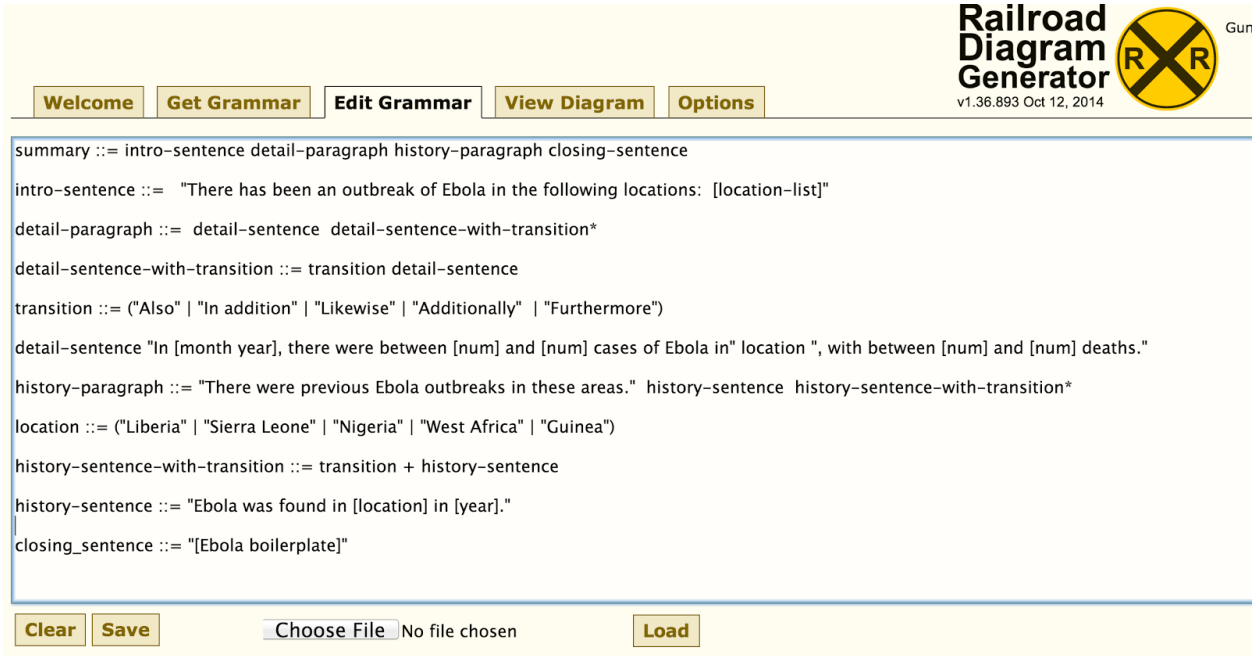

Figure 2: Grammar for Generating Summary

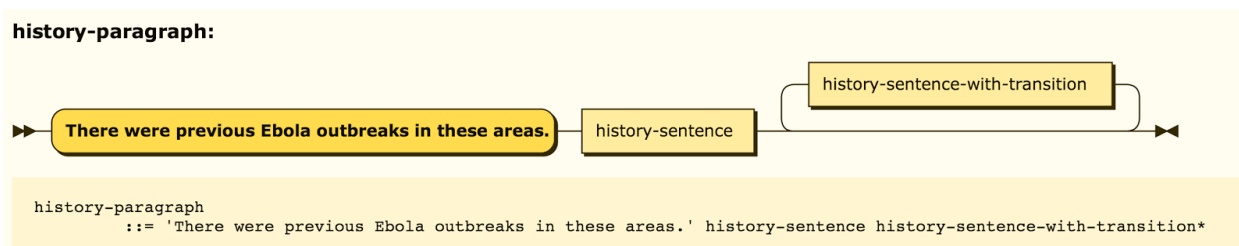

Figure 3: Grammar for Generating History Paragraph

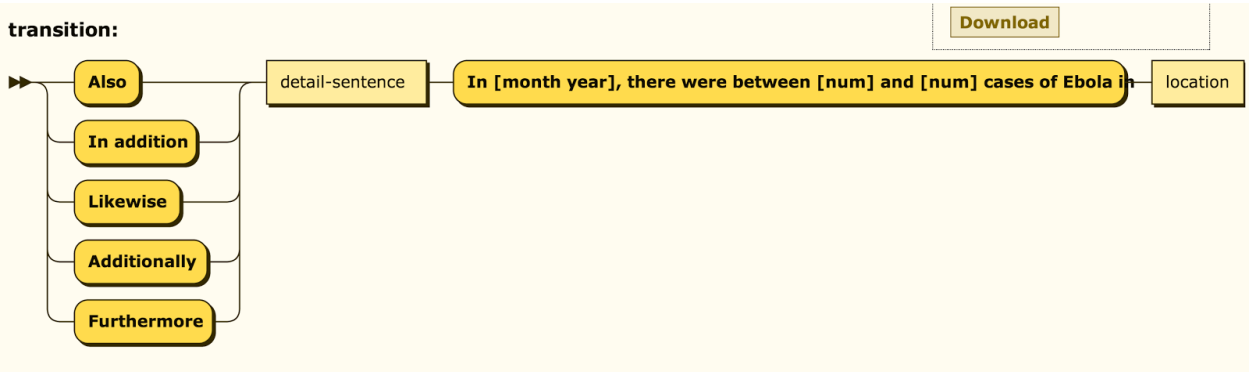

Figure 4: Grammar for Generating Sentences with Transitions

The results of the generation are as follows:

> There has been an outbreak of Ebola reported in the following locations: Liberia, West Africa, Nigeria, Guinea, and Sierra Leone.
>
> In January 2014, there were between 425 and 3052 cases of Ebola in Liberia, with between 2296 and 2917 deaths. Additionally, In January 2014, there were between 425 and 4500 cases of Ebola in West Africa, with between 2296 and 2917 deaths. Also, In January 2014, there were between 425 and 3000 cases of Ebola in Nigeria, with between 2296 and 2917 deaths. Furthermore, In January 2014, there were between 425 and 3052 cases of Ebola in Guinea, with between 2296 and 2917 deaths. In addition, In January 2014, there were between 425 and 3052 cases of Ebola in Sierra Leone, with between 2296 and 2917 deaths.
>
> There were previous Ebola outbreaks in the past. Ebola was found in 1989 in Liberia. As well, Ebola was found in 1989 in West Africa. Likewise, Ebola was found in 1989 in Nigeria. Additionally, Ebola was found in 1989 in Guinea. Also, Ebola was found in 1989 in Sierra Leone.
>
> Ebola virus disease (EVD; also Ebola hemorrhagic fever, or EHF), or simply Ebola, is a disease of humans and other primates caused by ebolaviruses. Signs and symptoms typically start between two days and three weeks after contracting the virus as a fever, sore throat, muscle pain, and headaches. Then, vomiting, diarrhea and rash usually follow, along with decreased function of the liver and kidneys. At this time some people begin to bleed both internally and externally. The disease has a high risk of death, killing between 25 percent and 90 percent of those infected with the virus, averaging out at 50 percent.[1] This is often due to low blood pressure from fluid loss, and typically follows six to sixteen days after symptoms appear.

Listing 12: Summarization of our Big Collection

Our approach presented problems when summarizing our small collection, in part because we hard-coded the choice of five locations, and the Encephalitis outbreak was really only confined to India. In the future, it would be better to stick to locations that appear more than a few standard deviations over the rest, rather than stopping at exactly five.

> There has been an outbreak of Encephalitis reported in the following locations: India, Delhi, Bangladesh, Africa, and New York.

In August 2014, there were between 56 and 500 cases of Encephalitis in India, with between 56 and 500 deaths. Additionally, in August 2014, there were between 56 and 500 cases of Encephalitis in Delhi, with between 56 and 500 deaths. Also, in August 2014, there were between 56 and 500 cases of Encephalitis in Bangladesh, with between 56 and 500 deaths. Furthermore, in August 2014, there were between 56 and 500 cases of Encephalitis in Africa, with between 56 and 500 deaths. In addition, in August 2014, there were between 56 and 500 cases of Encephalitis in New York, with between 56 and 500 deaths.

Encephalitis (from Ancient Greek ἐγκέφαλος, enképhalos "brain",[1] composed of ἐν, en, "in" and κεφαλή, kephalé, "head", and the medical suffix -itis "inflammation") is an acute inflammation of the brain. Encephalitis with meningitis is known as meningoencephalitis. Symptoms include headache, fever, confusion, drowsiness, and fatigue. More advanced and serious symptoms include seizures or convulsions, tremors, hallucinations, stroke, hemorrhaging, and memory problems.

Listing 12: Summarization of our Small Collection

Developer's Manual

Local Language Processing

The "preprocess" package includes runners for OpenNLP and Stanford Core NLP. Because OpenNLP uses a consistent programming model, each entity type can be extracted the same way: 1) read the model into a FileInputStream (the model files, which can be download from http://opennlp.sourceforge.net/models-1.5/, have a naming scheme of "en-ner-<entity>.bin."). 2) Pass the model FileInputStream to the TokenNameFinderModel() constructor, and pass this instance to the constructor of NameFinderME(). 3) Use a white space tokenizer to split your text, and pass the tokenized text to nameFinderMe.find(). This returns an array of Span objects, which can be iterated and printed. Part of Speech tagging is similar, except that you'll use the model called "models/en-pos-maxent.bin."

The StandfordCoreNLPRunner passes a Properties object to the StanfordCoreNLP() indicating the annotators to run on the supplied text.

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
pipeline = new StanfordCoreNLP(props);

// create an empty Annotation just with the given text
Annotation document = new Annotation(text);

// run all Annotators on this text
pipeline.annotate(document);
```

The central object from which you can pull part of all elements of the analysis is the Annotation. For example, to print the chain of coreferences, you can extract it from the Annotation like this:

```
    Map<Integer, CorefChain> graph =
            document.get(CorefCoreAnnotations.CorefChainAnnotation.class);

    Iterator it = graph.entrySet().iterator();

    StringBuilder corefChainStr = new StringBuilder();

    while (it.hasNext()) {
        Map.Entry pairs = (Map.Entry)it.next();
        corefChainStr.append(pairs.getValue() + ";");
        it.remove(); // avoids a ConcurrentModificationException
    }
    res.add(corefChainStr.toString());

    return res;

}
```

MapReduce Language Processing

Hadoop Streaming with Python Framework

The overarching structure of our MapReduce framework is as follows:

- *mapred_simulate.py*: Simulates MapReduce on local machine using I/O redirection.
- *mapred_computation.py*: Definition of all MapReduce jobs.
- *mapred_mapper.py*: Mapper capable of running any job in *mapred_computation.py*
- *mapred_reducer.py*: Reducer capable of running any job in *mapred_computation.py*
- *mapred_utils.py*: Utilities shared between the mapper and reducer.

MapReduce Simulation

The *mapred_simulate.py* script will directly simulate what happens in a Hadoop MapReduce job, just locally. This allows you to use your IDE of choice to locally debug your MapReduce job. Being able to debug your job logic separate from the server environment makes workflow much more efficient as it allows you to fail fast. The alternative is to run your job on a Hadoop cluster for testing which requires downloading each node's log files from HDFS to diagnose potential issues.

The simulation script will replicate Hadoop behavior using I/O redirection and intermediate files. The main routine in *mapred_mapper.py* containing your map function will be called on your specified input directory and the output (key, value) pairs will be stored in an output file. The script will then sort your keys and output the sorted order in another intermediate output file. Finally, the main routine in *mapred_reducer.py* will be called on the sorted (key, value) pair file and the output will be saved to another file. All output files are saved as to allow errors to be caught more easily, sometimes even without using the debugger.

Writing MapReduce Jobs

Writing a MapReduce job requires you to define a job within the compose, compute, combine paradigm used in *mapred_computation.py*. The "compose" function prepares the input source *file_obj* for the MapReduce job, typically an operation associated with the mapper or map phase. The amount of computation performed at the "compose" phase is completely up to the implementer. In many cases it's entirely feasible to perform all of the work in the "compose" function.

The "combine" function can be optionally defined to merge compose objects, the details of which are entirely up to the implementer. Normally, under the Java interface of the Hadoop

framework you can implement a combiner class the same as you would a mapper and reducer. The "combine" function adds this capability to the Hadoop streaming interface, which does not allow for an explicit combiner definition. The location and nature of combining is irrelevant, only the action of combining an arbitrary collection of compose objects (*compose_collection*) need be defined. For example, using the same combine function, a user can combine either all keys of the same id or just all keys output by all mappers.

The "compute" function is the step needed to be performed on compose object outputs to obtain the final output. It's possible that a "compute" function is not needed because of the work performed in the mapper. Regardless, the "compute" function will take a *compose_obj* and generate some result. The example below illustrates a definition of a MapReduce job which calculates the average letters per word using the compose, combine, compute paradigm.

```
def compose_average_letters_per_word(file_obj):
    total_words = total_char_len = 0
    for line in mapred_utils.line_reader(file_obj):
        tokens = [token for token in nltk.tokenize.word_tokenize(line) if token.isalpha()]
        total_words += len(tokens)
        for t in tokens:
            total_char_len += len(t)
    return {'total_char_len': total_char_len, 'total_words': total_words}


def combine_average_letters_per_word(compose_collection):
    cumulative_compose = {'total_char_len': 0, 'total_words': 0}
    for current_compose in compose_collection:
        cumulative_compose['total_char_len'] += current_compose['total_char_len']
        cumulative_compose['total_words'] += current_compose['total_words']
    return cumulative_compose


def compute_average_letters_per_word(compose_obj):
    total_char_len = compose_obj['total_char_len']
    total_words = compose_obj['total_words']
    if total_words != 0:
        return total_char_len / (total_words * 1.0)
    else:
        return -1
```

Mapper and Reducer

Provided you have implemented a MapReduce job following the compose, combine, compute paradigm it is very simple to port your job into your mapper and reducer files. For the mapper simply define a map function with the following signature and place it in the main routine for the mapper.

```python
def map_average_letters_per_word(input_file, output_file, separator='\t'):
    """
    Mapper for computing average letters per word.
    :param input_file: File to be mapped.
    :param output_file: The (key, value) pair output destination.
    :param separator: String used to split (key, value) pairs.
    :return: None
    """
    composition = mapred_computation.compose_average_letters_per_word(input_file)
    mapred_utils.write_key_value_pair(mapred_utils.get_file_name_for_key(input_file), separator,
                                      json.dumps(composition), output_file)
```

For your reducer define a reduce function with the following signature. Helper functions like *reduce_json* and *reduce_pickle* are predefined to make implementation even easier. Be sure to place a call to your reduce function in the main routine of your reducer file.

```python
def reduce_average_letters_per_word(input_file, output_file, separator='\t'):
    reduce_json(mapred_computation.combine_average_letters_per_word,
                mapred_computation.compute_average_letters_per_word,
                input_file, output_file, separator)
```

```python
def reduce_json(combine_function, compute_function, input_file, output_file, separator='\t'):
    """
    Generic reducer function for outputting json objects.
    :param combine_function: Combine function.
    :param compute_function: Compute function.
    :param input_file: The (key, value) pair input source.
    :param output_file: The (key, value) pair output destination.
    :param separator: Separator for (key, value) pairs.
    :return: None
    """
    kv_generator = mapred_utils.kv_pair_reader(input_file, separator)
    compose_collection = [json.loads(kv_pair[1]) for kv_pair in kv_generator]
    cumulative_compose = combine_function(compose_collection)
    mapred_utils.write_value(json.dumps(compute_function(cumulative_compose)), output_file)
```

Summarization

Template Filling

The Java codebase is divided into four packages:

- extract: Tools for extracting data from strings
- mapreduce: Hadoop implementations of Java language processing
- preprocess: Tools for extracting and creating frequency distributions for named entities and n-grams

- summary: Tools for generating template-based and NLG summaries.

The TemplateCreator class instantiates a sequence of Extractors and calls extractor.getBestAnswer(max_answers) on each in succession. The TemplateCreator can optionally call extractor.refine(str) to filter any answers on an arbitrary string.

```
public static void main(String[] args) {
    TemplateCreator instance = new TemplateCreator();
    instance.run();
}
private void run() {
    out.println("DISEASE:");
    Extractor e = new DiseaseExtractor(dir);
    printResult(e, 1);

    String disease = e.getBestAnswers(1).get(0);

    out.println("LOCATIONS:");
    e = new LocationExtractor(dir);
    e.refine(disease + " in ");
    printResult(e, 4);

    out.println("GOVERNMENT ACTION: ");
    e = new GovernmentActionExtractor(dir);
    printResult(e, 10);

    out.println("HEALTH ORGANIZATION ACTION:");
    e = new HealthOrgActionExtractor(dir);
    printResult(e, 10);

    out.println("NUMBER OF CASES:");
    e = new CasesExtractor(dir);
    printResult(e, 4);

}
```

```
public DiseaseExtractor(String dir) {
    List<String> candidateDiseases = new TopNBigramFilterStringStrategy(dir, "outbreak").getFilterStrings();
    filterStrings.add(candidateDiseases.get(0));
}
```

Each extractor pulls information from the sentences.txt file according to an instance of a FilterString Strategy. Current FilterStringStrategy implementations include the following:

- *TopNEntityFilterStringStrategy*: Filters sentences based on the top n most frequently occurring entities. For example, an instance created with new TopNEntityFilterStringStrategy("person", 5) will return all sentences contain a reference to the top 5 most frequently-occurring person names.

- *TopNBigramFilterStringStrategy:* Filters sentences based on the top n most frequently occurring bigrams. For example, an instance created with new TopNBigramFilterStringStrategy(4) will return all sentences contain a reference to the top 4 most frequently-occurring bigrams.

- *FileLookupFilterStringStrategy*. Filters sentences based on the contents of the supplied file. For example, we used this to find sentences that contained any health care organization, the names of which we had listed in a file.

See the JavaDoc for more information about implementations.

edu.vt.gruss.nlp.extract

Interface FilterStringStrategy

All Known Implementing Classes:

FileLookupFilterStringStrategy, TopNBigramFilterStringStrategy, TopNEntityFilterStringStrategy

---

`public interface FilterStringStrategy`

Created by rgruss on 11/15/14.

Method Summary

Methods

Modifier and Type	Method and Description
`java.util.List<java.lang.String>`	`getFilterStrings()`

The extract package includes two helper classes, SentenceFinder and RegexSentenceFinder that can be used from the FilterString implementation to return sentences matching a given string pattern.

Natural Language Summary Based on a Grammar

The NLG summarizer is a recursive descent generator where each element of the grammar implements the Node interface.

```
public abstract class Node {

    protected List<Node> children = new ArrayList<~>();

    public void emit() {
        for (Node node : children) {
            node.emit();
        }

    }
}
```

The top level is the Summarizer node, and it consists only of children corresponding to the paragraphs of the summary.

```
public class SummaryNode extends Node {

    public SummaryNode(List<Record> records) {
        this.children.add(new IntroSentenceNode(records));
        this.children.add(new DetailParagraphNode(records));
        this.children.add(new HistoryParagraphNode(records));
        this.children.add(new ClosingSentenceNode());
    }
}
```

Each ParagraphNode has SentenceNodes as its childrens. The DetailParagraphNode, for example, is responsible for parsing through the database (records.txt) and creating however many sentences are necessary to cover the reported cases in all the locations of interest.

Lessons Learned

1. Documents from the web are rarely 100% relevant. Because web pages from news sites tend to be digests of a variety of topics, documents should be tokenized into paragraphs, and relevance should be calculated on the paragraph level.
2. Without deep parsing, data extraction is guess work. Extracting numbers, names, and dates means little if you can't determine the exact semantic function of those entities.
3. Template filling is difficult. Summarization methods that involve topic modeling, sentence selection, and redundancy minimization probably work best.
4. Sentence tokenization is not as trivial as it appears. Due to the ambiguity inherent in the use of the '.' symbol in English, different tokenizers divide text different ways.
5. Sentence clustering does not yield good results because there is too little text in single sentences to provide meaningful distance measures.
6. When your collection is large enough, your system can do well even with extremely poor recall. Not every sentence describing disease transmission included "Ebola is transmitted by," but enough did that we were able to extract some decent data on transmission methods.
7. Machine Learning can provide some powerful tools for identifying important features in classifying texts. For example, an unexpectedly strong differentiating feature in classifying texts as not relevant to disease was the lack of the word "children."
8. Of all the techniques in our toolkit, Named Entity recognition yielded some of the most useful results. Extraction of entities like Dates, Locations, People, and Organizations was very informative in producing the final summaries. After finding the locations of the entities within a sentence, we were able to use that information to provide greater context on the meaning of that sentence, or paragraph. A dollar amount on its own is not very informative, but when that dollar amount is found in a sentence with location and date entities, and that sentence matches a regex indicating "cost" it becomes far more useful.
9. An incredible amount of language processing can be performed succinctly and quickly on the Unix command line.
10. Regular expressions are the foundation of many complex language systems.

Acknowledgements

Special thanks to Edward A. Fox, Xuan Zhang, Tarek Kanan, and Mohamed Magdy Gharib Farag

Sponsors

- NSF DUE-1141209
- IIS-1319578

References

"Apache Hadoop 2.6.0 - Hadoop Streaming." *Apache Hadoop 2.6.0 - Hadoop Streaming*. Web. 12 Dec. 2014. <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/HadoopStreaming.html>.

"General." *Apache OpenNLP*. Web. 12 Dec. 2014. <https://opennlp.apache.org/>.

"Railroad Diagram Generator." *Railroad Diagram Generator*. Web. 12 Dec. 2014. <http://www.bottlecaps.de/rr/ui>.

"The Small Files Problem." *Cloudera Developer Blog*. Web. 12 Dec. 2014. <http://blog.cloudera.com/blog/2009/02/the-small-files-problem/>.

"The Stanford NLP (Natural Language Processing) Group." *The Stanford NLP (Natural Language Processing) Group*. Web. 12 Dec. 2014. <http://nlp.stanford.edu/software/corenlp.shtml>.