Project Report

Tracking FEMA Project

CS4624 Spring 2015 - Dr. Edward A. Fox

Virginia Tech - Blacksburg, VA

April 28, 2015

Client: Seungwon Yang (seungwonyang@gmail.com)

Team:

- Tyler Leskanic (tyler47@vt.edu): Web Development and Hosting
- Kevin Kays (kkays@vt.edu): Data Processing
- Emily Maier (emilymaier@mykolab.com): Data Retrieval
- Seth Cannon (grim2103@vt.edu): Data Visualization

Table of Contents

Executive Summary

The TrackingFEMA project was started to deliver a finished product that would be a website visualizing the efforts of disaster response organizations (such as FEMA). The visualizations will be driven by a Javascript based library used to display various aspects of a disaster. The visualizations would be managed and setup from within a CMS. The data used in the visualizations will be parsed from HTML in Virginia Tech's Integrated Digital Event Archiving and Library (IDEAL). The current state of the project, are working processing scripts that extract information from IDEAL and then process them into set fields. Those fields (data) then manually have to be converted into the proper (intended) visualization and entered in the CMS. The future hope for this project is that, the whole process outlined above could be automated.

Part I: Application Overview

Project Description

The finished product will be a website visualizing the efforts of disaster response organizations (such as FEMA). The visualizations will be driven by a Javascript based library used to display various aspects of a disaster. The data used in the visualizations will be parsed from HTML in Virginia Tech's Integrated Digital Event Archiving and Library (IDEAL).

Objectives

The finished product should be able to:

- Depict the locations of relief efforts over time
- Depict the involvement of people within the agency over time
- Depict the types of relief efforts over time.
- Provides well-documented scripts for the retrieval and processing of the data.

Users

A 'User' for the product can be:

- Someone interested in seeing the efforts of an agency through a specific disaster.
- Someone working to expand the project, and track the efforts of the agency or other agencies over other disasters.

Part II: Functional Requirements

Scope

The project can be divided into four components:

The Data Parsing component

The Data Parsing component of the project works with Virginia Tech's Integrated Digital Event Archiving and Library (IDEAL). The archive includes news articles and Tweets about various disasters. This part of the project will use scripts to extract relevant information from the archive and convert it into a set of text files. The output of this part of the project will not be directly visible to the end user. Instead, the output will be used by the Data Processing component.

This component will use the Python programming language along with the BeautifulSoup library. BeautifulSoup allows information to easily be parsed from HTML files. Since the archive new articles are very heterogeneous, the library is extremely useful for grabbing the relevant information.

Figure 1: The main IDEAL webpage to be parsed.

The Data Processing component

The Data Processing component, which takes plain English text from the Parsing component as its input and produces meaningful data structures that relate significant information about the text.
Some key points concerning the processing component:

- The data produced will include:
    - Locations (country, state, city, etc)
    - Agency Names (FEMA, Red Cross, National Guard)
    - Names of People
    - Statistics (damages figures, relief costs, fatalities)
    - Types of relief (food, shelter, money)
    - Timestamps (Day/time the data was reported)
    - Phase of Disaster Management (Response, Recovery, Mitigation, Preparedness)
- This deliverable of this component will be a set of Python scripts
    - The scripts that do the processing will be well documented, so that they can be expanded and applied to other agencies and systems.
    - The scripts will be accompanied by an outline, and a detailed how-to doc for replicating its results.
    - The scripts will use the Natural Language Toolkit to aid in the processing of the text.

Website/Content Management System

Website/Content Management System, will be hosted in the cloud running on a Content Management System called RefineryCMS which runs on Ruby on Rails.

We will be providing two hosting environments:
1. The first hosting environment will be a local development environment which run inside a Virtualbox Virtual Machine running Ubuntu 14.04.1 LTS that has been preconfigured to run the site. The client can obtain a set up version of this VM at the end of the project if they so desired as all software used is Open Source.
2. The second hosting environment provided throughout the development of this project will be the staging environment. The staging environment will used to demo the site to the client as well as the class. It can also be used for visual and functional Quality Assurance. It will be hosted on a free Heroku slice and, should the bandwidth exceed what a free slice allows, other free options will be sought.

Note: Once the development process is complete (i.e. contract is over - end of semester) it will be the client's responsibility to migrate the site to their own hosting solution. All support for hosting expires after project completion.

Software Facts

The following software tools will be used for the Data Visualization component:

RefineryCMS - Version 2.1.5 - http://refinerycms.com/
Rails - Version 4.1 - http://rubyonrails.org/
Ruby - Version 2.0.0 - https://www.ruby-lang.org/en/
Bootstrap - Version 3.3.4 - http://getbootstrap.com/

The site will also be using Twitter's Bootstrap CSS Framework that will provide the base CSS and Javascript functionality. It will ultimately be tweaked to give the site its' own custom look. Since the site will use Bootstrap it will be also be out of the box responsive for support on Desktop and Mobile devices. Tablet support can be expected but is not guaranteed.

Usability and UX Basics

The CMS will give the content owner the ability to modify the pages of the website and change the content without involving a developer. This will make maintaining the site easy.

One of the key features will be the ability to add events to the site. This will be handled through the CMS and will be as simple as providing the basic information about the event/disaster which will go in the "Fast Facts" area of the event page. Then you will also provide the data for any visualization necessary for that event.

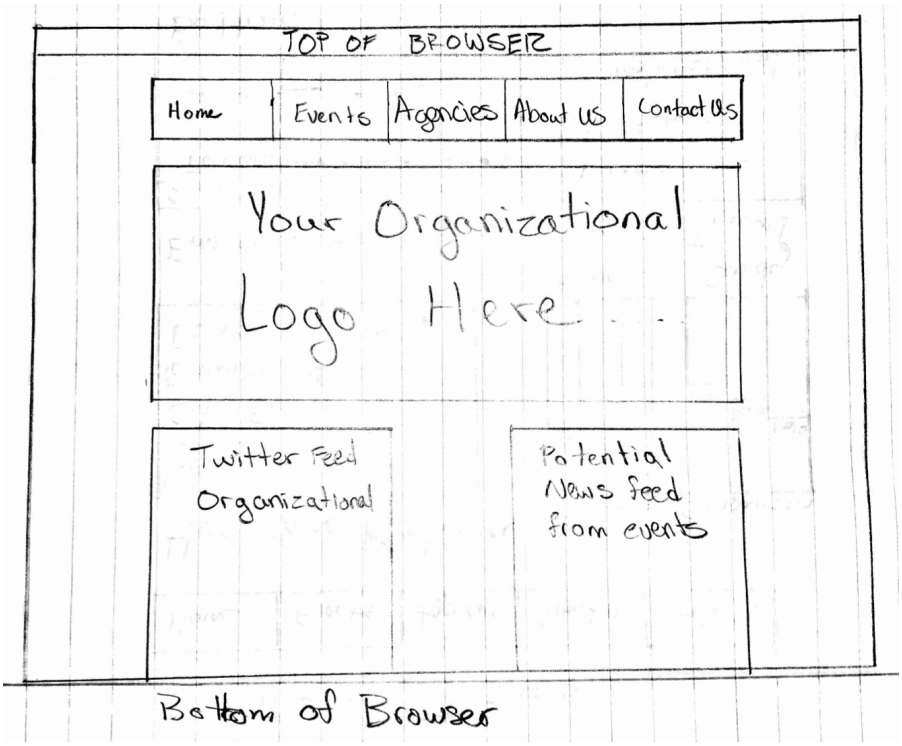


Figure 2: Front Page of CMS

Figure 1 is a rough sketch of what the front page of the CMS will look like. It will have a banner image with a logo. It will also contain two areas, each of which has the potential to handle news streams. The navigation bar will be center aligned and collapse to a mobile version with a hamburger symbol (☰ symbol) to expand.

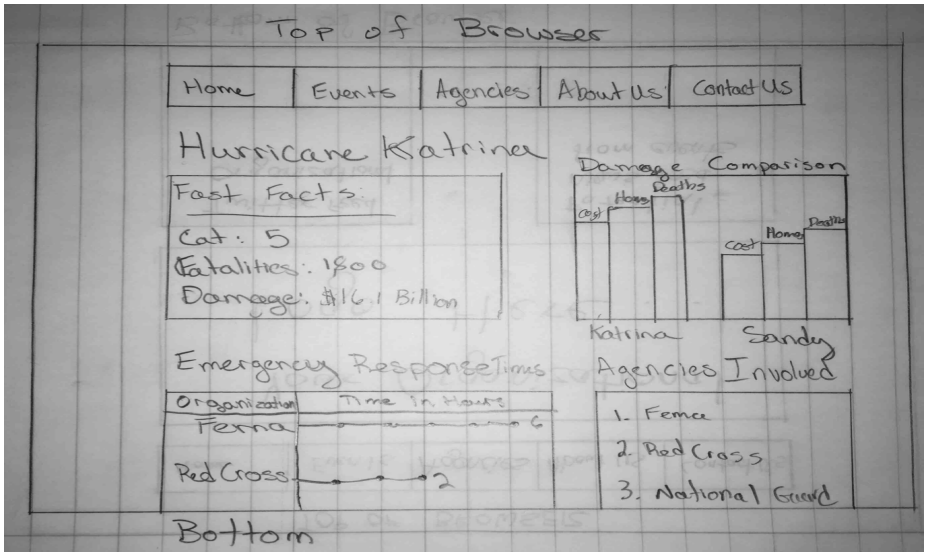


Figure 3: An Example Event Page

Figure 2 is an example of what an event page will possibly look like. It contains fast facts about the event as well as all the visualizations the event may contain from data. The visualizations presented on the site will contain data such as people involved in emergency responses, and other pertinent data as requested by the client for the emergency event.

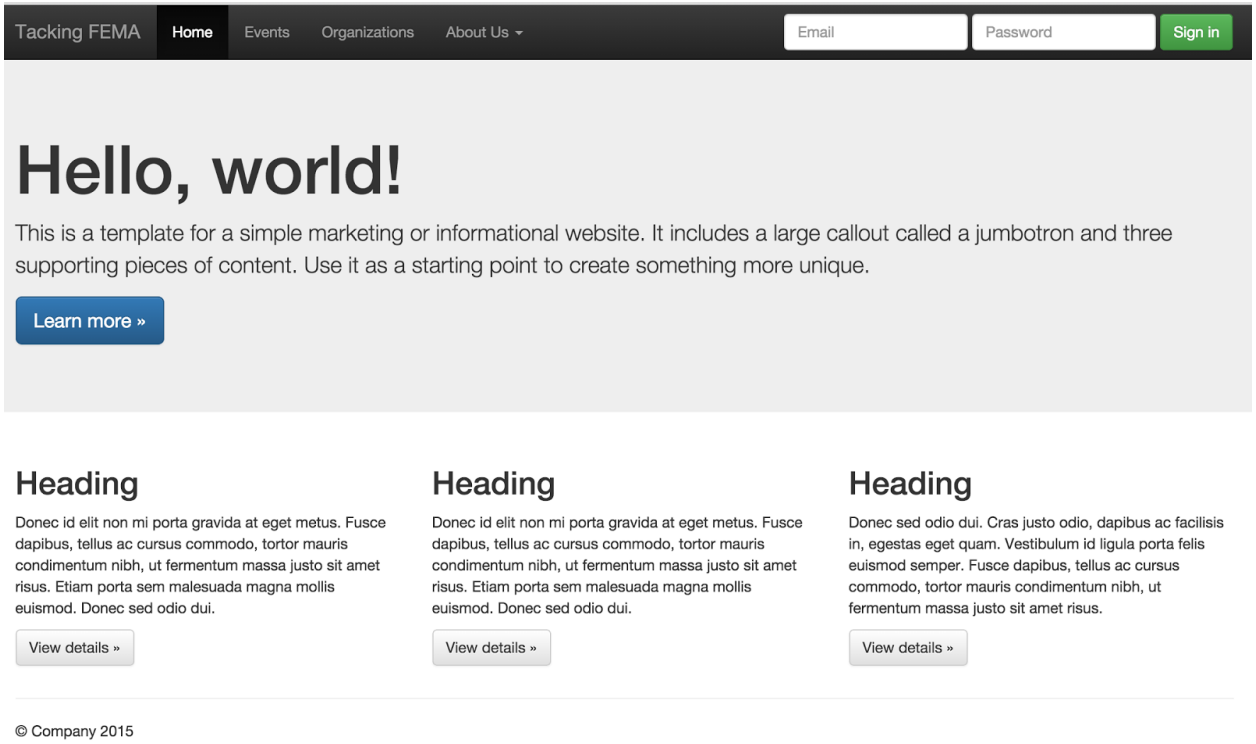


Figure 4: Homepage HTML Prototype (Dummy Content)

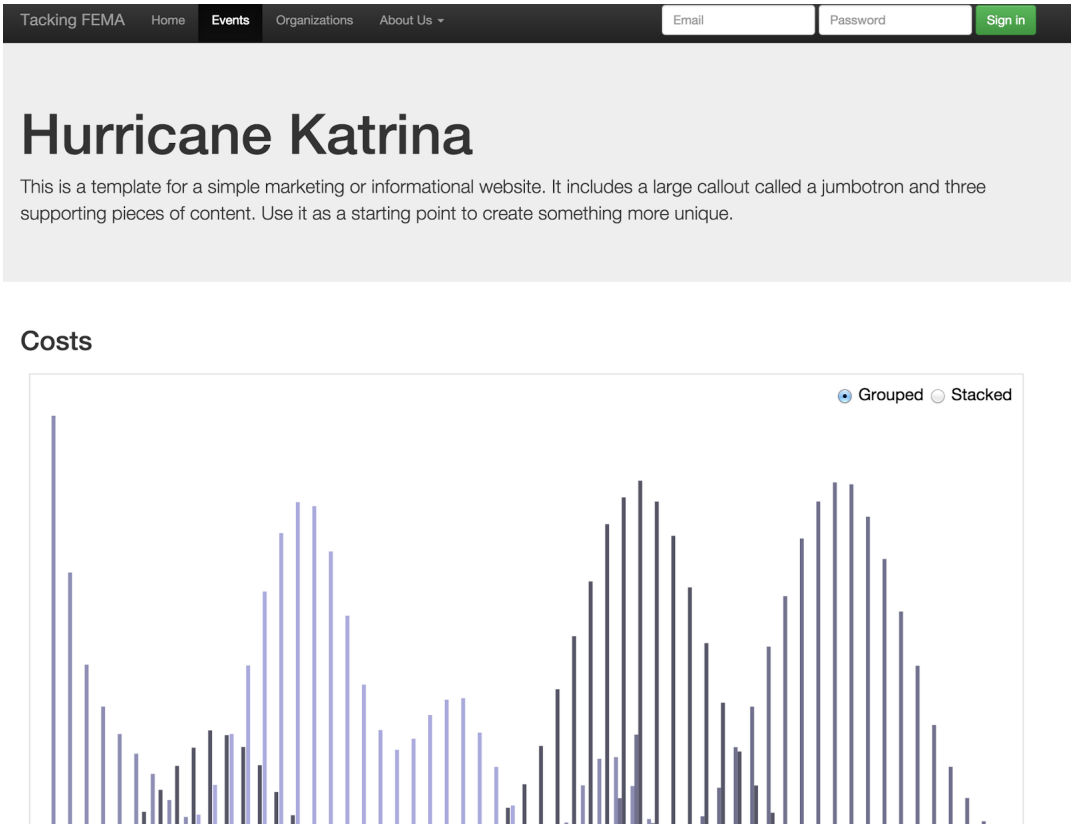


Figure 5: Prototype of Data Page

Refinement of the Homepage

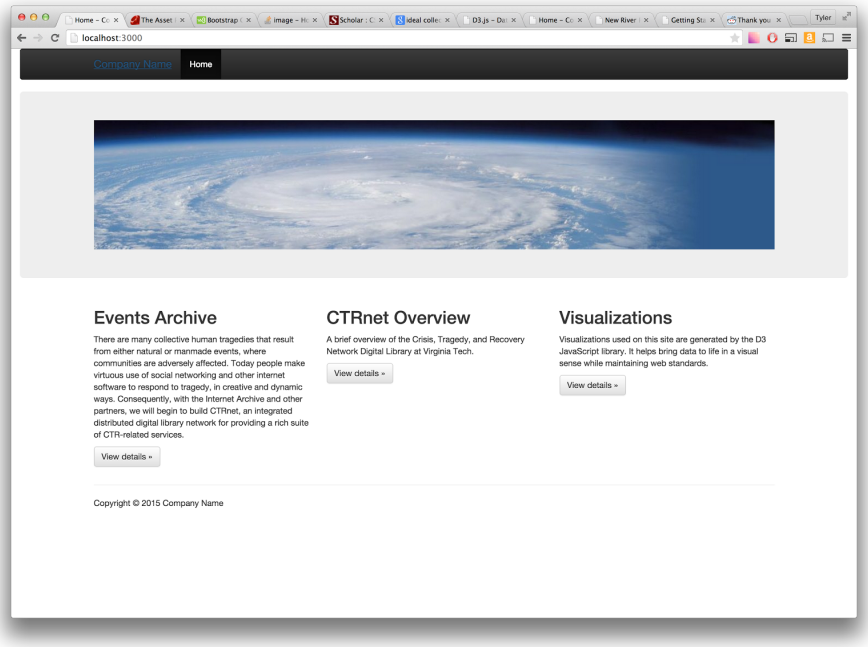


Figure 6: Homepage rough markup

Example Page with Visualization inserted

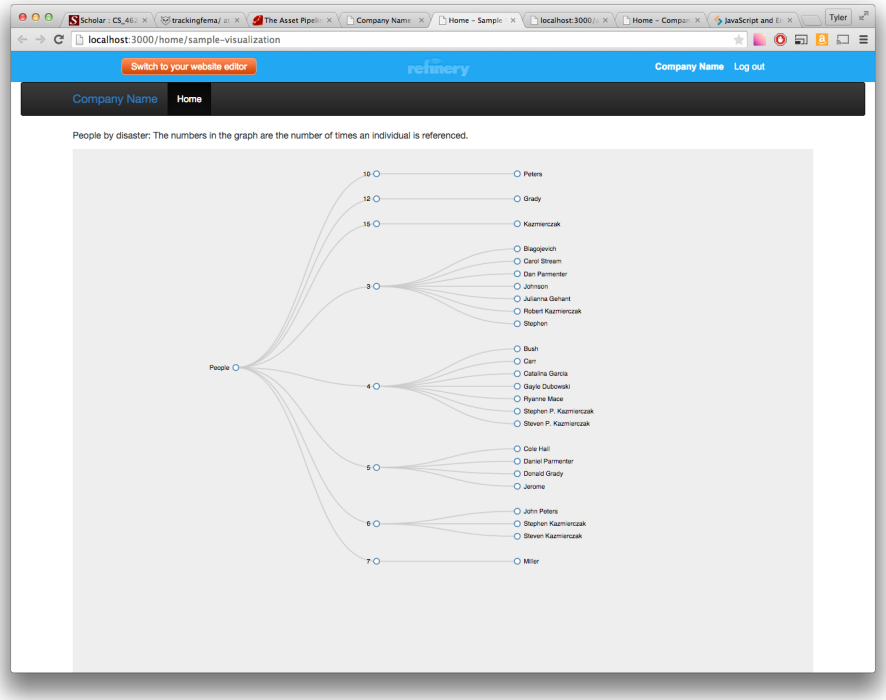


Figure 7: Rough markup of sample visualization page.

Usability and UX Basics

Currently the work in progress front end site is hosted on a development/staging server up on heroku. (Bare in mind this is a work in progress and not completely functional yet, all development is done in a local environment.)

The URL is: http://trackingfema.herokuapp.com/

Part III: Project Timeline

- By the middle of February, have detailed the interface between each part of the project (The interaction between Parsed Strings <-> Processed Strings <-> Visualizations <-> Website)
- By the end of February, have completed the detailed internal plans for each piece, and begin implementation.
- By the middle of March, continue working on implementation and have the basics for a CMS running.
- Midterm Presentation: At this point, each piece of the project should be in a somewhat usable (though buggy) state. Each group member will present their component to the client. The presentations will occur during the last week of March, with the exact date to be decided based on the availability of the group and the client.
- By the end of March: Complete implementation, have all group members rigorously test all pieces.
- By the middle of April, have the individual pieces debugged and completed.
- By the end of April, have all of the pieces working together, with the full system running.
- By the end of the course, reinforce the documentation of the project so that its source can be readily understood by the next group of people.

Part IV: Back-end Implementation

Structure

The data analysis part of this project is split up into two parts: Data Parsing and Data Processing. We are bifurcating the code in this way because the objects of the two components are very different. Data Parsing is focused on extracting info from complex and unreliable pages that were meant to be used by humans. It creates a machine-readable output with a standardized format. The Data Processing component takes that raw output and performs the transforms on it to make useful data for the Data Visualization component.

Data Parsing

Overview

The purpose of the Data Parsing component is to provide plain text to the Data Processing component. The text will come from the articles linked to by the Virginia Tech Events Archive.

Details

This component will consist of a script that produces a collection of text files.

Each disaster from archive-it.org will be stored in a separate folder named “Archive Title”, where Archive Title is the title from https://archive-it.org/organizations/156 (such as “Alabama University Shooting”)

Each article from archive-it.org will be stored as separate text file.

- The text files will follow the naming convention, “Archive Title_#”, where # is the arbitrary identifier of the article. The # may be reused across archives, but not within them.
- The contents of the text file will be split into a metadata section and a contents section, for 8 lines total.
  - The metadata section will consist of the Title, URL, and first capture date of the article, each on separate lines. (found on the archive listing, for example: https://archive-it.org/collections/1829)
  - The text section will consist of five lines. If any line is missing from any article, it will be left blank (the contents of the line will be “\n”). Any extraneous information (e.g. a second subheader) should be ignored
    - The first line will be the header line (stored as plain text with no line breaks)
    - The second line will be the subheader line (stored as plain text with no line breaks)
    - The third line will be the author(s) line (stored as “FirstName MiddleNameOrInitial LastName”)
    - The fourth line will be the date line (stored “DD-MM-YYYY”)

- The fifth line will be the remainder of the text (stored as plain text with no line breaks)

The script will be able to generate folders for specific Archives, provided a command line switch for Archive Title or Collection Number (e.g. “Alabama University Shooting” or “1829” for https://archive-it.org/collections/1829)

Refinement 1

The initial prototype for the Data Parsing stage was fairly simple. The Python script went to the archive-it site, went through the disasters and articles, and processed them. Unfortunately, this architecture did not work except for the very simple prototype. There were two main problems with it.

The first problem was actually getting the raw HTML for the disaster articles. The archive server that holds them is difficult to work with, and often throws HTTP errors at random or when too many articles are being downloaded. Additionally as a general principle, we can’t download articles too quickly.

The second problem is the parsing. The articles come from all over the web, and although there are some that have the same structure (like Wordpress), there are many with their own HTML layouts. We need to be able to extract data from all of these, and it takes lots of trial and error.

The solution to both these problems currently being implemented is to split up the Data Parsing segment into multiple subparts. One part downloads the main page listing disasters, as well as the disaster pages with links to articles. The HTML pages are saved to disc. This way, the database of articles can be built up a piece at a time, without having to redownload everything all the time. The drawback is that the pages take up large amounts of space on disc. Once we have all the kinks in the code worked out and the database downloaded, this might be more of a problem.

The other part does the actual parsing of the HTML articles. It reads the HTML saved to disk into BeautifulSoup, grabs the relevant data out of the HTML tags, and creates the output file for use by the Data Processing section. Since we have a set of the disasters downloaded as HTML, this section can be quickly run without having to download anything from the Internet, so it can easily be changed. This is important for continuing to refine the project, because we will need to run it many times in order to get the data extraction right. This part also generates meta-output, showing what it was and wasn’t able to find. Here’s an example:

['International School Shootings', 'Eight arrested in probe of Jewish seminary attack - CNN.com', True, False, False, False]
['International School Shootings', 'Libya blocks U.N. condemnation of Jerusalem seminary attack - CNN.com', True, False, False, False]
['International School Shootings', 'Blutbad in Winnenden - Amoklauf eines Schülers mit 16 Toten - Panorama - sueddeutsche.de', True, False, False, False]
['International School Shootings', 'Kauhajoki-lautakunta kieltäisi osan käsiaseista | Kotimaa | Kaleva.fi', False, True, False, False]

['International School Shootings', 'Kauhajoen koulusurmien muistopaikka peittyi ruusuihin | Kotimaa | Kaleva.fi', False, True, False, False]
['International School Shootings', 'Jerusalem yeshiva student: I shot the terrorist twice in the head - Haaretz - Israel News', False, False, False, False]
['International School Shootings', 'Winnenden-Amoklauf: "Er war bereit, alles niederzumetzeln" - Nachrichten Vermischtes - WELT ONLINE', True, True, False, False]
['International School Shootings', 'BBC NEWS | Middle East | In quotes: Jerusalem shooting reaction', False, False, False, False]

Each row contains information about a single article. First is the name of the disaster it came from, then the name of the news article. After that are four true or false outputs corresponding to the first four lines in the data output: title, subtitle, author(s), and date. It's true if the script was able to find that data in the HTML, and false otherwise.

The meta-output is very useful for seeing what we still have to work on. The output to the Data Processing section is voluminous, and it's hard to make sense of what needs to be fixed. The meta-output shows one of the most important parts: whether or not the meta-data for each article is being successfully extracted. We are going through articles where it was unsuccessful and changing the parsing code to be more complete.

Refinement 2

This phase was focused on downloading the articles from the archive-it database. Earlier, we had downloaded a small subset of the articles and done the data parsing based off of that. However, since our final goal is to have data from all of the disasters in the database, we decided to take this time to get as many of the articles downloaded as we could. The data parsing involved in is easier than it is in the actual articles, but it's still nontrivial. After this refinement, we have the bulk of the articles downloaded, but there's still some edge cases missing. We plan on tackling those time permitting after working on parsing the articles more.

Final

We spent the remainder of the time working on refining the parser more. One problem that we found later on is that the parser was pulling in large amounts of anchor text from the navigational areas of the page into the output content. This was inadvertently stuffing the output with keywords that didn't really apply to the article. Our first idea was to have the parser find the article contents and only output that. However, we determined that it would be almost impossible to do that in a way that doesn't completely remove the text of many articles. Eventually we decided to move all the anchor text in an article into a separate output section.

Data Processing

Overview

The purpose of the Data Processing component is to provide data to the Data Visualization component.

Details

The Data Processing component will obtain the following information about articles:

- Names of People involved will be extracted using Named Entity Recognition (http://www.nltk.org/book/ch07.html).
- Locations and Agencies will be extracted by searching text for a list of known Locations and Agencies
- Statistics (damages figures, relief costs, fatalities) will be extracted using taggers (http://www.nltk.org/book/ch05.html)
- Types of relief (food, shelter, money)
- Timestamps (Day/time the data was reported) will be extracted using taggers, and through the metadata provided by the Data Parser.
- Phase of Disaster Management (Response, Recovery, Mitigation, Preparedness) will be implemented by empirical analysis of the frequency of keywords in the text.

It will produce output as text files to be read by the Data Visualization section.

Current Progress

Currently, the parser is able to recognize names of people, locations, and agencies, with data structure support for types of relief. It stores these in “reference structures”, which are nested dictionaries.

Reference Structure

The purpose of the reference structure is to capture:

- An entity name
- What document(s) the entity was found in
- The number of times that entity was found each document

The structure consists of:

- A dictionary that maps a location, person, or agency to:
    - A dictionary that maps a document to a reference count

Recognizing Names of People

To recognize names of people, the processor uses the Natural Language Toolkit (NLTK). The bulk of the processing is done in the function get_nltk_nes (short for get nltk named entities) in Processor.py (see Appendix I).

In this function, each document is split into sentences, which are then turned into tokens and analyzed for their part of speech and usage. A good explanation of the details of the process can be found in chapter 7 of the nltk book (http://www.nltk.org/book/ch07.html). If their usage is "PERSON", the reference structure is updated.

Recognizing Agencies and Locations

To recognize Agencies and Locations, the processor uses a list of Agency and Location names loaded by the user of the script. In our case, the names are sourced from:

- Locations: http://jordonmeyer.com/text-list-of-us-cities/
- Agencies: http://en.wikipedia.org/wiki/List_of_charitable_foundations

The Agencies and Locations are read into a set when the ReliefStructures module is loaded, allowing efficient lookup. When a document is being processed, the processor simply looks up the current word in the Locations and Agencies sets. If it finds it, the respective reference structure is updated.

Relief Efforts

Recognizing specific efforts proves to be a difficult problem.
Relief Efforts will have the following attributes, when their recognition is working:

- A type categorization based on the types of relief defined by http://www.irs.gov/Charities-&-Non-Profits/Charitable-Organizations/Disaster-Relief:-Types-of-Assistance-Charity-May-Provide.
  That is:
  - Food
  - Clothing
  - Housing
  - Transportation
  - Medical
  - Cash Equivalent
- A boolean indicating whether the relief is long-term or short-term
- An 'amount' of relief, defined in some 'units'.

For example, if an article details an agency providing five hundred food rations to hurricane survivors, the Effort object will be:

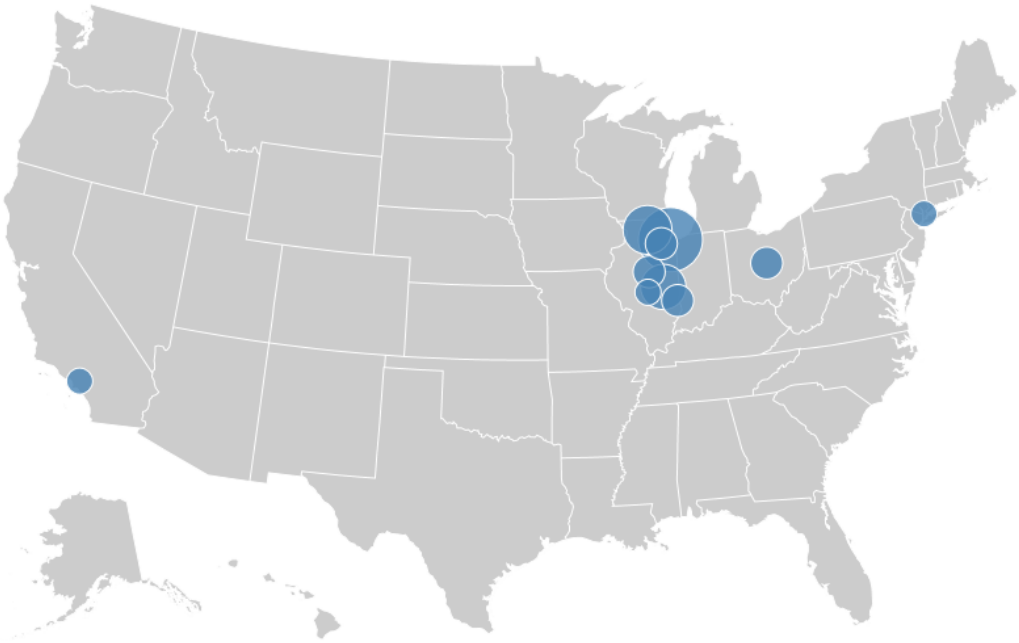
- Type: Food
- 500 "food rations"

- Short-Term

Data Visualization

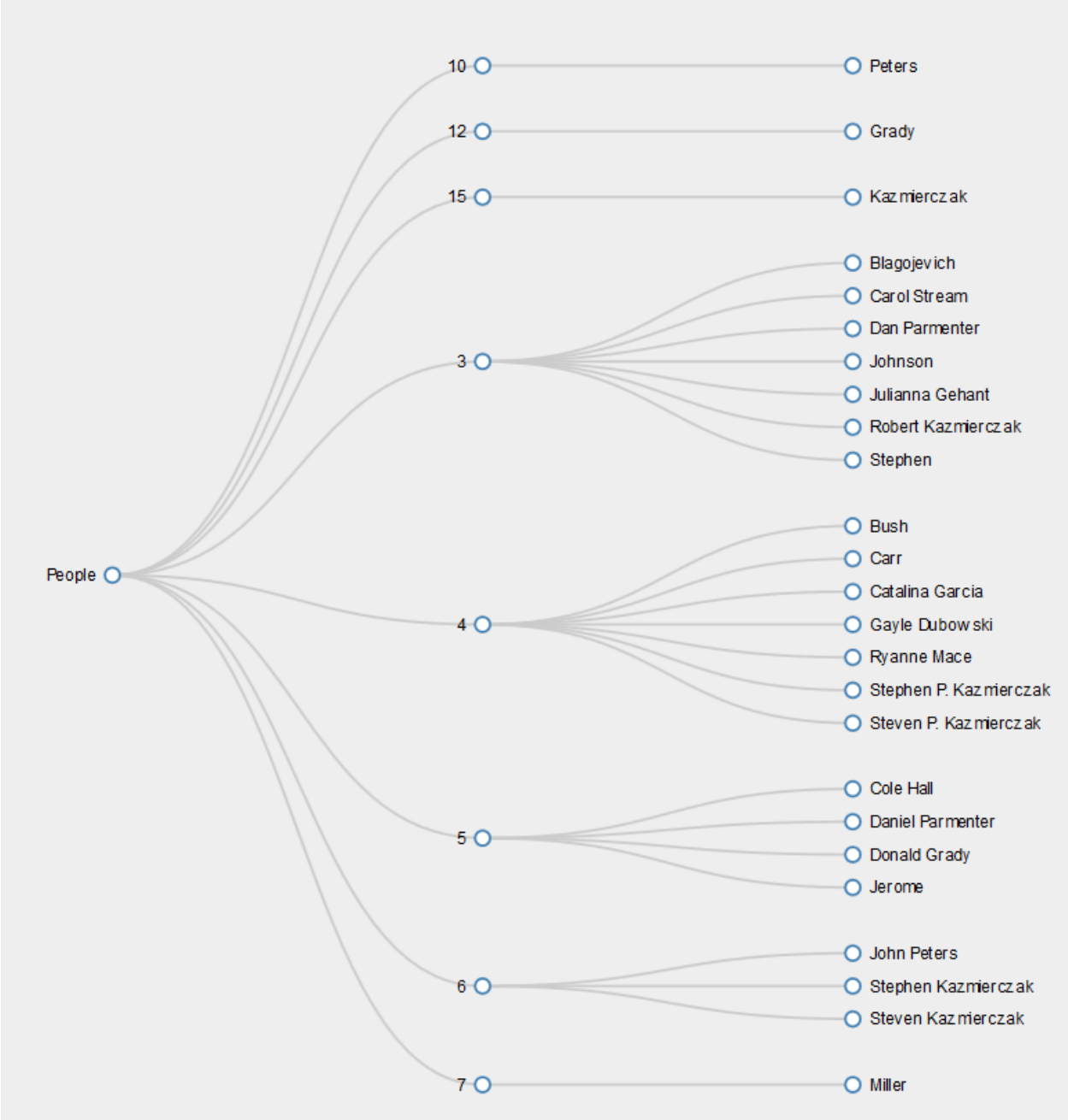
This data we parsed which represented locations involved and the people involved in the relief efforts of disasters will be graphically represented.

The first graph will show the locations which were involved with the relief efforts. To show this most thoroughly, we created a map of the United States. On this map, we place circles which grow based on the number of times a city is referenced. Appendix V shows the HTML code associated with this graph. However, it does use a couple of json data files which are too large to include which builds the map as well as identifies the cities and reference counts. For example, for the Northern Illinois University Shooting, the following graph represents the cities involved with its relief and quantifies the amount that they are involved.

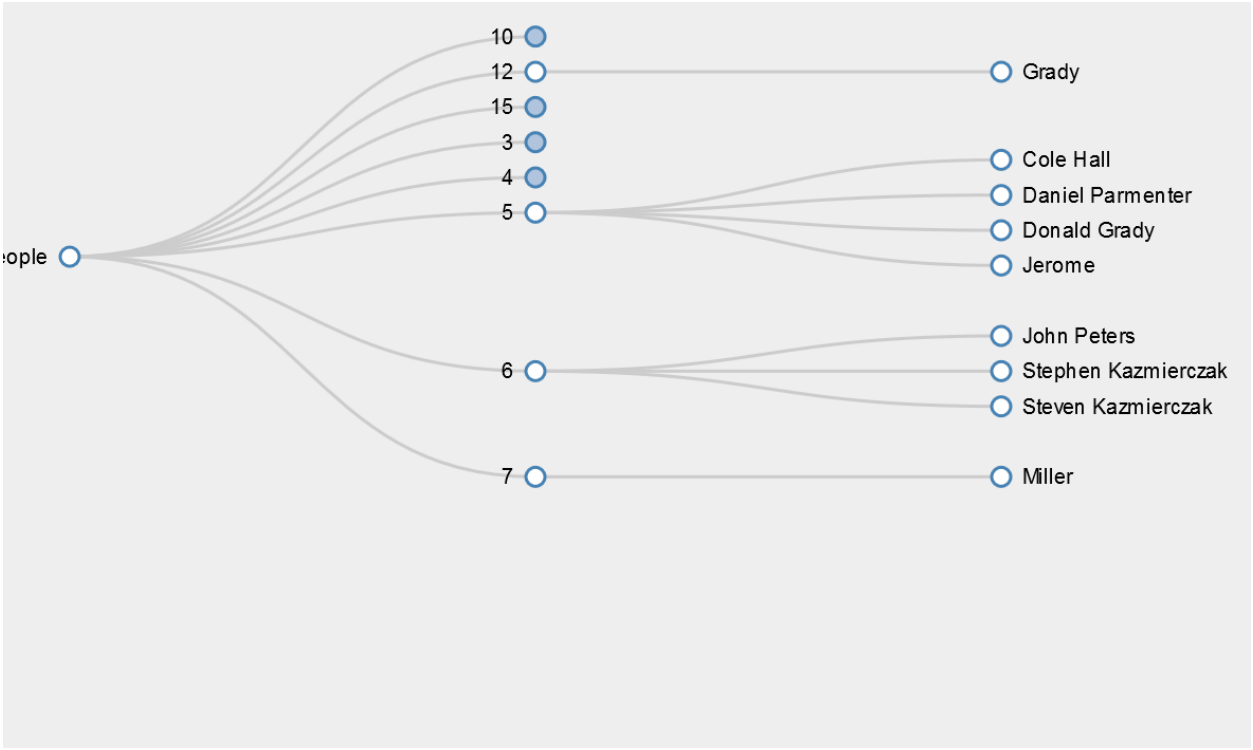


Furthermore, we were able to mine information regarding individuals who were involved in the relief efforts of a disaster. We compiled this information into a graph which would be expandable so the user could decide how much information and when the user would like to see the information. This list was sorted by the number of times an individual was referenced in regard to a disaster. The numbers in the graph represent the number of articles which reference the individual's efforts. Appendix VI shows the html code associated with this graph data tree. However it does use a couple of json data files which are too large to include which builds the tree and has the containers for the raw data of the people associated.

The following graph is a fully expanded tree of the people involved in the Northern Illinois University Shooting:



Here is another example of the same graph being condensed a bit for the user to more easily see the data, additionally, the user can pan across and zoom in and out in the graph to focus on different parts, also shown in following image of the graph:



Part V: Testing

Testing the Parsing Scripts

Disasters and Articles

The disasters.py and articles.py were not tested during this phase. The reason for this is that they had already been examined and issues found during prior phases, and have not changed significantly since the beginning. The scripts were used to download the set of archived articles for each of the disasters, and they have performed well. All articles that don't give a 404 error on the archive server have been downloaded. The main known issue for this scripts is that when a disaster has multiple pages of articles in its listing, the script only downloads the first page. We decided not to fix this problem due to the limited time available. The first page already has 100 articles for a disaster, and we have thousands of articles to process.

Processing.py

This script was the one we tested during this phase, and justifiably so, as it is much more complex and error-prone than the others. During prior phases, we did some simple testing where we looked at statistics in the parser output. We checked what percentage of the time the parser was able to find each piece of metadata (title, subtitle, date, and author), and then wrote more parsing rules to improve it. However, this testing method suffered from a serious flaw. Not every article has every piece of metadata, and avoiding false positives in the output

is just as important as avoiding false negatives. Simply listing the percentages that the parser was able to find something didn't tell us very much.

We decided to take a different approach during the testing phase. Instead of automated statistics, we performed manual verification. We randomly selected 20 articles and "audited" the parsed output file. We manually looked for the title, subtitle, author, and date in the article, and checked it against what the parser was able to find. This is our data table for the initial test:

File	Title found	Title Correct?	Subtitle found	Subtitle correct?	Author found	Author correct?	Date found	Date correct?
Youngstown Shootings #55	Yes	Yes	Yes	Yes	No	No	No	No
Norway Shooting July 23, 2011 #22	No	Yes	No	Yes	No	Yes	No	Yes
Hurricane Sandy (October 2012) #84	Yes	No	Yes	Yes	No	Yes	No	Yes
Virginia Tech Global Disasters Collection #82	Yes	Yes	Yes	Yes	No	No	No	No
Midwest Snowstorms (Feb 2011) #59	Yes	Yes	Yes	Yes	No	No	No	No
Indonesia Plane Crash	No	No	Yes	Yes	No	Yes	No	No

(September 2011) #18								
Texas Wild fire 2011 #32	No	No	No	No	No	Yes	No	No
Turkey EarthQuake(October 2011) #39	No	Yes	Yes	Yes	No	No	No	No
Encephalitis (India) #27	No	Yes	Yes	Yes	No	Yes	No	Yes
Guatemala Earthquake #47	Yes	Yes	Yes	Yes	No	No	No	No
Virginia Tech Global Disasters Collection #67	Yes	Yes	No	Yes	Yes	Yes	No	No
Virginia Earthquake (Aug 23rd, 2011) #49	No	Yes	No	Yes	No	Yes	No	Yes
Indonesia Plane Crash (September 2011) #93	No	Yes	No	Yes	No	Yes	No	Yes
Hurricane	Yes	Yes	Yes	Yes	No	Yes	No	Yes

Sandy (October 2012) #23								
East River Helicopter Crash (October 5th, 2011) #55	Yes	Yes	No	No	No	Yes	No	No
Midwest Snowstorms (Feb 2011) #12	No	No	No	No	No	No	No	Yes
Hurricane Irene (Aug 2011) #21	Yes	Yes	No	No	No	No	No	No
Somalia Bomb Blast #11	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Virginia Earthquake (Aug 23rd, 2011) #33	Yes	Yes	No	No	No	No	No	No
Global Food Crisis #28	Yes	Yes	Yes	Yes	No	Yes	No	No

Correct title: 16/20 80%
Correct subtitle: 15/20 75%
Correct author: 12/20 60%
Correct date: 8/20 40%

The results were generally as expected. The parser was usually able to find title and subtitle data, since that is stored is standardized things such as header tags. Author and date are more difficult to find, with different sites using different layouts. As we performed the tests, whenever the parser was found to have an incorrect output we added a new rule (or changed an existing one) to fix it.

After performing the fixes, we were not yet sure how much they had improved the script. We generated a new set of parsed output files with the new rules and did another audit of 20 random articles:

File	Title found	Title Correct?	Subtitle found	Subtitle correct?	Author found	Author correct?	Date found	Date correct?
Virginia Tech Global Disasters Collection #76	Yes	Yes	No	Yes	Yes	No	No	No
South-Eastern US Storms #72	Yes	Yes	Yes	Yes	Yes	Yes	No	No
CTRnet - Emergency Preparedness information #75	Yes	Yes	No	Yes	Yes	Yes	No	No
Virginia Earthquake (Aug 23rd, 2011) #27	Yes	Yes	No	Yes	No	Yes	Yes	Yes
Tucson Shooting Anniversary 2012 #18	Yes	Yes	Yes	Yes	No	Yes	No	Yes
Hurricane Irene (Aug 2011) #35	No	Yes	Yes	Yes	No	Yes	No	Yes
Turkey EarthQua	Yes	Yes	No	Yes	Yes	Yes	No	Yes

ke(October 2011) #32								
Indonesia Plane Crash (September 2011) #20	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Guatemala Earthquake #50	Yes	Yes	Yes	No	Yes	Yes	No	Yes
Nevada air race crash (September 16, 2011) #2	Yes	No	Yes	No	Yes	No	No	No
Somalia Bomb Blast #13	Yes	Yes	No	Yes	Yes	Yes	Yes	No
China Floods #38	Yes	Yes	No	No	No	Yes	No	Yes
April 16 Archive #5	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Philippines Floods (September 2011) #42	Yes	Yes	No	Yes	No	No	No	No
April 16 Archive #42	No	Yes	Yes	Yes	No	Yes	No	Yes
Virgnia Tech April16 Shootings	Yes	Yes	Yes	Yes	No	Yes	No	Yes

Remembrance #27								
New Zealand Earthquake #27	Yes	Yes	No	No	Yes	Yes	No	No
China Floods #35	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Chile Earthquake #1	Yes	Yes	Yes	Yes	Yes	No	No	No
Texas Wild fire 2011 #21	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Correct title: 19/20 95%
Correct subtitle: 16/20 80%
Correct author: 15/20 75%
Correct date: 12/20 60%

All of the statistics improved. At first glance, the fixes appeared to have improved the parser by quite a bit. Unfortunately, it's hard to tell for sure. Our sample size for the two tests was not very large, because each test required manual examination of the article and the parsed data. We may have gotten lucky by drawing a better set of articles this time.

Overall we found several flaws in the parser. One is that some of the rules for meta tags are not working properly. The cause of that is unknown at this time. Another is that titles and subtitles are different in theory, but often overlap in practice. Some articles parse out a subtitle without a title. In most of these cases, the "subtitle" would be better off parsed as a title, even though the HTML it came from would be considered a subtitle in other articles. So if the parser finds a subtitle but not a title, it should promote the subtitle to title.

These two fixes are not easy, and go into another problem that became clear while we were adding fixes: the parser's current architecture is poor. There are large amounts of cut-and-paste code from when we thought we only needed a few parsing rules. The code is overall very brittle. For the final product, we are going to refactor the parsing code and fix the problems that were discovered during testing.

Testing the Processing Scripts

The Mention files

The mention files were tested by randomly sampling several of the files. For each randomly selected file, the following tests were performed:

- Ensure that their is a mention file for each article
- Ensure that each mention file consists of Locations, Agencies, and Names (in that order)
- Ensure that the number of locations, agencies, and names listed is accurate.

The testing revealed the following issues:

- Some files have non-ASCII characters. The processing script needs to be updated to support them.
- When people are referred to by their last name and first name, there are separate entries for Firstname Lastname and just Lastname.
- Locations are limited to the list in Locations.txt, so some disasters (Japan Earthquake) don't get many mentions.

The Summary files

- Ensure that the mention count for a string in the mention file matches the number of occurrences in the raw file
- Ensure that the number of associated entities for each disaster matches matched the number of entities that were printed out
- Check to make sure the entities make sense
- Check to make sure the results make sense in context

The testing revealed the following issues:

- Whenever "New York" is mentioned, "York" is also recorded as being mentioned. That is, entities whose names are a substring of another entities name will be matched whenever the longer name is mentioned
- The nltk's name recognizer produces strange names. "Reader Editor, Timeline, Staff Parents, Repeat Insurance Inventory Legal Legal Admin", etc. were all recorded as being names of people. Filtering out all names which occur less than 3 times eliminates most of these erroneous entries.

Part VI: User Guide - How to run the application

Getting the application up and running is actually quite simple. All you need is an environment setup with Ruby, Rails, and Python. Unzip the project folder to the directory of your choosing and then navigate to that folder within a terminal. Simply run the following: 1) *bundle install* 2) *rake db:migrate* 3) *rails server*, this will start the rails application and you should be good to go. The other python scripts if needed also need to be manually run from the command line by typing python "*script name"*.

Appendices

Appendix I: Processor.py

```
"""Processor.py : Finds relationships between relief objects in article
documents."""
import os
import argparse
import multiprocessing
import logging
import nltk
from Document import Document
from ReliefStructures import Location, Person, Agency, NamedEntity, Relief,
ReliefType
from ProcessorUtil import ProcessorUtil

logging.basicConfig(level=logging.INFO)  # Comment this line out to silence the
script
OUT_DIR = os.path.join(os.path.dirname(__file__), "Output")
SUMMARY_DIR = os.path.join(os.path.dirname(__file__), "Summary")

def get_entities(named_entity_class, document):
    ref_dict = dict()
    for name in named_entity_class.names:
        if name in document.content:
            entity = named_entity_class(name)
            try:
                ref_dict[name] += 1
            except KeyError:
                ref_dict[name] = 1
    return ref_dict


# adapted from
http://timmcnamara.co.nz/post/2650550090/extracting-names-with-6-lines-of-pytho
n-code
def get_nltk_nes(document, node_type):
    """A generator that yields the names of objects of type node_type"""
    for sentence in document.sentences:
        for chunk in nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(sentence))):
            if hasattr(chunk, "label") and chunk.label() == node_type:
                yield [leaf[0] for leaf in chunk.leaves()]


def get_people(names, document):
    """Gets people based on a list of their names"""
    people = dict()
```

```
    for name in names:
        person = Person(" ".join(name))
        try:
            people[person] += 1
        except KeyError:
            people[person] = 1
    return people


def generate_meta(document):
    locations = get_entities(Location, document)
    agencies = get_entities(Agency, document)
    people = get_people(get_nltk_nes(document, "PERSON"), document)
    # Write each dictionary's contents to a length-delimited file
    with open(os.path.join(OUT_DIR, os.path.basename(document.filename)), "w")
as outfile:
        # Write the locations
        outfile.write(str(len(locations.keys())) + "\n")
        for location, count in locations.items():
            outfile.write(str(location) + " " + str(count) + "\n")

        # Write the agencies
        outfile.write(str(len(agencies.keys())) + "\n")
        for agency, count in agencies.items():
            outfile.write(str(agency) + " " + str(count) + "\n")

        # Write the people
        outfile.write(str(len(people.keys())) + "\n")
        for person, count in people.items():
            outfile.write(str(person) + " " + str(count) + "\n")


def process_file(filename):
    logging.info("Loading article %s", filename)
    try:
        doc = Document(os.path.join("Articles", filename))
    except UnicodeDecodeError as ude:
        logging.warning("Unable to load file %s: \"%s\"", filename, ude.reason)
        return
    generate_meta(doc)


def get_from_meta(filename):
    locations = dict()
    agencies = dict()
    people = dict()
    with open(os.path.join(OUT_DIR, filename), 'r') as meta_file:
        # First line tells us how many locations
```

```
        first_line = meta_file.readline()
        num_locations = 0
        try:
            num_locations = int(first_line)
        except ValueError as e:
            logging.warning("Unable to read file %s due to: unable to parse
\"%s\"", filename, first_line.strip())
            return None, None, None
        for i in range(num_locations):
            location = meta_file.readline().strip().rsplit(" ", 1)
            locations[location[0]] = location[1]

        # Second line tells us how many agencies
        for i in range(int(meta_file.readline())):
            agency = meta_file.readline().strip().rsplit(" ", 1)
            agencies[agency[0]] = agency[1]

        # Third line tells us how many people
        for i in range(int(meta_file.readline())):
            person = meta_file.readline().strip().rsplit(" ", 1)
            people[person[0]] = person[1]

    return tuple([{filename: locations}, {filename: agencies}, {filename:
people}])


def multiprocess_meta():
    # Make a pool using a process for all but one of the cores (leave one for
us)
    p = multiprocessing.Pool(multiprocessing.cpu_count() - 1)
    # Go through each file in an "Articles" sub directory
    dirs = os.listdir("Articles")
    # Spread the workload across each processor
    num_chunks = int(len(dirs) / (multiprocessing.cpu_count() - 1))
    logging.info("Multiprocessing %d files in groups of %d distributed across %d
processes.",
                 len(dirs), num_chunks, multiprocessing.cpu_count())
    p.map(process_file, dirs, num_chunks)


def process(read_articles=False, write_summaries=False):
    # Create the output directory if needed
    if not os.path.exists(OUT_DIR):
        os.mkdir(OUT_DIR)

    # First, process each article to a smaller, quick-to-read file
    if read_articles:
        multiprocess_meta()
```

```
    # Second, read those processed files
    # The dicts have the format: {filename: {entity_name: occurrence_count}}
    locations = dict()
    agencies = dict()
    people = dict()
    for locations_part, agencies_part, people_part in map(get_from_meta,
os.listdir(OUT_DIR)):
        if locations_part is None or agencies_part is None or people_part is
None:
            continue
        locations.update(locations_part)
        agencies.update(agencies_part)
        people.update(people_part)

    # Third, write summary information
    if write_summaries:
        if not os.path.exists(SUMMARY_DIR):
            os.mkdir(SUMMARY_DIR)

        ProcessorUtil.write_entity_mentions_by_disaster(locations, "location",
SUMMARY_DIR)
        ProcessorUtil.write_entity_mentions_by_disaster(agencies, "agency",
SUMMARY_DIR)
        ProcessorUtil.write_entity_mentions_by_disaster(people, "person",
SUMMARY_DIR)

    logging.info("Done")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--parse_articles", help="parse the raw article files",
action="store_true")
    parser.add_argument("--write_summaries", help="Write the summary files for
use in graphs", action="store_true")
    args = parser.parse_args()
    if args.parse_articles or args.write_summaries:
        process(read_articles=args.parse_articles,
write_summaries=args.write_summaries)
    else:
        parser.print_help()
```

Appendix II: Document.py

```
""" Document.py : Contains the Document class, which represents a document as output from the data parsing segment."""
import nltk

class Document(object):
    """
    Represents a document, and handles loading documents from text files via the init method.
    """

    LINE_HEADER = 0
    LINE_SUBHEADER = 1
    LINE_AUTHOR = 2
    LINE_DATE = 3
    LINE_CONTENT = 4

    filename = ""
    """The name of the parsed file this was loaded from"""

    disaster_id = ""
    """The name of the disaster the document relates to"""

    disaster_nid = ""
    """The number of the document in the disaster listing"""

    header = ""
    """The document's header, if specified"""

    subheader = ""
    """The document's subheader, if specified"""

    author = ""
    """The document's author, if specified"""

    date = ""
    """The document's date, if specified"""

    content = ""
    """The document's raw content, if specified"""

    words = list()
    """A list of words created with nltk.word_tokenize"""

    def __init__(self, filename):
        self.filename = filename
```

```
        self.disaster_id, self.disaster_nid = filename.rsplit("#", 1)
        with open(filename, "r") as docfile:
            raw_data = docfile.readlines()
            self.header = raw_data[self.LINE_HEADER].strip()
            self.subheader = raw_data[self.LINE_SUBHEADER].strip()
            self.author = raw_data[self.LINE_AUTHOR].strip()
            self.date = raw_data[self.LINE_DATE].strip()
            self.content = raw_data[self.LINE_CONTENT].strip()
            self.words = nltk.word_tokenize(self.content)
            self.sentences = nltk.sent_tokenize(self.content)

    def __hash__(self):
        return hash(self.filename)

    def __eq__(self, other):
        return self.filename == other.filename
```

Appendix III: ReliefStructures.py

```
"""ReliefStructures.py : Contains data structures for parties involved in
relief efforts"""

from enum import Enum
import logging

logging.basicConfig(level=logging.INFO)  # Comment this line out to silence the
script


class NamedEntity(object):
    """
    Interface for entities with proper names
    """

    NAMES_FILE = ""
    """Name of the file that contains the names for this entity"""

    @staticmethod
    def load_names(ne_class):
        """Loads names for the given named entity class"""
        with open(ne_class.NAMES_FILE, "r") as names_file:
            for line in names_file:
                ne_class.names.add(line.strip())

    name = ""
    """The actual matching name from NamedEntity.names"""

    def __init__(self, name):
        self.name = name

    def __hash__(self):
        return hash(self.name)

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name


class Location(NamedEntity):
    """
    Represents a location
    """
```

```
    NAMES_FILE = "Locations.txt"
    names = set()

    def __init__(self, name):
        NamedEntity.__init__(self, name)


class Person(NamedEntity):
    """
    Represents a person
    """
    NAMES_FILE = None
    names = set()

    def __init__(self, name):
        NamedEntity.__init__(self, name)


class Agency(NamedEntity):
    """
    Represents an agency
    """
    NAMES_FILE = "Agencies.txt"
    names = set()

    def __init__(self, name):
        NamedEntity.__init__(self, name)


class ReliefType(Enum):
    """
    (See
http://www.irs.gov/Charities-&-Non-Profits/Charitable-Organizations/Disaster-Re
lief:-Types-of-Assistance-Charity-May-Provide
    for a good list of types)
    """
    UNCLASSIFIED = 0
    Food = 1
    Clothing = 2
    Housing = 3
    Transportation = 4
    Medical = 5
    CashEquivalent = 6


class Relief():
    """
    Represents aid provided
```

```
    """
    long_term = False
    """Whether aid is long-term"""

    amount = 0
    """Amount of relief given, in units 'unit'"""

    unit = ""
    """Unit for amount of relief (dollars, # cans of beans, etc)"""

    type = ReliefType.UNCLASSIFIED
    """Type of relief provided"""

logging.info("Loading location names.")
NamedEntity.load_names(Location)
logging.info("Loading agency names.")
NamedEntity.load_names(Agency)
logging.info("Locations loaded: \n" + str(Location.names) + "\n")
logging.info("Agencies loaded: \n" + str(Agency.names) + "\n")
```

Appendix IV: ProcessorUtil.py

```
import os
import logging
# Makes it easy to represent the number of location mentions
# Creates a file with the following format:
# First line: document_name number_of_locations
# Second line: location_name occurrence_count

class ProcessorUtil:
   @staticmethod
   def entity_mentions_by_disaster(references):
       """Returns a dict of disasters from a dict of references: {disaster:
{entity_name: count}}"""
       disaster_dict = dict()
       for doc_name, ref_dict in references.items():
           disaster_name = doc_name.rsplit("#", 1)[0]
           for entity_name, count in ref_dict.items():
               if disaster_name in disaster_dict:
                   try:
                       disaster_dict[disaster_name][entity_name] += int(count)
                   except KeyError:
                       disaster_dict[disaster_name][entity_name] = int(count)
               else:
                   disaster_dict[disaster_name] = {entity_name: int(count)}
       return disaster_dict

   @staticmethod
   def write_entity_mentions_by_disaster(references, entity_type, summary_dir):
       # First, we need to get a dict
       disaster_dict = ProcessorUtil.entity_mentions_by_disaster(references)

       # Then, we need to print out that dict
       with open(os.path.join(summary_dir, entity_type +
"_mentions_by_disaster"), "w") as summary_file:
           for disaster_name, ref_dict in disaster_dict.items():
               summary_file.write(disaster_name + " " + str(len(ref_dict)) +
"\n")
               for entity_name, count in ref_dict.items():
                   summary_file.write(entity_name + " " + str(count) + "\n")


   @staticmethod
   def entity_relationships_by_disaster(references):
```

Appendix IV: index.html

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.states {
  fill: #ccc;
  stroke: #fff;
}

.symbol {
  fill: steelblue;
  fill-opacity: .8;
  stroke: #fff;
}

</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="http://d3js.org/topojson.v1.min.js"></script>
<script src="http://d3js.org/queue.v1.min.js"></script>
<script>

var width = 960,
      height = 500;

var radius = d3.scale.sqrt()
      .domain([0, 1e6])
      .range([0, 10]);

var path = d3.geo.path();

var svg = d3.select("body").append("svg")
      .attr("width", width)
      .attr("height", height);

queue()
      .defer(d3.json, "/us.json")
      .defer(d3.json, "/us-state-centroids.json")
//    .defer(d3.json, "/test.json")
      .await(ready);

function ready(error, us, centroid) {
  svg.append("path")
      .attr("class", "states")
      .datum(topojson.feature(us, us.objects.states))
```

```
      .attr("d", path);

  svg.selectAll(".symbol")
      .data(centroid.features.sort(function(a, b) { return
b.properties.occurrence - a.properties.occurrence; }))
      .enter().append("path")
      .attr("class", "symbol")
      .attr("d", path.pointRadius(function(d) { return
radius(d.properties.occurrence * 500000); }));
}

</script>
```

Appendix VI: people.html

```
<!DOCTYPE html>
<meta charset="utf-8">
<style type="text/css">

.node {
      cursor: pointer;
  }

  .overlay{
      background-color:#EEE;
  }

  .node circle {
      fill: #fff;
      stroke: steelblue;
      stroke-width: 1.5px;
  }

  .node text {
      font-size:10px;
      font-family:sans-serif;
  }

  .link {
      fill: none;
      stroke: #ccc;
      stroke-width: 1.5px;
  }

  .templink {
      fill: none;
      stroke: red;
      stroke-width: 3px;
  }

  .ghostCircle.show{
      display:block;
  }

  .ghostCircle, .activeDrag .ghostCircle{
      display: none;
  }

</style>
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
<script src="http://d3js.org/d3.v3.min.js"></script>
```

```
<script src="dndTree.js"></script>
<body>
<script>
<div id="tree-container"></div>
</script>
```

Works Cited

"Bootstrap · The World's Most Popular Mobile-first and Responsive Front-end Framework." *Bootstrap · The World's Most Popular Mobile-first and Responsive Front-end Framework.* Web. 15 Mar. 2015. <http://getbootstrap.com/>.

"Events Archive." *Events Archive*. Web. 16 Mar. 2015. <http://www.ctrnet.net/>.

"Heroku." *Heroku*. Web. 15 Mar. 2015. <http://heroku.com/>.

"Internet Archive Wayback Machine." *Internet Archive Wayback Machine*. Web. 16 Mar. 2015. <http://wayback.archive-it.org/>.

"Natural Language Toolkit¶." *Natural Language Toolkit — NLTK 3.0 Documentation*. Web. 15 Mar. 2015. <http://www.nltk.org/>.

"Refinery CMS." *Ruby on Rails CMS That Supports Rails 4.1 -*. Web. 15 Mar. 2015. <http://refinerycms.com/>.

"Ruby." *Programming Language*. Web. 15 Mar. 2015. <http://www.ruby-lang.org/en/>.

"Ubuntu 14.04.2 LTS (Trusty Tahr)." *Ubuntu 14.04.2 LTS (Trusty Tahr)*. Web. 15 Mar. 2015. <http://releases.ubuntu.com/14.04/>.

"Web Development That Doesn't Hurt." *Ruby on Rails*. Web. 15 Mar. 2015. <http://rubyonrails.org/>.