Trail Study

Final Report

CS 4624: Multimedia, Hypertext, and Information Access

Virginia Polytechnic Institute and State University

Blacksburg, VA 24061

Instructor: Dr. Edward A. Fox

Client: Abigail Bartolome

May 2, 2018

Marshall Hansen, Kevin Cianfarini, Andrew Eason, Shane Davies
Table of Contents

1. Table of Figures .................................................................................................2
2. Table of Tables ...................................................................................................3
3. Executive Summary ...........................................................................................4
4. Introduction ........................................................................................................5
   4.1 Objectives ......................................................................................................5
   4.2 Terminology ..................................................................................................5
   4.3 Statement of Functionality .........................................................................6
   4.4 Scope ...........................................................................................................6
5. Requirements ......................................................................................................8
6. Design ..................................................................................................................9
7. Implementation ...................................................................................................10
8. Testing/Evaluation/Assessment ........................................................................14
9. Finished Product ...............................................................................................15
   9.1 Application Frontend ..............................................................................15
   9.2 Example of a Scraped Blog Post ..............................................................16
10. User’s Manual ..................................................................................................17
11. Developer’s Manual .........................................................................................20
12. Lessons Learned .............................................................................................20
13. Future Work .....................................................................................................22
14. Acknowledgements .........................................................................................23
15. References .......................................................................................................23
1. Table of Figures

1. Architecture diagram for TrailStudy project ..........................................................10
2. Example of Django management commands..........................................................11
3. Example of an API query ......................................................................................11
4. Example of the JSON response returned from the API ........................................11
5. Example 2 of an API Query ................................................................................11
6. JSON Response Sample ......................................................................................12
7. Application Frontend .........................................................................................15
8. Original Blog Post ..............................................................................................16
9. Scraped Blog Post ..............................................................................................17
10. BASH command used to navigate into the capstone directory ............................17
11. BASH command used to create a virtual environment .........................................17
12. BASH command to check current version of Python ..........................................18
13. BASH command to check if Python3 is installed ...............................................18
14. BASH command to install project requirements and start server ....................19
15. BASH command to scrape all the blog posts ....................................................19
16. BASH command to tag all the gathered blog posts ..........................................19
17. BASH command to run the server ....................................................................19
18. BASH command to export data to CSV .............................................................19
19. BASH command to apply database migrations .................................................20
20. Blog site file formatting example ....................................................................20
2. Table of Tables

1. Table of Milestones........................................................................................................6
2. Submission Statistics......................................................................................................15
3. Blog Data Saved in Database .........................................................................................16
3. Executive Summary

This project is focused on the culture and trends of the Triple Crown Trails (Appalachian Trail, Pacific Crest Trail, and Continental Divide Trail). The goal of this project is to create a large collection of forum and blog posts that relate to the previously stated trails through the use of web crawling and internet searching. One reason for this project is to assist our client with her Master's Thesis. Our client, Abigail Bartolome is focusing her thesis on the different trends and different ways of life on the Triple Crown Trails, and the use of our tool will help her. The impact of our project is that it will allow our client to be able to sift through information much faster in order to find what she does and does not need for her thesis, instead of wasting time searching through countless entries with non-relevant information. Abigail will also be able to sift through what kind of information she wants specifically through the use of our tagging system. We have provided the dates, titles, and author of each post so she can immediately see if the article has relevant information and was posted in a timeframe that is applicable.

The project will have two main focuses, the frontend and the backend. The frontend is an easy-to-use interface for Abigail. It will allow her to search for specific tags, which will filter the blog posts based on what information she seeks. The tags are generated automatically based on the content of all of the forums and blogs together, making them very specific which is good for searching for the kind of content desired by our client. When she finishes adding tags, she can then search for blogs or forums that relate to the topics tagged. The page will display them in a neat format with the title of the article that is hyperlink-embedded so she can click on it to see the information from the article, as well as the author, date, and source of the post.

The backend is where all the heavy lifting will be done, but obviously is invisible to the client. This is where we will go through each of the blog or forum websites fed into the web crawler to store all of the relevant information into our database. The backend is also where the tagging system is implemented and where tags are generated and applied to blog posts. WordPress and BlogSpot (for the most part) have a uniform way of going through blogs, so our web crawler acts accordingly based on which website it is, and is able to go through until there are no more blogs on that site. All of the blog posts, contents, pictures, tags, URLs, etc. are stored in the backend database and then linked to our frontend so that we can display it neatly and organized to the liking of Abigail. From 31 sources we have collected 3,423 blog posts to which have been assigned 87,618 tags.

Together, the frontend and the backend provide Abigail with a method to both search and view blog post content in an efficient manner.
4. Introduction

4.1 Objectives

The objective of this project is to create a web crawling tool for our client in order for her to be able to quickly search through numerous blog posts and forums to find relevant information for her Master’s Thesis. Ultimately what our tool will be used for is to scrape information from hiking blogs and forums and sort the information gathered into a neat report-like format where the client can read through it with ease to find the information desired.

Our goals will be accomplished by breaking the project into three main sections: web crawling, tagging, and sorting. In the web crawling phase we will create a web crawler using Python, BeautifulSoup4, and requests. The crawler will be able to search through Blogspot and WordPress websites/blogs and extract all the information including author, title, text, etc. and sort it into a neat format that is easily readable. Tagging is essentially just allowing Abigail to sort through the different articles by applying tags, or filters, to her searches; this will allow her to find more quickly that information that she wants. For example if she only wants information pertaining to the Appalachian Trail and Trail Angels, she can apply those tags to her searches so that our web crawler only returns information that is related to that. Our final step is just how we will sort the information for our client. This step isn’t too hard and is pretty self explanatory but we just want to make sure that we present the information to our client in the best way possible.

4.2 Terminology


2. **PCT** - Pacific Crest Trail

3. **AT** - Appalachian Trail

4. **CDT** - Continental Divide Trail

5. **Trail Angel** - Good Samaritans on many hiking trails who replenish aid stations and help out hikers

6. **Web Crawler** - (may be referred to as web scraper or blog scraper) is the tool we wrote which is able to accumulate massive amounts of blogging data from websites that are fed to it. Crawling refers to programmatically following links in web pages and inspecting the resulting content.
7. **Tagging** - Annotating the data to place searchable categories onto each blog in our database

8. **DOM** - Document Object Model refers to the items displayed on the frontend of the application

9. **REST** - Representational State Transfer, this is an architectural style for sending data through HTTP

10. **TF-IDF** - *Term Frequency - Inverse Document Frequency* is a method for determining which ngrams are most significant to a given document

4.3 **Statement of Functionality**

The application created in this project will take all three of the sections named above - web crawling, tagging, and querying - in order to produce one all-encompassing report that the client can use.

The web crawler will scrape data from different websites using the one or many tags that the client gives it. Tags can include things like what trail they're specifically looking for, information that they want to search for like Trail Angels, littering, etc. The crawler will return the most relevant articles for the search. Once we have returned all of the most relevant articles we will sort the articles into an easy-to-read format so that the client can easily skim through the information to see if it is useful. All in all, the main purpose we plan to achieve and that our client wants us to achieve is that we can save her numerous hours trying to search through different websites, blogs, and forums filled with useless information. This application will allow her to receive information from thousands of blog posts in a matter of seconds. The layout we plan to give her will allow her to quickly skim through information to see whether she will want to use an article or not.

4.4 **Scope**

This is a semester-long project, so we set our goals so that they could realistically be completed within that time frame. However we have set some stretch goals for ourselves that we will work on if there is extra time in the semester after we have completed our main goals. I will outline each of our milestones in this section. For a brief overview, see Table 1.

*Milestone 1:*

Early on we just wanted to make sure that our group and the client could agree on a structure for the data that she can easily look through. We started off with about 5 sources she wanted us to use in our web crawler. She wanted us to be able to find at least 5 more at the start so that we had more information to use. Since then we have obtained about 25 more sources and it's
always an ongoing process. She can even add sources to our web crawler fairly easily once we have finished.

**Milestone 2:**

This is our web crawler stage. We implemented the web crawler that will allow her to easily obtain desired data. The web crawler at this point does not accept any tags or parameters, but just looks through websites of blogs and forums that we pass it and takes all the information from them.

**Milestone 3:**

This was just a check-up stage so that our client knows we are on track to be able to finish before the end of the semester. We used this meeting to show her everything we have completed as well as ask a few questions on what we are to do moving forward as well as find out about any potential changes to the scope of the project.

**Milestone 4:**

This is where we implement the sorting phase of our project. We have all the information sorted into an easily-readable structure that our group and client have agreed on.

**Milestone 5:**

This is where we will implement the final portion of our project, tagging. The tagging will just allow our client to look through blogs and forums based on some filters she applies. Ultimately this will increase the amount of time saved by our client because she can search for very specific things and our application will return only the post most relevant to her tags.

**Milestone 6:**

Work on the final report as a group and finalize everything for our client, providing deliverables and completing the project.
4.4.a Scope Table

<table>
<thead>
<tr>
<th>Milestone #</th>
<th>Task</th>
<th>Estimated Completion Date</th>
<th>Actual Completion Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milestone 1</td>
<td>Have structure for types of data. Obtain at least 5 sources.</td>
<td>2/7/18</td>
<td>2/6/18</td>
</tr>
<tr>
<td>Milestone 2</td>
<td>Implement a method for obtaining data</td>
<td>2/21/18</td>
<td>2/18/18</td>
</tr>
<tr>
<td>Milestone 3</td>
<td>Present initial obtained data</td>
<td>3/14/18</td>
<td>3/14/18</td>
</tr>
<tr>
<td>Milestone 4</td>
<td>Have organized data</td>
<td>3/28/18</td>
<td>3/27/18</td>
</tr>
<tr>
<td>Milestone 5</td>
<td>Implement tagging</td>
<td>4/4/18</td>
<td>4/3/18</td>
</tr>
<tr>
<td>Milestone 6</td>
<td>Have completed and organized project report</td>
<td>4/18/18</td>
<td>4/18/18</td>
</tr>
</tbody>
</table>

Table 1: Table of Milestones

5. Requirements

This project will aid a study that looks for examples of trail culture in the United States -- particularly that of the Appalachian Trail, Pacific Crest Trail, and Continental Divide Trail. The request for this project is that students collect blogs and forum posts relating to trails (e.g., by web crawling). Note that trail culture includes all stakeholders of a trail (hikers, trail angels, conservancy organizations, park rangers, etc.). Prior work has led to collection of many tweets, but that effort should be extended.

Project deliverables:

- Large collection of forum and blog posts relating to trails (metadata should be included)
● Labeling of the posts and blogs indicating the relevant trail being discussed (Appalachian Trail, Pacific Crest Trail, Continental Divide Trail, and Other)

● Description of work done so it can be replicated and extended

● Summary of results, with analysis and statistics

Expected impact of the project:

● This project could assist with a Masters student’s thesis, as well as add data to the repository of web pages for the GETAR project http://eventsarchive.org/.

● Could benefit students who are enrolled in Technology on the Trail by allowing them to categorize blog posts that are based on technology using our scraper. Due to our implementation of tagging, our software is capable of extracting tags that could be beneficial during the analysis of the impact that technology has while hiking or outdoors in general.

6. Design

In an attempt to segregate the code as much as possible, we split our project into modules. Those modules include the population module, the tagging module, the API module, and the frontend. Doing so allows our team to work together without much overlap, as well as write code that does exactly one thing for simplicity.

The population module is the section of the project that scrapes data from a list of blogs that is provided. The database is then populated with the data that is collected from scraping. After that is finished, the tagging module then goes through the database records, inspects the contents of each entry, and generates tags that are applicable to that entry.

Because we’ve decided that using web technologies is best suited for this project, we’ve split our logic into frontend and backend logic. The backend API module has several RESTful endpoints that expose some data to be consumed. The frontend module allows the user to interact with the data that was collected and tagged.
7. Implementation

The implementation for each module is built on top of a couple of pre-existing libraries and frameworks.

The **API module** is built with the [Django REST framework](https://www.django-rest-framework.org) available for Python. This provides us a good structure to serve files to our frontend, as well as the user to make requests for specific tags. The API provides rest endpoints that allow searching by tag. These endpoints respond with simple JSON that can be consumed on the frontend.
The API endpoints that are exposed to clients are structured like the following. This is a read only API. The database can only be written to from Django management commands as seen in Figure 2.

```
path('food/') {
    path('tags/') {
        GET
    }
    path('content/{id}/') {
        GET
    }
}
```

Figure 2. Example of Django management commands.

The `food/tags/` and `food/` endpoints take query arguments so that we can query the database dynamically from the frontend. Seen in Figures 3 and 5 are some sample calls that are valid within the API.

```
'/food/tags/?tag=<tag_text>'
```

Figure 3. Example of an API query.

They will query the database for all tags that contain, case insensitive, the tags text that you provide. For example, if you begin typing “hiking boots” it will return every generated tag that contains “hiking boots”. A sample JSON response from this endpoint for a query for hiking boots would look like Figure 4.

```
[ ... 
    { 
        id: 1, 
        name: "hiking boots blisters" 
    }, 
    ... 
]
```

Figure 4. Example of the JSON response returned from the API.

```
'/food/?tags=<tag_text>|<tag2_text>&order=-author'
```

Figure 5. Example 2 of an API query.
The query in Figure 5 has two query arguments, *tags* and *order*. The tags argument is a list of tag names separated by the | symbol. The order argument determines what field to sort by (author, pub_date, title, etc) and whether or not it should be ascending or descending.

The design choice to require the tags in this request to be textual and not ID’s is to allow for better search results. The tags generated and referenced by blog posts might include “hiking boot, hiking boots..., hiking boot blisters,...”. Coalescing similar tags turns out to be a natural language processing problem, so we decided it was sufficient to find all blog posts that are associated with tags that contain any of the tag text sent in the request. Therefore, a request with tag text “hiking boot” would query for “hiking boot, hiking boots, hiking boot blisters...”. This will be talked about more in the tagging implementation section.

After making a successful request, the API will respond with JSON that looks similar to Figure 6.

```json
[
  ...
  {
    title: "Blog Post Title",
    author: "Blog Post Author",
    pub_date: "12/25/2017",
    source: "<source_url>",
    blog: "Blog Title",
    file: "/files/content/<id>/",
  },
  ...
]
```

Figure 6: JSON-response sample

Our original intention was to store archives of blog posts as .warc files. We have since decided that WARC files don’t fit into our ecosystem because this is a browser based solution as it is. So rather than ingesting HTML, translating it to a WARC file, and then translating that back to HTML to be consumed we’ve decided to write the body content of the blog posts to the database. When a blog post archive is requested, we construct the HTML on the fly and serve that to the user. A list of sources of blogs has been provided in the references section if WARC files should be produced at a later time.

The scraper module makes use of the *requests* and *BeautifulSoup4* libraries. As of right now, it is accessible to the user as a Django management command. It takes as input the location of a sitefile which provides URLs to blogs that the scraper should target. For each URL in the sitefile, the scraper archives each article contained within the blog. It accomplishes this by several processes:
1. An object, Blog, representing the blog, wraps the URL and contains logic for iterating over pages of article posts. This allows us to iterate indeterminately and scrape each page of the blog.

2. For each page during the blog iteration, the HTML is pulled using the requests module and processed using BeautifulSoup. Doing this allows us to identify each article link present on the page.

3. After identifying each article on the page, the link to the article is crawled and the article HTML is requested.

4. With the content from the article URL, the parser identifies metadata about the article. This includes author information, publication date, and article title.

5. The parser then archives the article content, and the metadata into an ArticleArchive object.

6. Each article archive is added to a BlogArchive object which represents the entire archive of a single blog. This has a one to one relationship to each URL in the sitefile.

7. A list of BlogArchive objects is returned to the caller of the scraper. (In our case, this information is later written to the database.)

The tagging module relies on the python-rake module in Python. If you’d like to know how this is implemented, you can read more about the RAKE algorithm here. Like the scraper module, this is used as a django management command. It takes nothing as input, but it must be executed after scraping finishes. That is because the tagging module inspects records entered in the database and tags them. The general steps for how this works are as follows:

1. Launch a threadpool with 8 threads. Submit the tagging function and a list of all blogs currently in the database. (We do this in a threadpool for efficiency purposes).

2. With a list of tuples containing a blog post and all its associated tags, generate a set of unique tags from all of the tags that were generated.

3. Insert the set of unique tags into the database with a single query. Doing this with one query is much more efficient than multiple queries.

4. For each tuple of blogs and their associated tags (which are textual), generate a single query which fetches all database entries of tags from the text that we have.

5. With the list of database records, execute another single query that creates a many to many database relation between the blog and the tag.

The problem with the algorithm we used, Rapid Automatic Keyword Extraction, is that similar tags were generated, e.g., Hiking Boots, Hiking Boots…, and Hiking Boot Blisters. Coalescing these tags would prove advantageous for the user but the problem is that it's not a simple
problem to take on. Determining if two tags are similar enough to merge while also making sure not to merge two tags which don’t belong together is a natural language processing problem. It was decided that this was out of the scope of our project.

The frontend module is a simple static HTML file served from the backend to the user. Rather than serving separate assets for JavaScript and HTML, we decided what little JS we would need should be embedded directly into the HTML file to keep things simple. It makes use of jQuery and Bootstrap to do styling, Ajax requests to the API, and DOM updates.

The search box is a Select2 element that, when typed in, makes requests to the API and updates the results with each keystroke. When a tag (or multiple) is selected from the search box, a request is sent to the API to get its associated blog posts. Furthermore, the results may be sorted using two select fields on the page.

8. Testing/Evaluation/Assessment

All testing was done manually with the assistance of some testing scripts. On each iteration, new features were tested to ensure that they worked as intended when incorporated with the rest of the codebase.

Since it is quite easy to test sources, we used a Python script to manually test the compatibility of our 31 sources. As previously mentioned, WordPress and Blogspot usually have a standard way of going through blogs which our scraper looks for, and some of the sources do not follow this standard, so our scraper will not work for them. We collected between 40 and 50 sources, of which the 31 sources listed in the References section were fully compatible with the scraper.

Unfortunately no unit or integration tests were written for this project as it stands right now. If someone were to inherit this project in the future, that would be a good place to start to get an understanding of the code base.
9. Finished Product

9.1 Application Frontend

Figure 7: Application Frontend

When interacting with the app, the user will see a search box where tags will be suggested that match what the user types. Selecting multiple tags will increase the number of search results returned. The search results are shown in a table displaying the post title, author, date, and source, with the title linking to the post itself and the source linking to the blog source online. All of the posts will look very plain as they have been stripped of all styling and personalization in order to be easier to browse and for the client to study. The results table can be ordered by all of the categories and also sorted ascending or descending.

Statistics regarding the final submission to our client are given in Table 2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blog Sources</td>
<td>31</td>
</tr>
<tr>
<td>Blog Posts</td>
<td>3423</td>
</tr>
<tr>
<td>Searchable Tags</td>
<td>87618</td>
</tr>
</tbody>
</table>

Table 2: Submission Statistics

It should be noted that these are just the statistics from the blog sources that were put together for submission. A user can run the application on their own and add blog source URLs and change all of these values.
9.2 Example of a Scraped Blog Post

A blog post that our scraper takes in is shown in Figure 8. Our scraper will parse through the HTML from this page and determine the important information which is broken up and saved into our database (see Table 3). From the information in the database we are able to display the basic parsed data when our application is run, see Figure 9.

Figure 8: Original Blog Post

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>pub_date</th>
<th>source</th>
<th>body</th>
</tr>
</thead>
</table>

Table 3: Blog Data Saved in Database
Just recently, the hiking forums received the disturbing news that the body of a thru hiker "Stonewall" had been found near Cow Camp Gap shelter in Virginia. The authorities deemed it "suspicious" but nothing else has been brought to light yet. And so I was in church today and several came up to me, asking if I knew the hiker (I had been on the trail southbound through in May, though, so did not run into him as he was hiking southbound in July). They then asked how I could ever think of hiking by myself. That it's too dangerous. That life is at risk.

I recall when I stayed with my hosts in New York while hiking the AT southbound last year that they, too, worried. There are predators, nasties out there, etc. And for a few days afterwards, I was really dealing with fear, and that is not good when one is supposed to be out enjoying this wilderness experience.

Figure 9: Scraped Blog Post


First, the user must clone the git repository from https://github.com/kevincianfarini/capstone. In order to run this project, you need to have Python 3 installed. This is very important as any older version of Python will not work. In order to set up the program you must do the following:

First you must open a bash prompt and navigate to the project directory as shown in Figure 10.

```
$ cd capstone
```

Figure 10. BASH command used to navigate into the capstone directory.

The next step is optional but it is recommended to create a virtual environment inside the directory. That is done by using the commands shown in Figure 11:

```
$ python3 -m venv env
$ source env/bin/activate
(env) $
```

Figure 11. BASH command used to create a virtual environment.
The advantage of a virtual environment is that, when activated, the environment will alias your Python command to be Python 3 by default. Also, any packages you install inside the environment will be separated from your main Python installation.

If you choose not to do this the application should still work as long as you are certain you are using the correct version of Python (Python 3). To ensure your version of Python is correct type the following shown in Figure 12:

```
$ python --version
// Python version will print here...
```

Figure 12. BASH command to check the version of Python currently running.

If a version other than 3.x.x prints out then you are probably not running Python 3 you can try running the command in Figure 13:

```
$ python3 --version
```

Figure 13. BASH command to see if Python 3 is installed.

If this works and prints the proper version output then you will need to use the command ‘python3’ in place of ‘python’ for the rest of the commands in this guide. If not, you need to install Python 3.

The rest of this manual will assume that the virtual environment is in use. After you have started the virtual environment (or chose not to) and verified your python version you should run the following commands shown in Figure 14 to prep and run the application.

```
(env) $ pip install -r requirements.txt
(env) $ python manage.py migrate
(env) $ python manage.py runserver
```

Figure 14. BASH commands to install project requirements and start the server.

If the server starts without errors, you’ve been set up correctly. Hit Control-C to kill the server.
Run the command shown in Figure 15 next:

```bash
(env) $ python manage.py populate sites.txt
```

Figure 15. BASH command to scrape all the blog posts.

This step will take a long time. Figure 15 shows the command used to grab all of the blog posts from the internet, which is the bottleneck. After that's completed, you can begin tagging the data by running the command shown in Figure 16.

```bash
(env) $ python manage.py tag
```

Figure 16. BASH command to tag all the gathered blog posts.

After all of the data population and tagging is completed, you can rerun the server, shown in Figure 17, and access the search results by visiting localhost:8000 in your browser.

```bash
(env) $ python manage.py runserver 0.0.0.0:8000
```

Figure 17. BASH command to run the server.

To export all of the blog data to CSV format, close the server by pressing ctrl + c and run what is shown in Figure 18 in the command line.

```bash
(env) $ python manage.py export
```

Figure 18. BASH command to export the data to CSV.

Figure 18 processing will produce a CSV file saved in the project directory containing all of the blog posts. As opposed to a standard CSV where the values are separated by commas, the values in this file are separated by tab characters, ‘\t’. Contained in the CSV file is all of the metadata about posts that are shown to the user on the frontend. Additionally, the blog content in the CSV is wholly textual. This differs from the blog content stored in the database, which includes HTML tags (and thus might include some images when rendered on the DOM). During the export process, those tags are removed leaving nothing but text.
11. Developer’s Manual

Working with a Django Application

In order to make changes or delve into the application it is highly recommended to have experience or knowledge with the Django Python framework and RESTful web development. If you are not familiar with either of these then it would be helpful to review this Django tutorial, as well as the documentation for the Django Rest Framework and the BeautifulSoup Python libraries.

Getting Blog Posts

In order to save blog posts the database must be unmodified. To get it to this status you must delete the db.sqlite3 database file. After deleting the database file you should run the command shown in Figure 19 to migrate the data.

```
(env) $ python manage.py migrate
```

Figure 19. BASH command to migrate the database.

This will generate a new db.sqlite3 file that is fresh and unmodified. You will now need a .txt file that contains all of the blog sources you want to use. It should be formatted like the following snippet where the URL is followed by the symbols -> and the source of the site (BlogSpot or WordPress) as shown in Figure 20.

```
https://blogsite1.blogspot.com/ -> blogspot
https://blogsite2.wordpress.com/ -> wordpress
https://blogsite3.blogspot.com/ -> blogspot
```

Figure 20. Blog site file formatting example.

In our examples we have been calling this file “sites.txt”.

12. Lessons Learned

Throughout the course of this project, several blockers became evident that were hard to overcome. In short:

- Web crawling is messy
- Automated text generation is hard
Optimizing database interactions is key to a smooth experience.

We found out pretty early on that web crawling can be messy, and we were only dealing with two classes of sites: BlogSpot and WordPress. Even though one of the crowning features of both of these sites is the user friendly blog customization, a lot of the sites shared some of the same structure. The problem is that the structures were not similar enough to write a crawler that could understand all of the different kinds of customizations a blog user can make.

For example, our crawler always grabbed metadata about both blog posts and blogs. That data was one of the pain points of crawling. Some sites implemented their blog site title with an image, and thus couldn't be scraped. There were about 5 different HTML elements that blog post titles could be contained in (eg. <h1>, <p1>, <h2>, <div>) all with different CSS class names indicating it was the blog post title (eg. site-title, entry-header, title). Furthermore, timestamp data often came in different formats and had their own structure problems similar to the blog post title.

Thus capturing all of that data proved challenging. While we did manage to get most of these cases covered, we weren't able to get all of them. To remedy this, our group decided it was easiest to include some error handling within our crawler to detect these problems. If we detected one of these problems while crawling, we decided just to skip over that individual blog post. During development, we handpicked blogs to provide the crawler with. While searching for blog to choose, if there existed problems extracting metadata about the blog itself, we opted not to include that blog in favor of another that showed no signs of the complications mentioned above.

When it came to actually scraping the web pages, some blogs either had protections or JavaScript that interact poorly with the crawler. Some protections included redirecting the crawler to a captcha screen if a blog was being bombarded at high velocity with requests from the same IP. This was impossible to recover from. Others had features on their blogs where, rather than paginating their blog posts had them requested via AJAX using JavaScript on page scroll. We found no way to overcome this in a timely manner and decided it was best to consider those blogs too much of a tangent from our goal.

When implementing tagging, we found out that automated tag generation has a lot of edge cases that were hard to deal with. TF-IDF didn’t produce results that were suitable for tags in our system. Feeding this algorithm several blog posts about hiking produced no results related to hiking but some of the highest rated results included “gum”, and “dirt”. Thus we abandoned that in favor of RAKE as described in the sections above.

The main problem we ran into while implementing tagging with RAKE was the occurrence of similar tags that should really be combined into a single tag. (eg. hiking boots, hiking boot, hiking boots…) Our team attempted to resolve this using the Python built in sequencematcher. That solution proved to be both error prone as well as extremely slow. On rlogin, running
multithreaded with 40 cores, it took 3 hours to run for 15,000 tags (we now have close to 90,000). Furthermore, it would coalesce tags that shouldn’t have been grouped, like 30 mile hike and 3 mile hike.

This problem is a natural language processing problem. It was decided it was out of scope for this project, so instead we implemented a quick and dirty work around described above in the Implementation section.

Finally, optimizing database calls towards the end of the project really helped make things smooth for populating and tagging. Initially, our team took the naive approach to inserting data into the database. Given a list of blogs that all have their own associated blog posts, do a nested for loop to insert blog posts into the database with their associated blog database entry using the Django ORM. The problem with this was that we were generating a SQL query to insert data into the database for every single blog post that we had living in memory from crawling.

When we stated to amass a large amount of data, this solution became unsuitable. It was taking upwards of 40 minutes to insert data into the database that was already scraped from the internet. A similar problem was evident with tags. Thus the solution to this was to do a single SQL query and bulk insert the data into the database. Doing so sped up the database write times in both tagging and blog post insertion dramatically. The insertion time for generated records dropped from 40 minutes to less than a minute.

13. Future Work

In order to make this application better in the future there are a few main features that can be added: mainly, more sources and a better tagging mechanism or the ability to add new tags (not just select from a currently available list) from the UI.

The first of these three features, an increased source list, is relatively easy to accomplish and can be an ongoing process as our current implementation of the software pulls from a sources list. This means that it is as simple as adding the new sources to the list and running the application to see if any harmful JavaScript will break the system. Towards the goal of adding more sources, another possible addition would be compatibility for other types of blogs. Currently the application supports both Blogspot and WordPress blogs and while this covers a decent percentage of blog currently available, it would be nice for a more robust compatibility.

The second feature mentioned, more detailed and customizable tagging, was not part of the original scope of the project due to time constraints and Abigail’s guidance for the specific tags she wanted for her Master’s Thesis. Due to tagging being expensive in terms of resources because of the need to search each piece of scrapped data separately, this would probably be one of the hardest and most detailed features to implement. We decided to use a tagging
algorithm, RAKE, for our project because creating our own tagging mechanism was far out of the scope of this project and this was the best we could find. Given more time, and a funding for this project, we most likely could have found a better tagging algorithm, seeing that a lot of the high-tier ones required a payment or subscription of some kind. For those who might inherit this project in the future, some of the available ones considered were the Text Analytics API provided by Microsoft and Keyword Extraction API by Marketplace.

14. Acknowledgements

We would like to thank our client, Abigail Bartolome (abijbart@vt.edu), for her insight and knowledge about the Triple Crown Trails and her technical expertise which helped in the completion of this project.

We would like to thank, our professor, Dr. Edward A. Fox (fox@vt.edu), and the GTA’s: Supritha Patil (patil93@vt.edu) and Yilong Jin (e1337@vt.edu), for their knowledge and criticisms throughout the semester. Without their help and feedback this project would not have been possible.

Thanks go to NSF for support by grant IIS-1619028.

15. References


Blog Sources

1. http://blissfulhiking.blogspot.com/
2. https://hikethecdt.blogspot.com/
3. https://hikethecrown.blogspot.com/
4. https://realfamilycamping.blogspot.com/
5. https://travelstrailstriumphs.wordpress.com/
8. https://hikinghokie.wordpress.com/
11. https://sherahikes.wordpress.com/
15. https://kevbosite.wordpress.com/
17. https://terriwalkerblog.wordpress.com/
18. https://cdtkiddo.wordpress.com/
19. https://kgontheat.wordpress.com/
24. https://jupiterhikes.wordpress.com/
29. https://stepintonature.wordpress.com/
31. http://7days6nights.blogspot.com/