CSCI 572 Homework 1 Report

Tasks:

1. Download and configure Nutch to crawl AMD, ADE and ACADIS

a. Identify and make changes to Nutch configuration to deal with politeness.

Out of the box, Nutch is an extremely polite crawler. It obeys robots.txt, it limits the number of threads-per-host to one, and it has a crawl delay of five seconds. (These settings are found in runtime/local/conf/nutch-default.txt) These are good settings to use when crawling the web, although it is in most cases not necessary to wait a full five seconds before fetching another page from a given server. We chose to reduce this delay to one second in order to speed up crawling without abusing our web hosts. (It took us about a week and a half to arrive at this number. We had been crawling with delays of five seconds and three seconds because we did not want to get blocked or banned from the websites). Apart from that, we left the other settings pertaining to politeness unchanged. It is extremely impolite to ignore robots.txt and to ramp up the number of threads per host, so we refrained from doing so.

b. Identify and make changes to Nutch configuration to deal with URL filtering.

We had to make several changes to Nutch's URL filtering in order to crawl these websites effectively. Nutch uses a URL filter plugin which filters URLs based on regular expressions. For example, it blocks all URLs beginning with ftp, file, and mailto. For our purposes, we were trying to obtain all types of files, especially data, so we enabled ftp and file URLs (we allowed mailto URLs to remain blocked). Along the same lines, Nutch blocks several different file types such as GIFs, Javascript, CSS, etc. We allowed all of these file types to be crawled. In addition, Nutch blocks certain special characters such as the question mark (?), which appears in the URL of the first seed, AMD. So in our very first test crawl, we learned the hard way that we needed to enable these special characters to be appear in URLs.

It took a bit more work to get the domain-based URL filtering right. We ran a few experimental crawls of the AMD website as well as the ADE website, in which all domains were allowed, in order to gather a list of commonly found domains on these sites. The reason for this is that both of these sites link to other websites, so we want to allow some of these other websites in order to get a good amount of polar data. AMD is essentially a directory which links to many other data repositories. ADE has its own data but links to other sites as well. On the NSIDC homepage where the ADE website is based, there are eight links to other websites (not including ACADIS) containing polar data. Once we ran these experimental crawls, we dumped the crawl data and made a short list of the most commonly occurring domains which seemed relevant. This list appears in our regex-urlfilter.txt files (found in runtime/local/conf).

c. Create team identification information and label your Nutch bot via Nutch configuration.

Agent names used: NutchCrawler999, AVY

2. Perform crawls of AMD, ADE and ACADIS

a. Develop a program or script to iterate through your Nutch crawl data and classify what MIME types you encounter.

We wrote a python script **mimetypes.py** which takes a csv file as input. This csv file is obtained by calling readdb <input_directory> -dump <output_directory> -format csv.

b. Identify at least 15 different MIME types that you encounter while crawling.

We have attached a file **mimetypes.txt** which contains 15 different MIME types which were encountered during all the crawls.

c. Deliver a list of at least 100 URLs that you have difficulty fetching and identify why (e.g., protocol issue, behind a web form, requires Ajax, etc.).

We wrote a python script **urlmetadata.py** which takes a csv file as input and gives a text file containing unfetched urls. File attached **unfetched_urls.pdf**, contains the URLs which we had difficulty fetching and **unfetched_urls_reasons.pdf** contains proper reasoning.

d. Deliver the crawl statistics from your crawls of each repository.

File attached (**CrawlStats.txt**). It contains crawl statistics of each repository. For your convenience we have included a piechart (**piechart.pdf)** for all the crawl statistics

3. Perform crawls with the enhanced Tika and Nutch Selenium

After Crawl 1 we built the latest version of Tika and installed Tesseract and GDAL successfully, followed by upgrading Tika in Nutch-Trunk. It took us a while to figure out how to build Nutch-Selenium plugin and integrate it into Nutch-Trunk as the NUTCH_SELENIUM patch which was provided was for an older version of Nutch. So we had a hard time figuring out how to make the changes provided in the patch file.

After integrating Selenium with Nutch, we ran a crawl with a depth of 2 on all the seeds and we were able to confirm that we had integrated it successfully. We started with the crawl of seed 1. But by the time we started the actual crawl the web sites started to block our crawler. So we only managed to get some data from seed 1.

We ran scripts to check how much more data was fetched to by enhanced Tika and Selenium .We got 676 urls (File attached: **previously_unfetched_urls.pdf**) that were not fetched without using Selenium. So that proves that Selenium and tika did enhance the performance. After that we were not allowed to crawl and we could not explore more features of Selenium.

4. Deduplication Algorithms

a. Exact Duplicates:

Initially, we tried computing hash of just the first few characters of each document and stored it in a hash file. If the hash computed of the document existed in the hash file then it meant the files were duplicates and we skipped it, else we stored the new hash generated in the hash file. This algorithm was very fast, but it was inefficient since it wrongly classified the files as duplicates because most files had a similar first few characters.

Then we tried computing the hash of the entire document instead of just the first few characters. This performed quite well compared to the previous algorithm as we got the files that were exact duplicates.

Finally we improved the algorithm by dividing the documents into N parts and picking the first k characters from each of the part. Then we computed the hash based on the characters picked. This avoids the problem of a common prefix for all or most documents and we need not examine entire documents with other document.

However, we were not able to get all the duplicates because of the presence of spaces, tabs and other small character differences.

b. Near Duplicates:

The near-duplicate detection algorithm we chose to implement was MinHash with Jaccard similarity. We chose this algorithm because of its clear description and intuitiveness.

Here is an outline of the algorithm as we have implemented it. For each document in a given set of documents, we compute it's w-shingling (in our case, we used w=4). We apply a hash function to each of these shingles, and store the 200 smallest values as the fingerprint for that document. For a document D, we'll refer to this fingerprint as minhash_200(D). Note that if a document contains less than 200 shingles, we take all of the shingle-hashes as its minhash_200.

In order to compute the Jaccard similarity between two documents A and B, we first compute the set X, which contains the 200 smallest values in the intersection of minhash_200(A) and minhash_200(B). Next, we compute the set Y, which contains the intersection of X, minhash_200(A) and minhash_200(B). The Jaccard similarity is approximated by taking |Y| / k where k = min(|minhash_200(A)|, |minhash_200(B)|). (Often, k = 200, but there are cases in which we have short documents, and in such cases k can be less than 200.) If this value exceeds a certain threshold, we say that documents A and B are duplicates. We empirically chose .9 as the Jaccard similarity threshold.

Our implementation is a slight variation of the traditional MinHash algorithm. In the original algorithm, a certain number (let's say 200) different hash functions are applied to the shingles of a given document, and then the minimum-valued shingle for each of those 200 hash functions is used to create the fingerprint for the document. We instead used a common variation of MinHash in which only one hash function is applied to each document. To generate the fingerprint, we take the 200 minimum-values produced by that hash function. This helps computationally by allowing us to avoid computing multiple hash functions. Because of this choice, the way we approximate the Jaccard similarity is slightly different than the way it is done with traditional MinHashing.

For our deduplication, we eliminated empty files from consideration. Even though empty files are theoretically duplicates of each other, in practice we do not feel it is correct to filter these URLs.

There are two areas in which our near-deduplication algorithm can be improved. The first has to do with the hashing scheme. We chose to use Java's built-in hashCode() method for Strings in order to hash our shingles. This has served us well, but it likely does not scale to larger datasets and it might be prone to collisions. This is because hashCode() returns a 32-bit integer. In the literature, a 64-bit hash is often used. We thought 32-bit integers would suffice for our project though, as we have tens of thousands of URLs. The second and more important way that our algorithm can improve has to do with its computational complexity. It compares the pairwise similarity for all of the documents in the dataset, with a little bit of optimization. Even with our slight optimizations, however it still runs on the order of n^2 time. This worked fine for us in testing; it was even able to compute near duplicates for our first ACADIS crawl (which had ~6000 URLs) within ~4 minutes. However, it ended up taking a few hours to run for the larger datasets, where we crawled ~30000 URLs. In some of the literature we have read, they mention some techniques that can be used to greatly reduce the number of comparisons that need to be done.

5. URLFilter Plugins for Nutch

We have created 2 URLFilter plugins, one for each Exact Duplicates and Near Duplicates.

The Plugins read from files (ExactDuplicates.txt and NearDuplicates.txt) and check the incoming url and if it matches the urls present in these files, it rejects the incoming url as being a duplicate (either exact or near duplicate) and it is not fetched.

Extra Credit:

On running Apache Tika Jaccard Based Image Similarity (with threshold value set to 0.01) on Crawl 1 data we got the following results.
Seed 1 has total of 1567 jpg files. 4 clusters were formed.
Seed 2 has total of 3244 jpg files. 2 clusters were formed.
After analyzing the JSON output, similarity-scores and the JPEG images closely, we found one cluster with maximum number of images because they had similar metadata.
The snapshots of the output (after visualizing the JSON output using D3) have been attached along with the similarity-scores and JSON cluster files (inside d3-data folder).

Opened an issue at the Nutch Jira instance:
(By Trevor Claude Lewis)

(Link: https://issues.apache.org/jira/browse/NUTCH-1953)

Title: Integrate uBlock into Nutch to block Ads.
Description: I feel µBlock (https://github.com/gorhill/uBlock) or any other related plugin which can block Ads can be integrated into Nutch that way we can skip the Ads in the Nutch crawls.

Questions:

1. Why do you think there were duplicates?

- URL parameters used for tracking and sorting
- Order of parameters which render the exact same result
- HTTP & HTTPS version of the same website

2. Were they easy to detect?

We were able to detect the duplicates using algorithms we designed for exact duplicates and near duplicates.

It was relatively easy to detect duplicates. However, it is more difficult to know if we are getting good recall -- that is, if we are finding most of the duplicates in the dataset. Also, sometimes there are many found duplicates, and in such cases it is also difficult to know if we are getting good precision, i.e. if most of our reported duplicates are truly duplicates.

The table relow shows our deduplication results across our crawls. We'll talk more about these results in question 3 below.

	AMD Crawl 1	ADE Crawl 1	ACADIS Crawl 1	AMD Crawl 2
Exact Duplicates	6437	3020	5	155
Near Duplicates	9058	2533	695	297
Total URLs	30638	31408	2739	3253

3. Describe your algorithms for deduplication. How did you arrive at it? What worked about it? What didn't?

A description of our deduplication algorithms as well as how we arrived at them can be found above in section 4.

Here we'll discussed what worked and what didn't. We performed preliminary testing on the relatively small set of URLs in our ACADIS Crawl 1 and AMD Crawl 2. We were happy with our results both in terms of the number of exact and near duplicates returned as well as the runtime of our programs. From there we moved onto our larger crawl datasets. We were unpleasantly surprised to see that our program slows down quite a bit with a large increase in URLs. It is not *that* surprising though, because as we mentioned in section 4, the near duplicate algorithm is O(n^2). It took about 3-4 hours to run on our 30,000-URL datasets. Further, we were disheartened to see so many exact- and near-duplicates returned for the AMD dataset. But, after some reflection, we have some ideas as to why. The AMD dataset has

many URLs which are full of very similar GET parameters, and we have a strong suspicion that there are some pages which loop back to pages which are the same or at least very similar. Also, we realized that many pages can share a lot of style and layout information, and so our threshold of 0.9 Jaccard similarity might not be high enough to capture this for certain websites. It is possible that AMD just has a lot of duplicates. Even if that is not the case, we feel our results for the other crawls are appropriate.

4. Describe the URLs that worked and why?

Our near-deduplication algorithm does much better on detecting duplication between files with a lot of text vs those which do not have a lot of text, because we are using shingling. Along the same lines, it performs better for those file types with tika parsers than those which cannot be parsed by tika.

5. What MIME types did you retrieve from these websites?

We found a variety of MIME types across these websites. This includes pdfs, office files, videos, images, html, javascript, xml, and some more interesting files such as Google KML files. A comprehensive list of crawled MIME types can be found in our submission directory. We have a list of 15 MIME types as well as three files which contains MIME types encountered for each of the three seeds.

6. Were they mostly web pages? Did you get science data (e.g., HDF and NetCDF and Grib files?) If not, why?

We got a lot of html pages along with lots of media data such as png, mpeg, jpg.
Also we noticed that we got a lot of pdf files. We didn't get science data such as HDF or NETCDF. This is most likely due to these files being hidden behind AJAX forms. For example, we tried manually visiting some ftp files that we were unable to fetch and found that some of them required you to fill in a username and password OR simply click a button saying that you want guest access. But we did get some Google Earth files (such as kml and kmz).

7. Did the Selenium plugin help?

We were not able to crawl much with Selenium as the blocking came into effect. Selenium was helpful as it allowed us to interact with forms and follow links which required javascript for rendering. It was informative as we came to know what URL was being fetched and why some failed. Some issues why the urls failed was because it took long to load the page, some were simply robots denied.

8. Did you get more data after installing the Tika updates and recrawling?
After updating Tika and Selenium, we ran the crawl on test round of depth 3 for seed 1. We got more URLs for this than we had for the same config without Selenium.

9. Do you think you achieved good coverage of the 3 repositories?

No. From the information that the Professor provided us about the data contained in the 3 repositories, we think that we did not achieve complete coverage of the three repositories. The Professor mentioned that to get a good idea about the data we need about 200,000 urls and about the 20-30 gigs of data. We managed to get around 30,000 urls for each seed and around 4-5 gigs of data. The reason behind this is not being to fully deploy Selenium plugin and crawl with that.

10. Also include your thoughts about Apache Nutch and Apache Tika – what was easy about using them? What wasn't?

I think Apache Nutch is great open source tool for crawling websites. It is a very powerful tool as we can include plugins when we need it and use different ones as the need be.

The hard part about using Apache Nutch is the figuring about the configuration, while keeping in mind the constraints on politeness, domain restrictions and getting the maximum data in the smallest time possible.

The good part about Apache Tika is that it is already integrated in nutch. Tika helps us parse the most commonly find mime types. It also gives the option to add custom mime types. Tika even became powerful with the addition OCR and GDAL. Tika helps us look at the data by parsing it and converting it to files.