CSCI 572: Assignment I Report

Crawling and Deduplication of Weapons Images Using Nutch and Tika

Team number: 33
Team members: Gaurav Shenoy, Mahesh Kumar Lunawat, Pramod Nagarajarao, Presha Thakkar, Karthik Kini

Task 1: Nutch installation and configuration

a. Politeness

We made the following configuration changes in the nutch-site.xml to make the crawling polite as instructed in the assignment description.

1) fetcher.threads.per.queue set to 2
2) fetcher.threads.fetch set to 5
3) fetcher.server.min.delay set to 1 : (As fetcher.threads.per.queue > 1)
4) fetcher.max.crawl set to -1, this will ensure that the fetcher will never skip pages and will wait the amount of time required from robots.txt crawl-delay.
5) Named of our nutch bot/agent name: Team33 CSCI572
6) In the agent description, we are mentioning that we are crawling as part of our assignment.
7) We also mentioned our email under agent.email: gvshenoy@usc.edu
8) We also used http.agent.rotate property along with http.agent.rotate.file to specify different agent names in the file
9) In the http.robot.rules.whitelist property we added the seed urls to ensure they get crawled.

b. URL Filtering

1) Regex URL filtering, In conf/regex-urlfilter file following changes were made.
   i) We allowed the files which we required to be fetched
      (gif|GIF|jpg|JPG|png|PNG|ico|ICO|css|CSS|sit|SIT|eps|EPS|wmf|WMF|zip|ZIP|ppt|PPT|mpg|MPG|xls|XLS|gz|GZ|rpm|RPM|tgz|TGZ|mov|MOV|exe|EXE|jpeg|JPEG|bmp|BMP|js|JS)
   ii) Skip URLs with slash-delimited segment that repeats 3+ times, to break loops
      -.*(/[^/]+)/[^/]+\1/[^/]+\1/
   iii) Skip social networking sites. Ex :
      -^(http|https)://([a-z0-9]*\.)*facebook.com/
      -^(http|https)://([a-z0-9]*\.)*instagram.com/
      -^(http|https)://([a-z0-9]*\.)*twitter.com/
      -^(http|https)://([a-z0-9]*\.)*youtube.com/
      -^(http|https)://([a-z0-9]*\.)*apple.com/
      -^(http|https)://([a-z0-9]*\.)*yahoo.com/
2) Suffix URL filtering, conf/suffix-urlfilter.txt file following changes were made.
   We allowed all image types which were blocked.
3) mimetype filtering [conf/mimetype-filter.txt]
   We added image along with html to be fetched, and blocked the rest of the content.
4) adaptive-mimetype filtering [conf/adaptive-mimetypefiler.txt]
   We allowed all image types which were blocked in this file

Task 2: Nutchpy, 100 urls with problem and crawl statistics

a. **Nutchpy to get the different mimetypes**
   We have written a python script, printImageMimeType.py to list different image mimetype obtained from the crawl. We have obtained 10 different image mimetypes and have imageMimeTypes.txt contains the 10 mimetypes.

b. **100 Urls with problem**
   first_crawl.rtf, contains the list of more than 100 urls, the problem with the http response and probable resolution for it.

c. **Crawl statistics**
   Below table lists the brief crawl statistics from the first round of crawling, the detailed statistics are included in the following files crawl_1.csv, crawl_2.csv, crawl_3.csv. We used 3 machines to do our crawling throughout the assignment as this would distribute the workload and make the crawling quicker.

TOTAL urls:	1564150
retry 0:	1544658
min score:	0.0
avg score:	4.6111953E-5
max score:	1.555
status 1 (db_unfetched):	257776
status 2 (db_fetched):	1045840
status 4 (db_redir_temp):	150894
status 5 (db_redir_perm):	8747
CrawlDb statistics:	Done

Table 1: Brief cumulative statistics from first round of crawling

Task 3: Nutch Selenium Protocol

We studied the protocol-interactiveSelenium plugin and added our custom handler CrawlHandler.java to handle multiple cases like behind the web form, AJAX based crawling cases etc. After going through the error logs from the first crawl and manually inspecting the 83 seed urls, the following cases handled by our plugin, they are:

a. Behind the web form: Manually registered to websites and then performed login to website using custom handler, where without login we were not able to crawl the data behind forms.
b. Handled javascript pagination using selenium custom handler and were able to crawl more data through pagination.
c. Handled ajax based interaction like searching guns or weapons data through search field and crawling data specific to weapons.
d. Handled ajax based continuous scrolling with the help of selenium handler and were successful in retrieving the relevant urls.

The CrawlHandler.java is submitted with the report.

Task 4: Install and upgrade Tika and Tesseract

As mentioned in the description we installed and upgraded Tika with tesseract and verified the installation manually by running tesseract and tika command against TIFF image.

Task 5: Re-run your weapon crawls with enhanced Tika and Selenium

a. **Identify at least 100 URLs that you have difficulty fetching and identify why?**
   Please refer second_crawl.rtf which contains the list of 100 URL's, HTTP Error codes and the associated description for the error occurred during the crawl.

b. **Are the URL'S present from 2d still?**
   Some of the URL's are still present, but lot of the URL's like www.arguntrader.com, www.theoutdoorstrader.com which were identified in the first crawl were resolved with the protocol-Interactiveselenium plugin by writing custom handler's. Many parsing errors were fixed by the Integration of updated Tika and Tesseract.

c. **Did the enhanced Tika parsing assist with that?**
   Yes, the enhanced Tika parsing did assist with getting better metadata and it is reflected in the improved statistics obtained in the second crawl. With Tika parser we were able parse and fetch various URL's that weren't able to fetch in step 2d.

d. **Deliver your updated crawl statistics from each repository your crawl.**
   Below table lists the brief crawl statistics from the second round of crawling.

TOTAL urls:	1908239
retry 0:	1891040
min score:	0.0
avg score:	0.05055744
max score:	1.0
status 1 (db_unfetched):	1476487
status 2 (db_fetched):	332612
status 4 (db_redir_temp):	5519
status 5 (db_redir_perm):	63709
CrawlDb statistics:	Done

Table 2: Brief cumulative statistics from second round of crawling

Task 6: Deduplication Algorithms

To get the merged and parsed text from dumps following steps were applied –

1. We first merge the different segments created for the dumps using the following command
```
<path to nutch\runtime\local>\bin\nutch mergesegs merged <path to crawl
folder>\segments\*
```

2. To get the required parse text we run the following command 0
```
<path to nutch\runtime\local>\bin\nutch readseg -dump merged\* result_out -
nocontent -nofetch -nogenerate -noparse
```

a. **Exact Duplicates**
We used the extracted Image's metadata to check if there are duplicates among the crawled images. The MessageDigest class in java provides the functionality of a message digest algorithm like MD5 or SHA. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. We used MD5 in our exact duplicates deduplication algorithm. For each Image metadata we ran the algorithm once by passing the parse metadata [This metadata is generated by the Tesseract from Tika] as content and second time by passing the content metadata and parse metadata to the algorithm. Hash value for the metadata was generated and stored in a HashMap as a key. The value for that key in HashMap is an object which contains the original and the duplicate Image URLs and also a count of duplicate Image URLs. Each time the same hash value is generated, the object value is retrieved and the Image URL is added to it. Also the count of duplicate Image URLs is increased. We observed that when we used parsed metadata we got 14% exact duplicates. When we used the content metadata we got 3% exact duplicates. This was due to we did not have much differentiating metadata being generated in the parse metadata obtained from the updated Tika. Below are the pie charts depicting both the statistics. Our implementation of the algorithm abc.java is uploaded with the assignment.

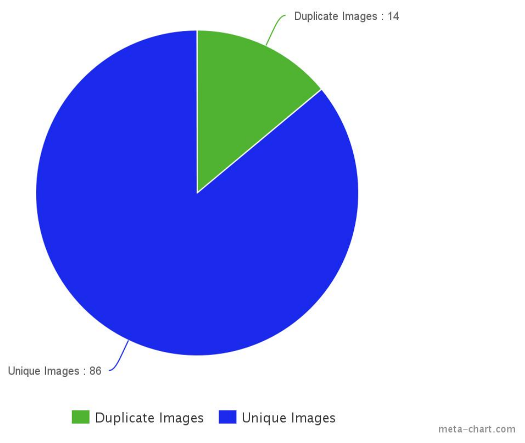

fig1: Pie chart exact duplicates, parse metadata

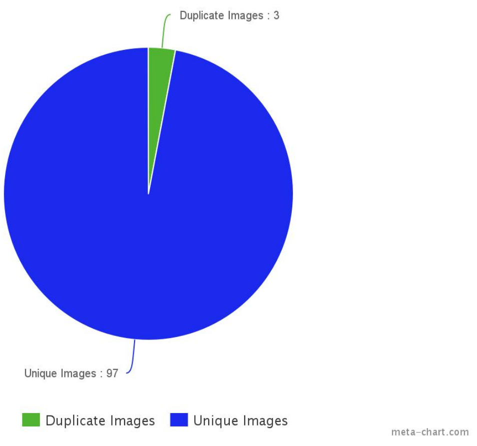

fig 2: Pie chart exact duplicates, content metadata

Algorithm:
```
A. Initialize an two empty HashMap whose 'keys' will be the hash of the URL content and 'values' will be the URL. One HashMap is store Content Metadata and the other for Parse Metadata. Initialize a Group of URL and counter.
B. Read from the Dump file line-by-line.
C. while Dump File != Null
    a. Get URL
    b. if URL = image
```

```
            i.   Fetch Content metadata.
            ii.  Remove nutch defined properties (Ex: nutch.fetch.time) from Content Metadata.
            iii. Create MessageDigest of Content Metadata.
            iv.  Take Parse Metadata and create its Message Digest.
            v.   Compare Content Metadata and Parse Metadata Message Digest values with
                 corresponding Hash Map values.
            vi.  if match found
                     1. Make group of URL's of those duplicates.
                     2. Increment duplicate counter.
            vii. end if
        c. end if
D. endwhile
```

b. Near Duplicates

We have implemented the algorithm based on shingles and Jaccard Similarity in Java. We used the extracted Image's metadata to check if there are duplicates among the crawled images. For each Image metadata we ran the algorithm against the parse metadata [This metadata is generated by the Tesseract from Tika]. Metadata is used to determine jaccard similarity between two Image URLs and if it's above 0.9 then the URLs are considered to be near duplicates. Metadata is tokenized based on blank space as splitter and shingles of length k words are created from tokens. Shingles are hashed using SHA-265 in Java MessageDigest. Since number of shingles will be very large, we select k for shingles, which is computed based on number of tokens in parsetext, and only those hashes which are 0 on mod k are selected. Finally, intersection and union of two sets of hashed shingles are computed using addAll and retainAll functions in Java ArrayList. Jaccard similarity can now be computed using size of intersection and union sets. The output sets of similar Image URLs are grouped and exported in a text file (results_near.txt). We observed that when we used parsed metadata we got 22.34% near duplicates. As said earlier this was due to the fact that we did not have much differentiating metadata being generated in the parse metadata obtained from the updated Tika (And a few images authored by the same site have similar metadata). Below is the pie charts depicting the statistics.

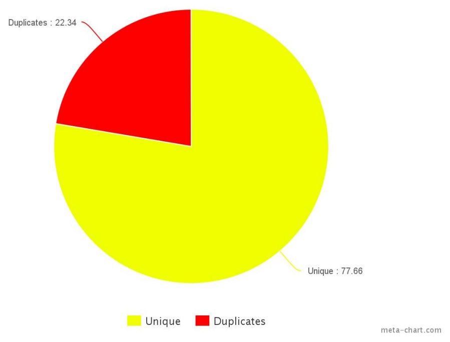

fig3: Pie chart near duplicates, parse metadata

Algorithm:

```
A. Initialize the class FetchedRecord. Store recno, url, parsetext, hashstring and similarurls in an Arraylist.
B. Read from the Dump file line-by-line.
C. while Dump File != Null
        a. Get URL
        b. if URL = image
            i.   Fetch Parse Metadata
        ii.   Call computeShingleHash method
                1. The method takes the text of the file and convert it to its tokens.
                2. For each token, a MD5 hash value is calculated and is stored in a string.
        iii.  Call findSimilarity method
                1. For each record calculate intersect and union of shingles.
                2. Calculate Jaccard Similarity (Jaccard Similarity = intersect/union)
                3. If Jaccard Similarity > 0.9 (Threshold value)
                        01. Similar URLs are found
                4. endif
                5. if similar URL size > 2
                        01. add to the group
                        02. Print the group
                6. endif
    c. endif
D. endwhile
```

Task 7: Nutch Similarity scoring filter focused crawling

a. **Similarity scoring filter**
We studied the nutch similarity scoring filter and understood its working using the cosine similarity algorithm. We then came up with the goldstandard.txt and stopwords.txt files to be included in our configuration directory. We included all the words that were related to guns (we verified against the past crawls data) in the goldstandard file. After making changes in nutch-site.xml, we did the crawl.
We analyzed the results that we got from the nutch default OPIC scoring filter and similarity scoring filter. The results which we got from the similarity scoring filter were up to a higher depth and also we got more images in data from lesser number of crawls related to guns.
(Suggested Change : OPIC has a function which can take only the top-scored URLS and generate crawldb with the same. A similar kind of functionality can be added Similarity scoring to make its focussed behaviour more prominent.
Also, We found that the documentation on how URLs were scored doesn't have proper documentation (especially on how children are scored based on their parent), this would be really helpful if provided, and would increase the understanding of the cosine similarity algorithm which the method uses.)

b. **Compare the resultant URLs from using this Filter to your results from Step 6, were your deduplication algorithms more effective? which ones (6a or 6b) and why ?**
From enabling the similarity scoring filter we were able to crawl and obtain images faster. Hence when we crawled for the same number of cycles(25) once with the similarity scoring enabled and once without it. We observed that we got significantly higher number of images in the earlier one. Our deduplication algorithm were effective with the content metadata, ideally we would have liked to get higher amount of metadata to compute the duplicate images. After comparing the two algorithms we observed that the exact duplicates with the content metadata used as content for our hasing function was much more effective in determining actual duplicate images compare with the near duplicates algorithm. This was due as mentioned earlier lack of quality metadata. Which resulted in lot of images being classified as near duplicates even though they were different. We experimented with the different level of threshold but found that 0.9 was the optimal value.

c. **Updated crawl statistics using the similarity scoring filter.**
Below table lists the brief crawl statistics from the third and last round of crawling, this was the longest crawling done by the team with all the plugins enabled.

TOTAL urls:	18260640
min score:	0.0
avg score:	0.032335192
max score:	1.0
status 1 (db_unfetched):	2914280
status 2 (db_fetched):	624214
status 4 (db_redir_temp):	10998
status 5 (db_redir_perm):	69781
CrawlDb statistics:	Done

Table 3: Brief cumulative statistics from third round of crawling

Other questions related to the assignment mentioned in description:

a. **Why do you think there were duplicates? Were they easy to detect? Describe your algorithms for deduplication. How did you arrive at it? what worked about it? what didn't ?**
We got a higher number of duplicates then we expected. Most of it was because the metadata of many images was same. A few images did not have a lot of metadata even after parsing by Tika, which added to the number of duplicates.
Other then that, a couple of sites had similar images for http and https protocol. A lot of duplicates were also because of similar purposed files : Example the jpg files containing facebook image.
The duplicates of particularly gun images were hard to detect, but genuine duplicates such as common images used across various sites for same purpose were easy to detect.
We have described our deduplication algorithms above, where in we have mostly used Parse Metadata and Content Metadata.
We tried various approaches before like trying deduplication algorithm on Parsed Text (which worked good with content, but didn't work that well with images), using all of parsed text, parse metadata and content metadata (which received very less duplicates almost 0%, as no two pages were completely similar) and trying only parse metadata or only content metadata, but nothing seemed work better than using content metadata and parse algorithm along with each other. The decision of removing some nutch generate data from content data was that it had specific properties might not match because of nutch, and even if the images are actually similar, we might not get them as duplicates.
We came to a conclusion to use Hashing algorithms for exact duplicates as the their metadata will be the same as well. Whereas, for near duplicates jaccard similarity seemed to be a good alternative as it also looks for place (index) of the text along with text (which helps to reduce the scope of it being a duplicate).

b. **Describe the URLs that worked and why?**
Some of the URLs produced a lot of images when we crawled they were, slickguns.com guntraders.com etc. This is due to the sites did have lot of content (mostly product description) with the images and also these sites.

c. **Was there a particular correlation between gun type and website?**
Yes, in some case we got particular types of gun from a given website. For example www.hipointfirearmsforums.com had a lot of Hi-point guns (especially hi-point pistols and carbines)

images. While we even got sites where guns were for specific purpose for example : www.billingsthriftynickel.com has most of the guns used for hunting.

d. **Were duplicates indicative of sloppy selling, or simply dealers promulgating the product to multiple sites?**
There was sloppy selling, but it wasn't that high. Most of the images and text suggest that the exact same descriptions of an advertisement were hard to fine.

e. **Are there any weapons that aren't necessarily firearms, but more dangerous devices (e.g., explosives)?**
Yes, in many of the websites we got other weapons like knife, bazooka, etc. We found different ammunitions and tools that can be used along with guns like cartridge, bullets, sniper scope, cases, magazines, different optics and sights related to guns etc. Also another interesting observation was in some of the websites like gundeals.com which had lot of hidden advertisement and events related to escort services.

f. **What relationships do the clusters obtained from your deduplication algorithms tell you? Are they simply related to the ways that the pictures are edited, or indicative of anything more?**
Many are indicative to the way how pictures are edited. Some indicate that the same seller has provided made an ad on different sites. Some symbol images which are used very frequently for CSS purposes are similar too, which might indicate that the images are used for the same purpose.

g. **Also include your thoughts about Apache Nutch and Apache Tika. what was easy about using them? What wasn't?**
Apache Nutch appears to be a very useful tool for crawling. The installation and getting started with the tool was very easy and intuitive. Apache Nutch is also very useful in parsing the fetched data. One of the improvement probably would be to have better documentation regarding its usage and how we can fine tune its properties for optimal performance.
Apache Tika was easy to install but after integration we noticed that some of the images were not parsed as well as others.