We spent one week or so to download all the files in full dump, and a portion of Common Crawl dataset. We noticed that there are some duplicate records in certain folders in Common Crawl, and we reported the issue by email to the professor.
The Byte-based fingerprint algorithms BFA, BFCC, FHT rely on the content of file rather than metadata which is packed in CBOR, and since we didn't know how much time it would take for us to finish downloading CBOR, we decided to work on raw data from full dump dataset. Common Crawl dataset was probably post processed after downloading by people who previously worked on it. Chances are we can get more accurate records in the full dump. But the tradeoff that we also noticed in the end is that some of the files are missing in the full dump, but are actually there in Common Crawl.
The format of the CBOR data is pretty useful, when it comes to direct access to the probable content type from the HTTP response field in the CBOR files. However, if we wanted to know if an original file is empty, going through all the CBOR files is probably not a very good idea since we'd have to read them into memory and use proper libraries to load them in JSON, which definitely takes more time than reading the metadata of the file to tell if the length of it is zero.

We used Tika 1.11 as per assignment guideline and faced an issue https://issues.apache.org/jira/browse/TIKA-1856 , hence we downloaded tika source with issue fixed, built tika and started using Tika version 1.13. The new version was advantageous as it helped to identify some of scientific data file MIME types within the application/octet-stream files.
We focused more on identifying new magic bytes by analyzing the results of FHT supported with BFA and BFCC. We were able to identify approx**. 100 + 50*4** (xml version header) new magic bytes across **20 MIME types**.
An excerpt from thesis- An Algorithm for Content-Based Automated File Type Recognition, Mason B McDaniel.
The overall difference in accuracy between All Options and only Option 3 was negligible. When Option 3 was wrong, the overall guess was normally wrong, even in cases where Option 1 or 2 provided the correct type. For example, when Option 3 was only 25% accurate across DOC, PPT, and XLS files, Option 2 was 75% accurate. However, when the results were combined, more weight was placed on Option 3, resulting in the misidentification of most of the files.
Another example of this is the first MP3 file used for accuracy testing. This file was incorrectly identified by Option 3 as an RM file. Option 1 correctly identified it as an MP3 file, however more weight was placed on Option 3 resulting in the overall misidentification of the file as an RM file. This suggests that improvements could be made in the assurance level equations for use in combining the scores.
We planned to use a variant of assurance level equation for combining scores from multiple options
Our approach was to generate 2 overall scores:

a) combining scores weighted by assurance values of BFA and BFCC (Option 1 & 2)
b) scores weighted by assurance values of FHT (Option 3)

Use max of 2 overall scores, to determine actual overall score
But since we lacked time we could not experiment our approach.

BFA runs pretty quickly since it simply calculates the frequency of bytes in a file. But it might consume more memory in some cases, especially when a file that we worked on is very large as some video files. FHT runs pretty efficiently, since it doesn't need to read the whole content of the file to memory, only the header and trailer parts are loaded into memory, so it only took around one minute to finish FHT algorithm.

D3 was really helpful since it's more straightforward for us to take a look at those visualization of byte analyses and recognize the difference or improvement of our work than comparing raw JSON results. Outputs from different algorithms could be difficult to discern if there were no visualization.

Since we didn't know how much time the three algorithms would need to run, especially the second is recognized really slow, we started picking up types that are in the range from 1,000 files to 9,999 files. We didn't pick up the types that don't have a lot of files, since too few sample files might cause considerable inaccuracy.

The types that we chose are as followings:

application_atom+xml	application_x-tika-msoffice
application_dif+xml	application_xml
application_gzip	application_zip
application_rdf+xml	image_vnd.microsoft.icon
application_rss+xml	text_plain
application_vnd.ms-excel	text_x-matlab
application_x-sh	video_quicktime

1. Why Tika's detector was unable to discern the MIME types?

Some of the files are 0KB, which means there is nothing that Tika's detector could discern. Some files may be truncated, so that only a part of the file is there.

2. Was it lack of byte patterns and specificity in the fingerprint?

We may not have enough samples in the dataset to cover all the cases in the real world.

3. An error in MIME priority precedence?

Since we only had some of the byte patterns that are generated from the algorithm, we only updated those which have a high value.

We didn't notice that all the existing priority numbers in the XML file were integers. Later on, we modified the result of priority, making float numbers to integer numbers.

4. Lack of sensitivity in the ability to specify competing MIME magic priorities and bytes/offsets?

When we assigned the priority of some magic bytes, we didn't know what exactly those existing priority number mean. After checking the majority of the priority value, we decided on putting 50 as the base priority with a coefficient that we produced using our analysis result.

But we only chose those correlation which are greater than or equal to 0.80, and most of the results that we put in the Tika mime type XML file are greater than 0.95.

Tika is really easy to use. As a group, we have two people primarily writing code in Java, and one in Python. Tika ran fast during the process of file classification and generating maps of file name, file type, and file key defined in CommonCrawlData format. We also used Tika and put different arguments in the constructor and found out different ways to invoke Tika methods. And Tika server makes it considerably easy for us to write Python script to use Tika detector through its web services portal.
Another thing that we noticed is that Tika seems not able to take advantage of those magic bytes in the trailer. We'd like to work on Tika to make more powerful by integrating trailer detection with relative offset from the end of the file.
In the meantime, we also encountered some warning issues with it, when we were running Tika Similarity to get Edit Distance similarity and Cosine Distance similarity.

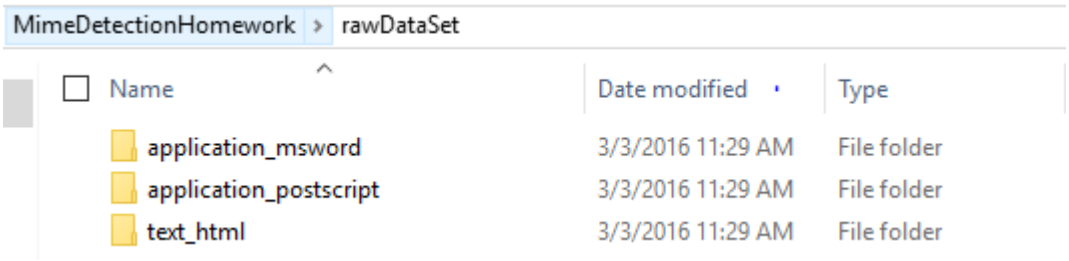
D3 based Pie Chart visualization was developed for the MIME diversity of the TREC DD-Polar dataset using the existing JSON breakdown from GitHub :
(https://github.com/chrismattmann/trec-dd-polar/)
This can be found at: **d3_pieCharts_Polar_Jsons/index.html**

Files from all sub-folders tree were moved to a top level directory using batch script :
for /r %i in (*) do @move "%i" .

```
        1 file(s) moved.
        1 file(s) moved.
        1 file(s) moved.
        ……
        1 file(s) moved.
```

All empty sub-directories were deleted using :
for /d %i in (*) do @rmdir /s /q "%i"

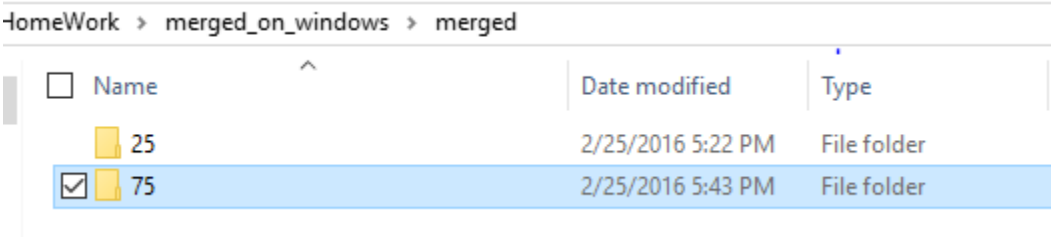
Tika.detect() was leveraged in **fileClassify.java** to classify the files into different folders



Classification can be cross-validated using **fileAnalyse.java**

```
C:\Users\Ravi\Desktop\MimeDetectionHomework>java fileAnalyse
rawDataSet\application_msword
fileType : application/msword :
00E0703641B2FBC5B4FA0E5D7E096801F68C484E3292C4AD063C705D88633FDC
fileType : application/msword :
0B2BBFA70156EF1D43D251AF9A1A39EE63F8A403810F0E98D3753281DAC3B6A9
```

Each of 14 types were further grouped under 75% and 25%.



Json files representing fingerprint for each file type was created:
java **bfa** --create rawDataSet output_dir
Creating file FP for :
C:\Users\Ravi\Desktop\MimeDetectionHomework\output_dir\application_msword.json
Creating file FP for :
C:\Users\Ravi\Desktop\MimeDetectionHomework\output_dir\application_postscript.json
Created Finger Print Json Files

Byte Frequency Analysis was performed on 75% of 14 MIME types, updating fingerprint json
java **bfa** --analyseFreqency rawDataSet output_dir

Byte Frequency Correlation was performed on remaining 25% of 14 MIME types, updating fingerprint json
java **bfa** – updateCorrStrength rawDataSet output_dir

D3 line chart visualization for BFC analysis can be found at: **d3_bfa/index.html**

Byte Frequency Cross Correlation:

We downloaded around 60 GB of CBOR data, but found out that would take too much time to download all the files, as well as we don't have that much of space on our computer to store them.

Java program NamasteNihaoFileClassification is a classification to count how many types there are in our own folders, and it can also create several maps to store file and new content type with our updated mime type xml file, and finally dumps a JSON file to tell how many files each type has.
We also tried to dump all the files in the maps, including duplicate map, type-file map, file-type map, but it took so long time, and after almost 12 hours, dumping around 4 GB data in memory was still incomplete.

The Byte Frequency Cross Correlation result is in the **d3_bfcc** directory. You can find it in visualization.html file.

The Python scripts that we used in this assignment are main.py, utility.py, helper.py.

To accelerate running Python programs, we also used PyPy, a Just-In-Time compiler which compiles Python code in byte code, which provides pretty good speed.

In main.py, there are three options to choose:
1. runs byte frequency analysis
2. runs byte frequency cross correlation
3. runs byte frequency cross correlation on a folder which contains several folders, each containing files of the same type

Utility.py is the file which primarily pre-processes or post-processes on the dataset or generate certain intermediate files that we need.
In utility.py, there are multiple options to choose:
1. outputs all the files a folder contains, including all the subfolders, and the count of the files
2. reads the content of the JSON description file in TREC Polar dataset, and output
3. generates a JSON file in the format that we defined
4. outputs the content type of the HTTP Responded file contained in CBOR in a specific folder (including subfolders)
5. generates move commands in shell for all the files in a specific folder
6. generates a JSON file which contains a map of type and file
7. generates a JSON file which contains the count of each type
8. merges content type JSON file and key file JSON file into one file
9. merges multiple JSON files into one file

In helper.py, there are
1. moves all the files that are in one folder to another directory
2. gets all the magic bytes generated from File Header Trailer algorithm in another Java program, and generates a JSON file containing magic byte objects in it
3. gets all the magic byte objects in the JSON file, and generates mime type xml snippets from magic bytes
4. splits the final count JSON into 2 JSON files, a new one and an old one

FHT:

We implemented FHT (File Header Trailer) algorithm by reference with An Algorithm for Content-Based Automated File Type Recognition (Mason B. McDaniel). The program reads the first 4,8,16 bytes in the header and last 4, 8, 16 bytes in the trailer. We found out most magic bytes in different types of files distributed in first 8 bytes. Since this algorithm runs very fast compared to the other two algorithms, especially the Byte Frequency Cross Correlation, we decide to try more. So we extend the number of bytes to 50 in both header and trailer, trying to find out more useful information. We also generated D3 visualization, which confirms to the distribution in the paper. Actually we use NVD3, which enhance the ability of D3. It shows obvious of the magic bytes by the bigger symbol.

Magic Bytes:

FHT analysis helped us identified magic bytes from header and trailer, (NEW_magic_bytes_all_files.json) we found 3372 magic bytes across different mime-types on all raw data set(not just on chosen 15 types for BFA & BFCC) whose correlation value was above threshold defined 0.8 (2859 bytes of correlation value exactly =1). Of these bytes only alphabets were used to update tika-mimetypes.xml, as updating other number and special characters were not possible as tika-app compiled, but failed to build, as some of the default test cases which where were executed as part of maven install failed.

The mime magic bytes were updated to tika-mimetypes.xml. Updated tika-mimetypes.xml and tika-app jar can be found under magic_bytes directory

Magic byte updates can be found in diff captured in magic_bytes/new_magic_bytes_found.txt

If mime-type had existing magic bytes, new magic bytes were added using same priority level.

If mime-type didn't have any magic bytes, new magic bytes were added with priority="50".

fileCount.py utility was used to generate json file for files being classified :

C:\Users\Ravi\Desktop\MimeDetectionHomework>python fileCount.py rawDataSet > initialClassification.json


```
[
   {    "count": 2,          "mimeType": "application_msword" },
   {    "count": 3,          "mimeType": "text_html"          },
   {    "count": 3,          "mimeType": "application_postscript"   }
]
```


Result of Tika MIME type detection after updating magic bytes can be found in **magic_bytes_comparison/**comparison_output.txt

magic_bytes_comparison/compare.java was used to identify the difference in mime type classification before updating magic_bytes (initialClassification.json) and after updating magic_bytes (finalClassification.json) to generate **comparison.json**

Tika MIME Classification comparison with magic bytes update

pieChart visualization : d3_MimeDiversity_MagicBytes_pieChart/index.html

barChart visualization : d3_MimeDiversity_MagicBytes_barChart/index.html

EXTRA CREDIT (Tika Similarity)

Cosine distance:
To two points, measure the similarity between two vectors (each document is characterized by a vector) by calculating the cosine of the angle between them. The value of each dimension corresponds to the number of times that term appears in the document.

Jaccard Distance:
The size of the intersection divided by the size of the union of the sample sets

Edit Distance:
How dissimilar two objects are.

Files of the same type are clustered together. We ran Jaccard Similarity, Edit Distance Similarity, and Cosine Similarity on 2 types of folders, respectively. Each CSV file generated by Edit Distance or Cosine has the files of the same type. So our observations are that all the elements are in one cluster. From what we see, we couldn't tell the difference between different results from the three algorithms. But we could foresee that if we chose different file types over different folders, the result would be different, and each visualization would have multiple clusters. And the difference between 3 algorithms would be somehow different from what we have.

Cosine distance:
To two points, measure the similarity between two vectors (each document is characterized by a vector) by calculating the cosine of the angle between them. The value of each dimension corresponds to the number of times that term appears in the document.

Jaccard Distance:
The size of the intersection divided by the size of the union of the sample sets

Edit Distance:
How dissimilar two objects are.

Files of the same type are clustered together. We ran Jaccard Similarity, Edit Distance Similarity, and Cosine Similarity on 2 types of folders, respectively. Each CSV file generated by Edit Distance or Cosine has the files of the same type. So our observations are that all the elements are in one cluster. From what we see, we couldn't tell the difference between different results from the three algorithms. But we could foresee that if we chose different file types over different folders, the result would be different, and each visualization would have multiple clusters. And the difference between 3 algorithms would be somehow different from what we have.