

DetectFt -A File Detecting Tool

A Project Report

Submitted by

**Antara Sinha
Aditya Raj Handa**

Under the guidance of

Dr. Ajit Kumar

Abstract

File type detection is a difficult but very important step in keeping data safe. Here we talk about three methods to detect file type, using an extension, magic number and content-based file type detection, and a few algorithms to help do so. Knowing file type can help in filtering files, preventing scams, steganography, retrieving data during hard disk analysis in digital forensics.

Introduction

A file in a computer is a place for recording and storing data, and information can be written in a file. These files can be edited and transferred. Similar types of files have the same format. The file format is the layout of a file that tells us what kind of data is stored and organized in a file so basically file type is the name given to a specific kind of file.

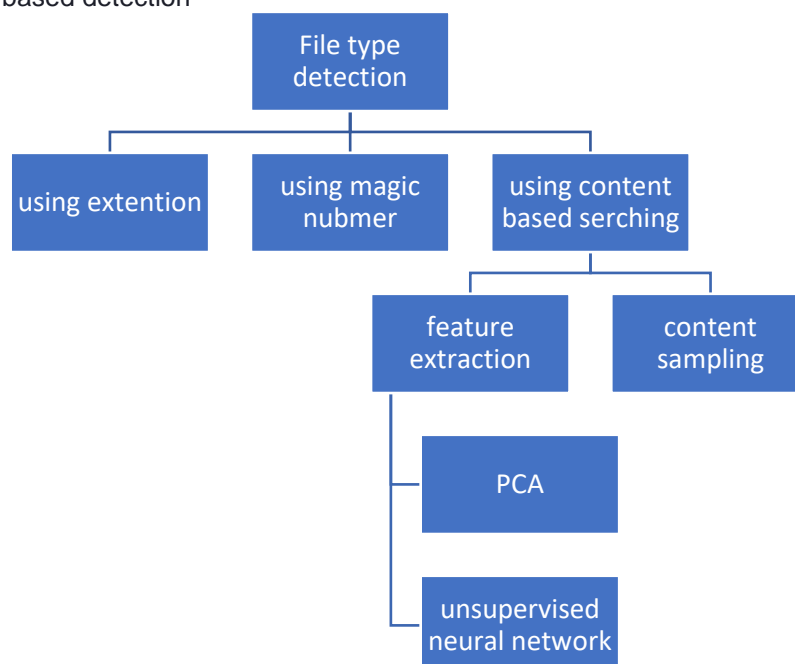
Privacy breaches have become so common nowadays an easy way to overcome this is to use an email attachment filter so this will block certain attachments that may harm our devices so this will help in keeping a track of whatever file enters or leaves our system. It is important to filter attachments as they may have malicious content hidden in them and our device needs to be safeguarded, but emails are not the only way files enter or leave a system thus we need to have a stronger system method that will keep our system safe. Even filtering file types depends on having a knowledge of the true filetype of the file, and safety cannot be achieved without knowing the true filetype.

The identification of a file's type is an important problem in both digital forensics and cybersecurity. Applications rely on protocol or file format specifications to correctly access the header fields and data in files. However, to process an input file, the application must first be able to determine the correct file type.[1]

File type detection has the most usage and importance in the proper functionality of operating systems, firewalls, intrusion detection systems, antiviruses, filters, steganalysis, computer forensics, and applications dealing with the file type classification. Steganography tools can hide messages inside of pictures or other files without changing their appearance. Steganalysis [7], which detects hidden content, majorly depends on detecting the file type first, many companies lose data worth a lot of money due to steganography.

Three methods are using which file type can be detected [1][8]. They are identifying file type by

- file extension,
- magic bytes,
- content-based detection



Detecting file type using File Extension

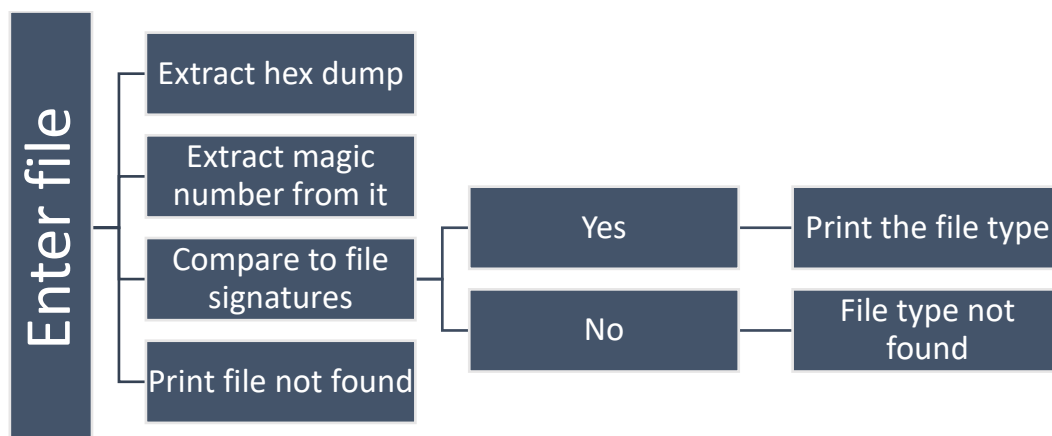
The most common method to detect file type is by looking at the file extension. The extension can easily be spoofed by just renaming the file. To detect a file type by looking at the extension we do not need to open the file, thus making it the fastest way to detect. The speed factor of detecting file type is important due to the volume of files that must pass through these utilities, and makes up for an important reason why this method is preferred. Another reason we look at extension is that it is highly applicable which means most file types have an extension attached to them thus making this applicable to both types of file binary as well as text. [1]

The major problem that we observe while using this method is that extensions can be spoofed and altered very easily. It doesn't take a lot of effort to alter the file extension. Coming to Linux and Unix systems files here do not have extensions there is an option that allows optional extensions of any string regardless of file type, this allows executables and scripts to be hidden. Let's consider an example that we have a file named happy.txt and looking at the extension we can say it's a text file, This script would not even need to have its extension changed, and could be run by typing `./happy.txt`. [1]

Windows being a little more dependent on file extensions, double-clicking on a file name lets consider the same example happy.txt, it should open the file as a text file but it will give us a message file does not exist but if we open the same file using mp3 it will open it as a music file. So when we try and filter files based on file extension can easily be fooled and bypassed. So someone who is trying to get past the filter will change the extension to something that is allowed.

File Type Detection with Magic Bytes

Another method of file type detection uses what are referred to as "magic bytes". This method is only applicable to binary files and is based on matching signatures that vary in length from two to 46 bytes in file headers. Magic bytes are typically the first couple of bytes in a file. There are no standards to what a file type can contain so a creator can always include a unique sequence to identify their file type. [1]



The main problem with magic bytes is that it is only compatible with binary file types. i.e. magic numbers only exist for binary files and not all binary files need to have magic numbers associated with them. This means this method is only applicable to a certain type, thus skipping several file types. So the risk associated with this method is a lot of files especially ASCII based files will be ignored as no magic number is assigned to them. So, for example, ignoring C++ files that leave software development station can be problematic.

Another problem arising with this method is that there is no standard defined way as to how we can define a file type this gets people to define the magic numbers as they wish, and these vary in length and don't give specific answers.

One more problem that we face is that there is no source of magic number that has all the signatures and can be trusted, there are many sites that counter each other's information, that is their magic numbers suffered from each other. Thus, this contradictory information cannot be trusted. this could cause false positives.

-

Detecting file type by searching Content-Based

The final method of file type detection is to consider the file contents and using statistical modeling techniques. This method reveals the malicious file whose contents do not match the file type they claim to be. this looks at the byte values so each byte has 8 bits so 256 different values can be accepted. The Binary File Descriptor(BFD) of a file can be found by reading the contents of that file and keeping a count of a repetitive byte. A file of the same type have the same characteristics, and extracting this can help with file.

Let us consider we take the file print of the file is taken by extracting some features of the file it is then compared to the file print of sample files that can be taken earlier. file print is nothing but the fingerprint of a file. The original principle is to use the BFD of file contents and manipulate them with its statistical features. Such statistical measurements together form a model of the chosen file type, sometimes called a centroid. a single file type can have multiple centroids(multi-centroids). The centroids are then compared to an unknown sample file, and the distance between the sample and the centroids is then calculated. If such distance is lower than a predefined threshold, the examined file is categorized as being of the same file type that the centroid represents.

Under content-based file type detection there are 2 main methods [5]-

- Feature extraction/selection
 - Principle Component Analysis
 - Unsupervised Neural Network
- Content sampling technique

-

Feature Extraction/Selection

Feature selection and Feature extraction is the problem of optimally selecting the statistical features. feature extraction(intersection) works by creates a smaller set of features from linear or nonlinear combinations of the original features, and feature selection(union) chooses a subset of the original features. the new subset in from feature selection has the same dimension as the original one.[2]

This can also be explained as considering a few byte patterns that most frequently occur may be sufficient to represent the file type, so we make a subset of these high-frequency byte patterns as features. Since each file type has a different set of high-frequency byte patterns, we merged the sets of high-frequency byte patterns into a unified set of features for all the file types, which is to be fed into the classifier. To merge them we tried two strategies: union(feature selection) and intersection(feature extraction). The union here combines the feature sets of all the file types, while the intersection extracts the common set of features among the file types. The result of union operation may also include low-frequency byte patterns for certain file types if those patterns occur in other file types frequently. In contrast, the result of intersection operation guarantees that only the high-frequency byte patterns are included, this causes an issue here because if a high-frequency byte pattern does not occur frequently in all the file types it will be omitted.[5]

These methods are used to reduce the number of effective features that represent the data set while retaining most of the intrinsic information. reducing the dimension will result in losing data. The goal of dimensionality reduction is to preserve the relevant information as much as possible

Principle Component Analysis

This is a feature extraction technique. It is a linear transformation of the coordinate in which the data is described. Even a small number of principal components are usually sufficient to make up most of

the structure in the data. PCA tries to preserve as much information possible while finding a lower-dimensional subspace.[5]

Unsupervised Neural Network

Nowadays feature selection is done using neural networks, and this can be done using auto-associative networks. this means that the input and output nodes are equal, so let's consider it is d , and consider a hidden layer of k nodes along with linear activations. Now this network has a unique minimum and the hidden layer which maps the input space onto the output subspace. This procedure is equivalent to the linear Principle Component Analysis[8,9].

There is a Cybenko theorem[10] which states that a three-layer neural network with n input neurons, nonlinear transfer functions in the second layer, and linear transfer functions in the third layer of r neurons can approximate any continuous function from R^d to R^k , with an arbitrary accuracy and exploiting this theorem will give us a much better compression .

And this is only true if the number of neurons in the second layer is sufficiently large. After this, a three-layer MLP, multi-layer perceptron is used to compress.

Now another MLP is used to expand the compressed data and make an approximation of the input since the target of this network is not specified. So now, a five-layer MLP is formed in which the desired outputs are the same as the inputs.

So in the architecture of a feed-forward auto-associative neural network with five layers, the outputs of the third layer are the compressed data, called the nonlinear principal components.

The backpropagation algorithm is used while working with neural networks and to implement this we use two distinct passes of computation exists .the first pass the synaptic weights remain unaltered and the function signals are computed neuron by neuron and the second pass starts with the outer layer bypassing the error signals in layer by layer manner [5].

Content sampling technique

So it is possible that while looking at file content to detect file type the entire content may not be required and we can detect file type by the only looking at some of the content thus this will take a huge amount of time if we search the whole file content. so here we try to sample the content to reduce time in obtaining byte-frequency distribution.

To see the effectiveness of this method, we sampled the initial continuous bytes and sampling a few small blocks in random locations in a file and then we heard these two methods, the first one being obtaining data depending on location this is the fastest but this method may be biased. the second method overcomes this problem by gathering location-independent data but this is slower since the files are sequential access.[2][5]

Algorithm

While designing an algorithm there are certain goals should be kept in mind that will help achieve more effective results. number one being accuracy the algorithm should be a straight answer to our problem and must give precise answers. In our case, we need file prints so our algorithm must generate file print automatically which should be minimum. This process should not cost us a lot of time and most importantly this algorithm should be file size independent .

The three algorithms that will be discussed are [3] [4] -

- Byte frequency analysis (BFA) algorithm
- Byte frequency cross-correlation (BFC) algorithm
- File header/trailer (FHT) algorithm

1. Byte frequency analysis (BFA) algorithm

A file is a collection of bytes, each byte has 8 bits associated with them and can represent 0 to 255, using this information we can create a frequency table by counting the occurrence of each byte value.

many frequency distributions tables of file types are similar which helps the user to get information about unknown file types. While many file types have a very distinct distribution that can be used to differentiate them from other file formats.

The issue with BAF is that it only compares the overall frequency byte, ignoring the other characteristics, observing these characteristics can help give a better result.

1.1 Building the byte frequency distribution

First, we start by counting the number of occurrences of each byte value for a single input file. initially, we start by constructing an array of size 256 and initialize all of them to 0, now For each byte in the file, the corresponding element of the array is incremented by one. after doing this for all elements divide each element is now divided by the number of times the most frequent element occurs, so that we get the frequencies between the range 0 to 1.

In case some files have a higher byte frequency occurrence than others then the normalization will show a higher spike at these values to overcome this issue we could pass the frequency distribution through a companding function to emphasize the lower values. The companding function results in a frequency distribution that is still normalized to 1. This is true since the most frequent byte value was normalized to 1, and the companding function with an input value of 1 results in an output value of 1.

1.2 Combining frequency distributions into a fingerprint

After taking the average result of multiple files we make a fingerprint of common file types. to add new frequency distribution we use a simple averaging equation .A fingerprint is generated by averaging the results of multiple files of a common file type into a single fingerprint file that is representative of the file type as a whole. To add a new file's frequency distribution to a fingerprint we use the following simple averaging equation, where Apart from the byte frequency distributions, the fact that the frequencies of some byte values are very consistent between files of some file types, while other byte values vary widely in frequency can be used for refined comparisons. so now, if a byte value always occurs with a regular frequency for a given file type, then this is an important feature of the file type and is useful in file type identification. After calculating the correlation factor by comparing the frequency for each file we can combine all the correlation factor to form one correlation strength score for each byte value of the frequency.

While calculating the correlation factor we take the difference between that byte value's frequency score from the input file and the frequency score from the fingerprint and If the difference between the two frequency scores is small, then the strength increases toward 1 else decreases towards 0.

Therefore, if a byte value always occurs with the same frequency, the correlation strength should be 1. If the byte value occurs with widely varying frequencies in the input files, then the correlation strength should be nearly 0.

1.3 Comparing a single file to a fingerprint

After the above steps, we compare the unknown file's byte frequency distribution to the byte frequency scores and the associated correlation strengths stored in each file type fingerprint and pick the best match

2 Byte frequency cross-correlation (BFC) algorithm

Consider an example where two characters have an equal or similar frequency, so studying the cross-correlation, between the byte value frequencies can be measured can help strengthen the identification process.

So what BFC does is that it builds the byte frequency cross-correlations of individual files, to construct a file print and compares to get a numeric number

2.1 Building the byte frequency cross-correlation

We need to have two key information while using BFC

- the average difference in frequency between all byte pairs and
- correlation strength similar to the BFA algorithm.

Correlation strength score depends on what the kind of frequency relationship do two files share, byte values that are close will have a higher correlation strength score than ones that are not close.

To characterize the relationships between byte value frequencies we first build a two-dimensional 256×256 cross-correlation with indices ranging from 0 to 255 in each dimension.

Now consider two-byte values i and j and if they are being compared, then array entry (i, j) contains the frequency difference between byte values i and j while array entry (j, i) contains the negative of the corresponding (i, j) location. Since using both the array is of no use we use the lower half of the array to store the correlation strengths of each byte value pair. so now entry (i, j) contains the frequency difference between byte values i and j while array entry (j, i) contains the correlation strength for the byte pair.

$(0,0)$ stores the number of files used to compute fingerprint and to calculate the frequency difference we just subtract the individual frequency score of the byte values.

After normalizing the frequencies with a range 0 to 1 we get a range from -1 to 1 where -1 denotes that the frequency of byte value i was much greater than the frequency of byte value j . A score of 1 indicates that the frequency of byte value i was much less than the frequency of byte value j . A score of 0 indicates that there was no difference between the frequencies of the two-byte values.

2.2 Combining cross-correlations into a fingerprint

Once the frequency differences between all byte- value pairs for an input file have been calculated, the correlation factors can then be combined with the scores already in the fingerprint to form an updated correlation strength score for each byte value pair.

The more the number of files the more accurate is the correlation, If at least one file has been previously added into a fingerprint, then the correlation factor for each byte value pair is calculated by subtracting the pair's frequency difference from the new file and the same pair's average frequency difference from the fingerprint. This results in a new overall difference between the new file and the fingerprint. If this overall difference is very small, then the correlation strength should increase toward 1. If the difference is large, then the correlation strength should decrease toward 0.

We calculate correlation strength using the same equations used in BFA and the Number of Files field is incremented by 1 to indicate the addition of the new file.

2.3 Comparing a single file to a fingerprint

First, we compute a score similar to BFA, As the difference between these values decreases, the score should increase toward 1 else 0. After this, we calculate the assurance level. The higher the assurance level, the more weight can be placed on the score for that fingerprint. On comparing the cross-correlation array to the score and strengths we can pick the best match

3 File header/trailer (FHT) algorithm

The above two methods can identify a great number of files, but some files have patterns that can be difficult to identify. So we analyze file headers and trailers to help identify files .these are patterns that appear at fixed locations that are beginning and end as the name suggests.

3.1 Building the header and trailer profiles

First, we need to decide how many bytes are to be analyzed, after this consider H to be the number of bytes for header files and T to be the number of bytes in the trailer so now we form a 2 X 2-dimensional array of dimensions $H \times 256$ and $T \times 256$ respectively.

All 256 bytes can be scored independently for each byte position based on their occurrence frequency at a corresponding position.

An individual file's header array is initially set to 0. so for each byte position the array corresponding to the value of the byte is filled with correlation strength of 1. An exception occurs in a case where the input file is shorter than the header or trailer length then we fill the missing byte rows with -1. The trailer array is similarly constructed.

3.2 Combining header and trailer Profiles into a Fingerprint

We create a fingerprint by averaging the correlation strength just like we do in BFA and BFC

3.3 Comparing a single file to a fingerprint

Now we compare the files header and trailer information to the cross-correlation scores and correlation strengths and then pick the best and Setting the assurance level equal to the maximum correlation strength allows even a few bytes with very high correlation strength. so a few bytes can make a huge difference in recognizing a file.

Future Work

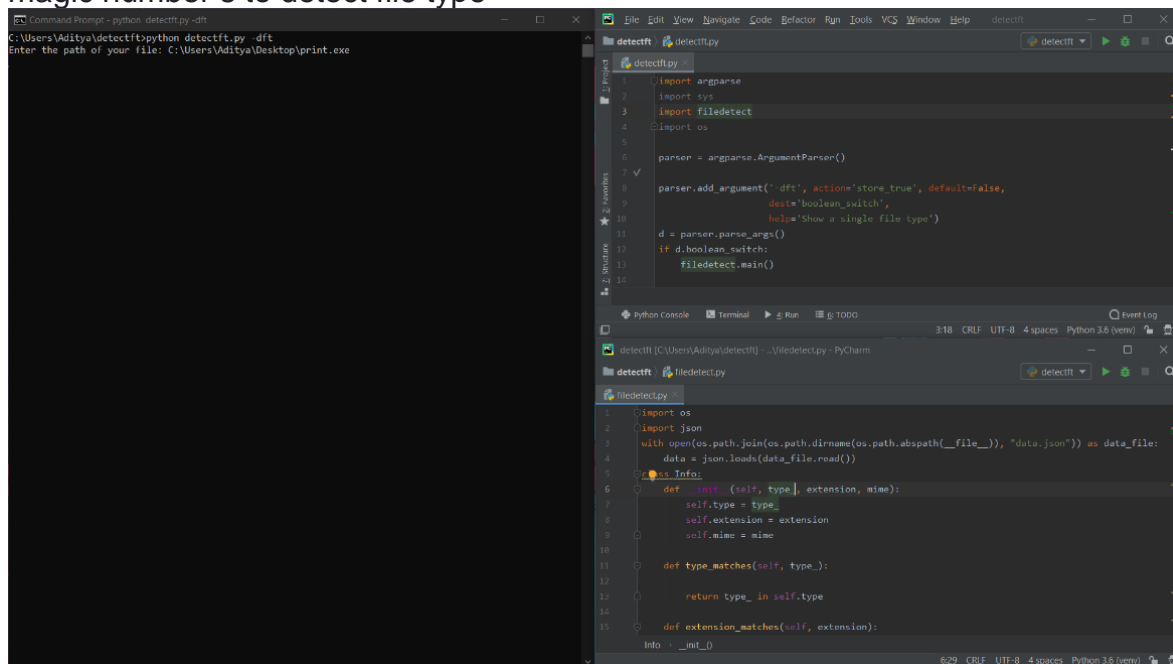
A possible way to improve file type detection process is by using string searches that may be embedded in certain file types and also the use of machine learning. Another method to improve is by making the byte distribution scanning faster.

We can do so by taking samples of a random selection of a file's bytes and verify the profile of the random selection against known signatures. thus increasing the speed This would increase speed while decreasing the certainty of results, but we must take correct samples [1] Again using strings to identify can help indicate file type faster.

Result

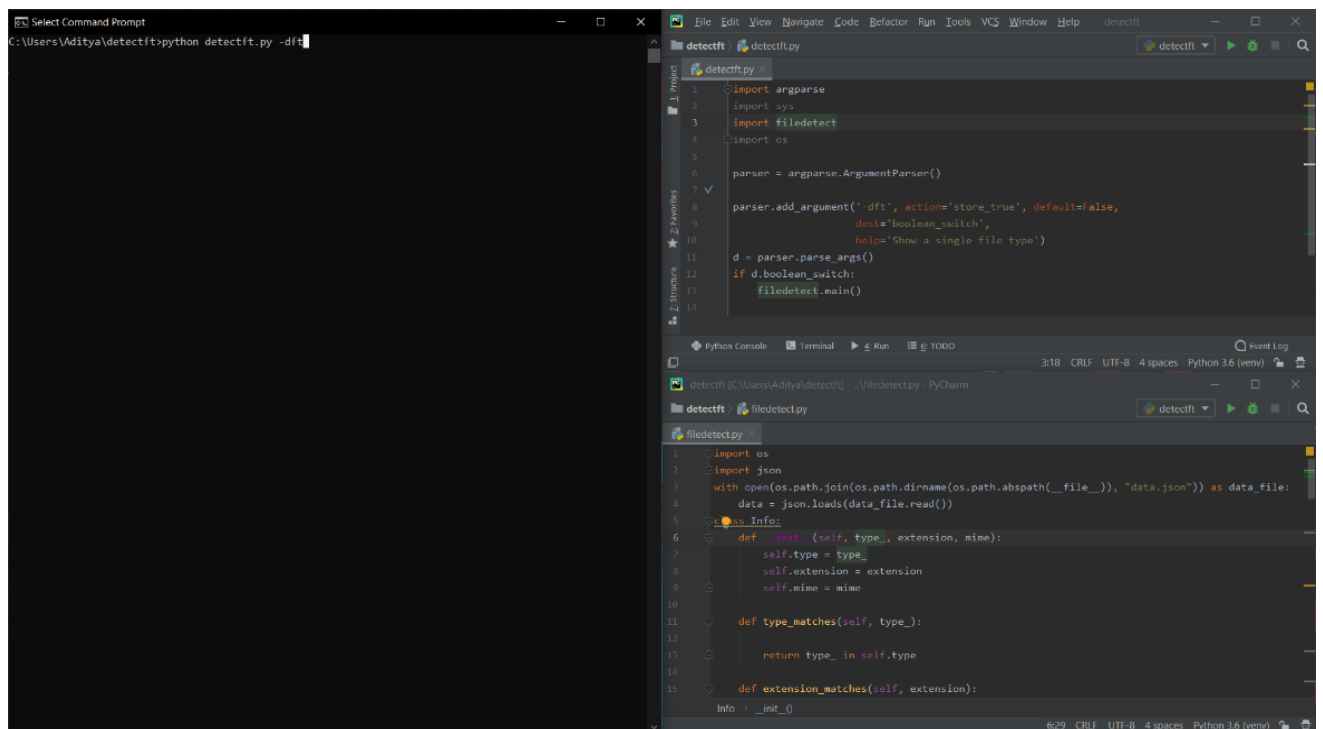
We have seen three methods to detect file type out of which content-based searching is the best, as it is accurate and valid to a larger number of files. Under content-based file type detection, we saw two methods Feature Extraction and Content Sampling Technique, here the feature extraction methods proved to be better as it is highly effective as it achieves high accuracy using a small number of features[2]. Moreover, it can substantially save classification time.. when it comes to the algorithm we looked at three algorithms BFA, BFC, and FHT, here BFA had the fastest approach but then it wasn't accurate, BFC has only limited usage, and though slow FHT gave the most accurate results. [3]

Deliverable: Here are a few pictures of the code used along with the output using magic number s to detect file type



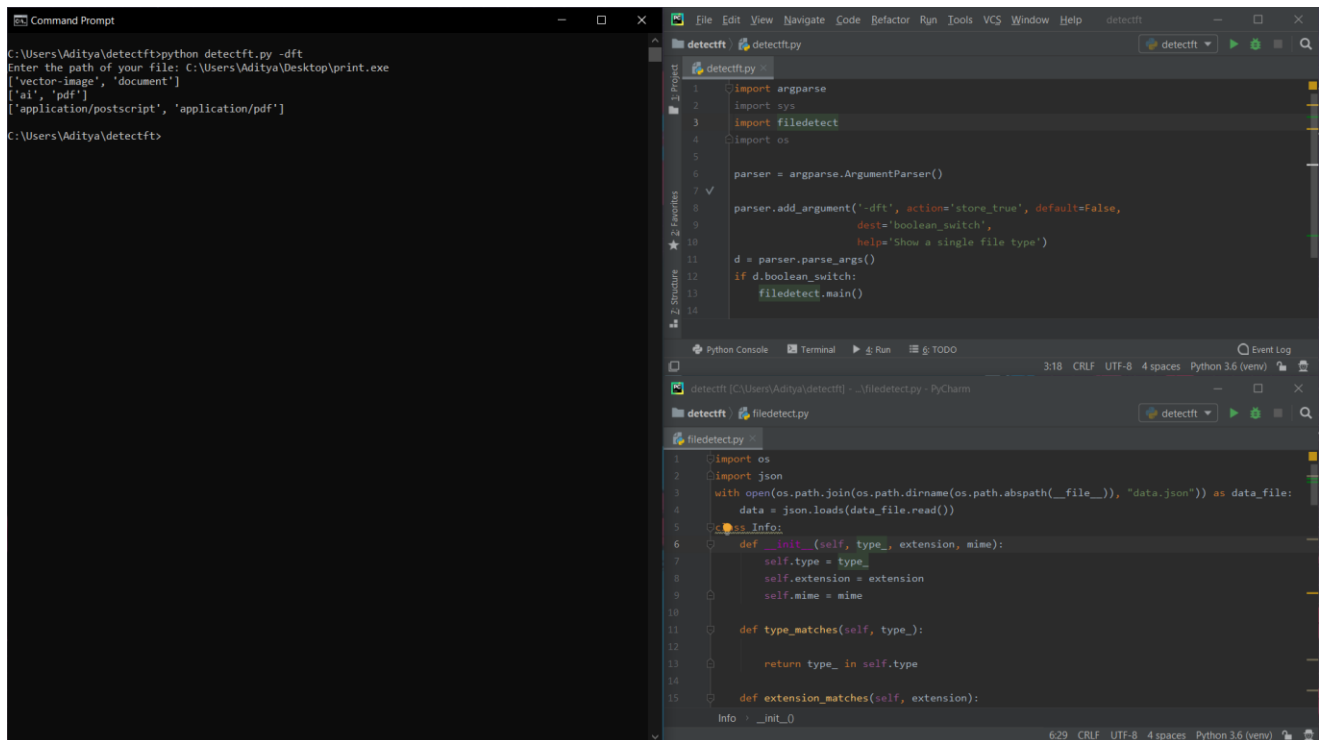
```
detectft.py
1 import argparse
2 import sys
3 import filedetect
4 import os
5
6 parser = argparse.ArgumentParser()
7
8 parser.add_argument('-dft', action='store_true', default=False,
9                     dest='boolean_switch',
10                    help='Show a single file type')
11
12 d = parser.parse_args()
13 if d.boolean_switch:
14     filedetect.main()
```

```
filedetect.py
1 import os
2 import json
3 with open(os.path.join(os.path.dirname(os.path.abspath(__file__)), "data.json")) as data_file:
4     data = json.loads(data_file.read())
5
6 class Info:
7     def __init__(self, type_, extension, mime):
8         self.type = type_
9         self.extension = extension
10        self.mime = mime
11
12    def type_matches(self, type_):
13        return type_ in self.type
14
15    def extension_matches(self, extension):
16        return extension in self.extension
```



```
detectft.py
1 import argparse
2 import sys
3 import filedetect
4 import os
5
6 parser = argparse.ArgumentParser()
7
8 parser.add_argument('-dft', action='store_true', default=False,
9                     dest='boolean_switch',
10                    help='Show a single file type')
11
12 d = parser.parse_args()
13 if d.boolean_switch:
14     filedetect.main()
```

```
filedetect.py
1 import os
2 import json
3 with open(os.path.join(os.path.dirname(os.path.abspath(__file__)), "data.json")) as data_file:
4     data = json.loads(data_file.read())
5
6 class Info:
7     def __init__(self, type_, extension, mime):
8         self.type = type_
9         self.extension = extension
10        self.mime = mime
11
12    def type_matches(self, type_):
13        return type_ in self.type
14
15    def extension_matches(self, extension):
16        return extension in self.extension
```



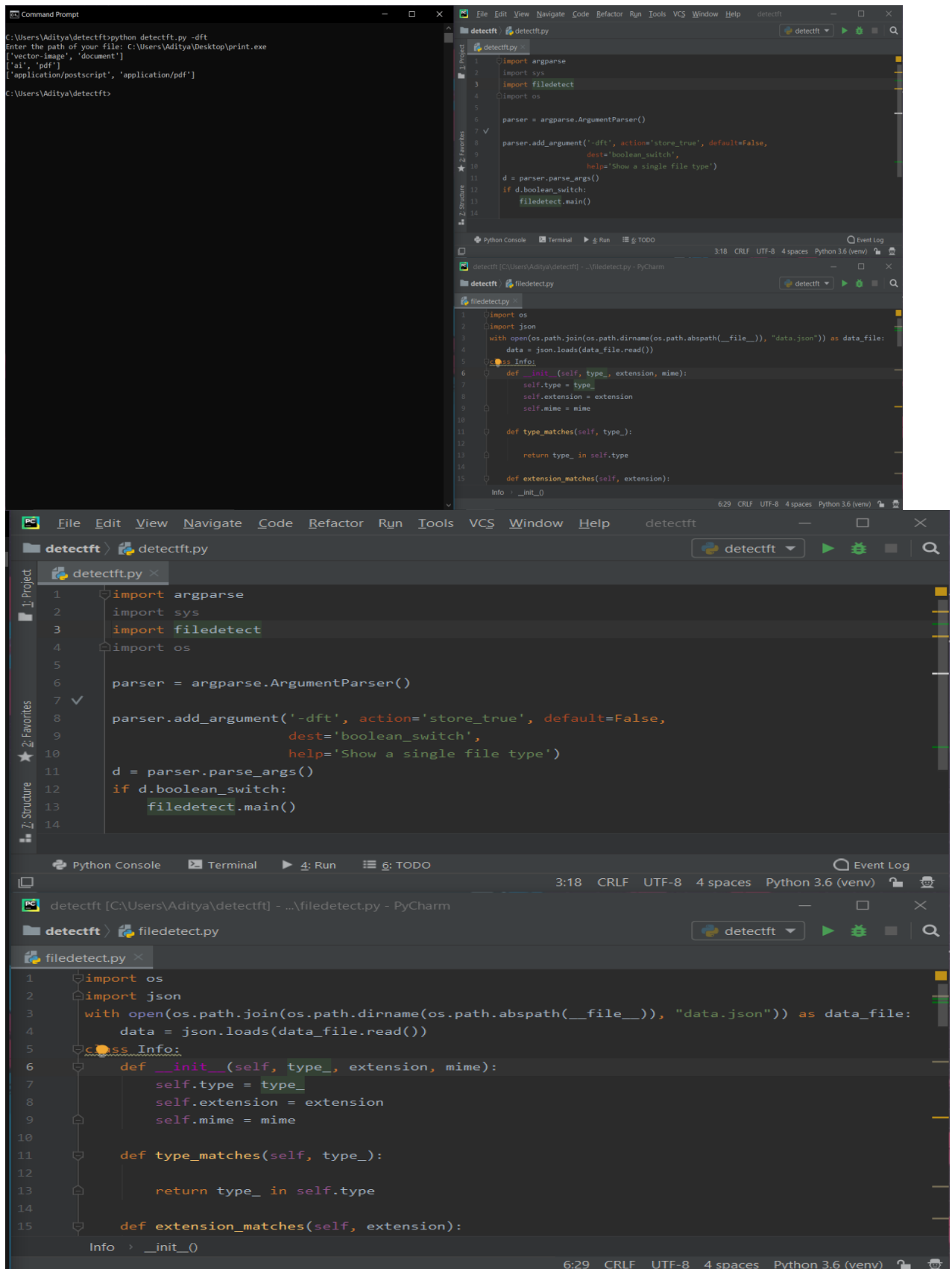
The image shows a development environment with a Command Prompt and a code editor. The Command Prompt on the left shows the execution of a Python script named `detectfft.py` with the `-dft` flag, which prints a list of file types: `['vector-image', 'document']`, `['ai', 'pdf']`, and `['application/postscript', 'application/pdf']`.

The code editor on the right displays two Python files. The top file, `detectfft.py`, uses `argparse` to handle command-line arguments and `filedetect` to perform file type detection. The bottom file, `filedetect.py`, defines a `FileInfo` class with methods for initializing file information and matching file types and extensions.

```
Command Prompt
C:\Users\Aditya\detectfft>python detectfft.py -dft
Enter the path of your file: C:\Users\Aditya\Desktop\print.exe
['vector-image', 'document']
['ai', 'pdf']
['application/postscript', 'application/pdf']
C:\Users\Aditya\detectfft>
```

```
detectfft.py
1 import argparse
2 import sys
3 import filedetect
4 import os
5
6 parser = argparse.ArgumentParser()
7
8 parser.add_argument('-dft', action='store_true', default=False,
9                     dest='boolean_switch',
10                    help='Show a single file type')
11
12 d = parser.parse_args()
13 if d.boolean_switch:
14     filedetect.main()
```

```
filedetect.py
1 import os
2 import json
3 with open(os.path.join(os.path.dirname(os.path.abspath(__file__)), "data.json")) as data_file:
4     data = json.loads(data_file.read())
5
6 class FileInfo:
7     def __init__(self, type_, extension, mime):
8         self.type = type_
9         self.extension = extension
10        self.mime = mime
11
12    def type_matches(self, type_):
13        return type_ in self.type
14
15    def extension_matches(self, extension):
16        return extension in self.extension
```



References

- [1] File Type Detection Technology- Douglas J. Hickok, Daine Richard Lesniak, Michael C. Rowe, Ph.D.
- [2] Fast File-type Identification Irfan Ahmed, Kyung-suk Lhee, Hyunjung Shin, ManPyo Hong Ajou University, South Korea {irfan, klhee, shin}
- [3] Content-Based File Type Detection Algorithms Mason McDaniel and M. Hossain Heydari¹ Computer Science Department James Madison University Harrisonburg, VA 22802
- [4] Mason McDaniel, Automatic File Type Detection Algorithm, Masters Thesis, James Madison University, 2001.
- [5] A New Approach to Content-based File Type Detection--Mehdi Chehel Amirani Mohsen Toorani Ali Asghar Beheshti Shirazi
- [6] FILExt, The file extension source.
- [7] Cole, E., Hiding in Plain Sight, Wiley, Indianapolis, 2003.
- [8] P. Baldi, and K. Hornik, Neural networks and principal component analysis
- [9] T.D. Sanger, Optimal unsupervised learning in a single linear feedforward neural network
- [10] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst.