Computational Intelligence to aid Text File Format Identification

Santhilata Kuppili Venkata, Alex Green
The National Archives

Abstract

One of the challenges faced in digital preservation is to identify the file types when the files can be opened with simple text editors and their extensions are unknown. The problem gets complicated when the file passes through the test of human readability, but would not make sense how to put to use! The Text File Format Identification (TFFI) project was initiated at The National Archives to identify file types from plain text file contents with the help of computing intelligence models. A methodology that takes help of AI and machine learning to automate the process was successfully tested and implemented on the test data. The prototype developed as a proof of concept has achieved up to 98.58% of accuracy in detecting five file formats.

1 Motivation

As an official publisher and guardian for the UK Government and England and Wales, The National Archives[1](TNA) collates iconic documents from various government departments. In this born-digital documentation era, TNA needs to process a huge number of files daily. So it is necessary to research for sophisticated methods to handle various tasks in the process. File format identification of plain text files is one such example.

1.1 How a simple plain text file can create confusion?

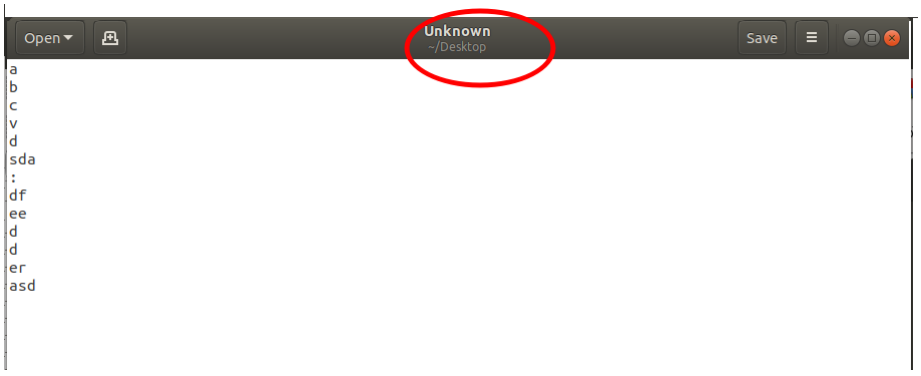

Figure 1: A sample text file with no file extension

In this digital era, files are often generated in an integrated development environment. Each document is supported by multiple files. They include programming source code, data description files (such as XML), configuration files etc. Contents of the supporting files are often

[1]http://www.nationalarchives.gov.uk

human-readable. i.e they can be opened as plain text files using a simple text editor. But if the file extensions are missed or corrupted, it is hard to know how to put the file to use!! A sample file with missing extension is shown in the Fig. 1. The file shown here contains some characters from the Roman alphabet written in a column. At first glance, one would think that this must be a simple exercise to typewriting characters in order. Someone familiar with the Unix environment cannot rule out the possibility of this file being a part of a bash script/commands. To a naive user, they appear to be a set of encrypted commands. Nevertheless, the question remains, how can we make use of the file even though it does not make any sense at the moment? If we have thousands of such files, it is impossible to examine each file physically, so we need to automate the process of file type identification.

1.2 How Big is the Problem?

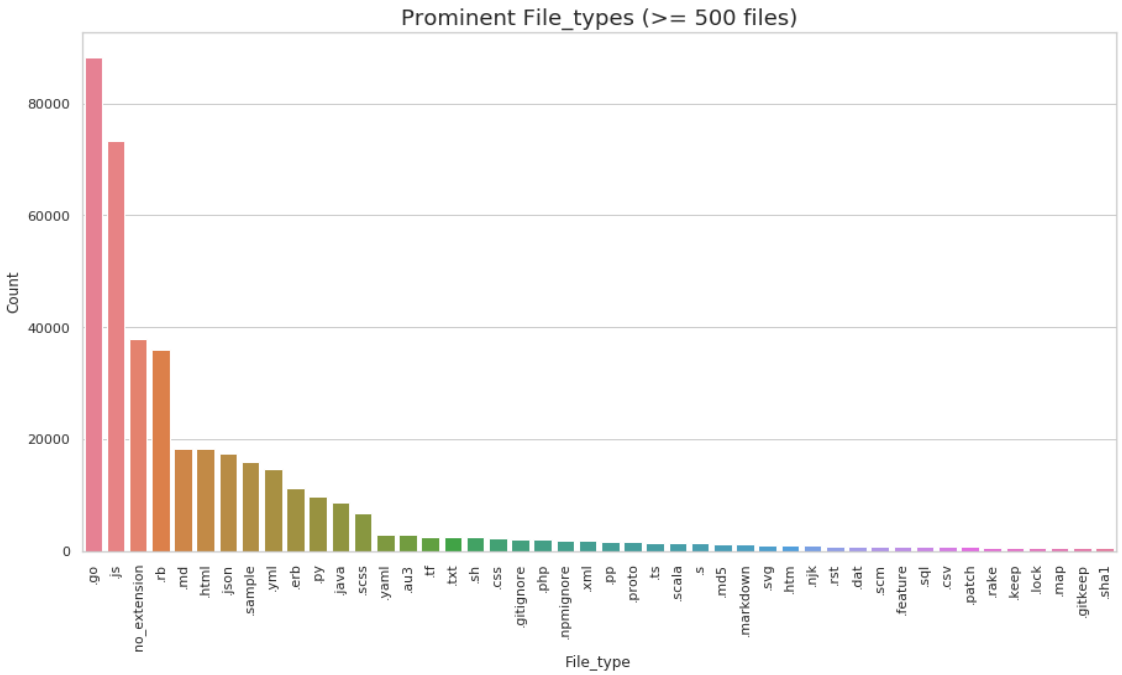

Figure 2: Some prominent file types found in the repositories

To estimate the enormity of the problem, we have cloned files from publicly available Github repositories of the Government Digital Service (GDS)[2] and The National Archives[3]. As TNA handles similar data, it makes sense to use these repositories as a testbed. In all, we have cloned 1457 public repositories from these two sources. They contain over 410,000 files representing 928 file types that can be opened with a simple text editor program. Fig. 2 shows a partial bar graph of file type categories that contain at least 500 files of each format in our sample dataset.

1.3 Our Starting Point

It is an extensive task to develop a single classifier model that classifies all 928 file types. So we have grouped files into 14 categories to understand the priority of the file types that need

[2]https://github.com/alphagov
[3]https://github.com/nationalarchives

immediate attention. The distribution of 13 main file categories (in blue) and the corresponding number of files (in red) for each category available in the corpus is shown in Fig. 3. The 14th category is a collection of all file types that cannot be categorized as any of the remaining 13 file categories. From the above patterns and DROID [1] reports, we found that programming

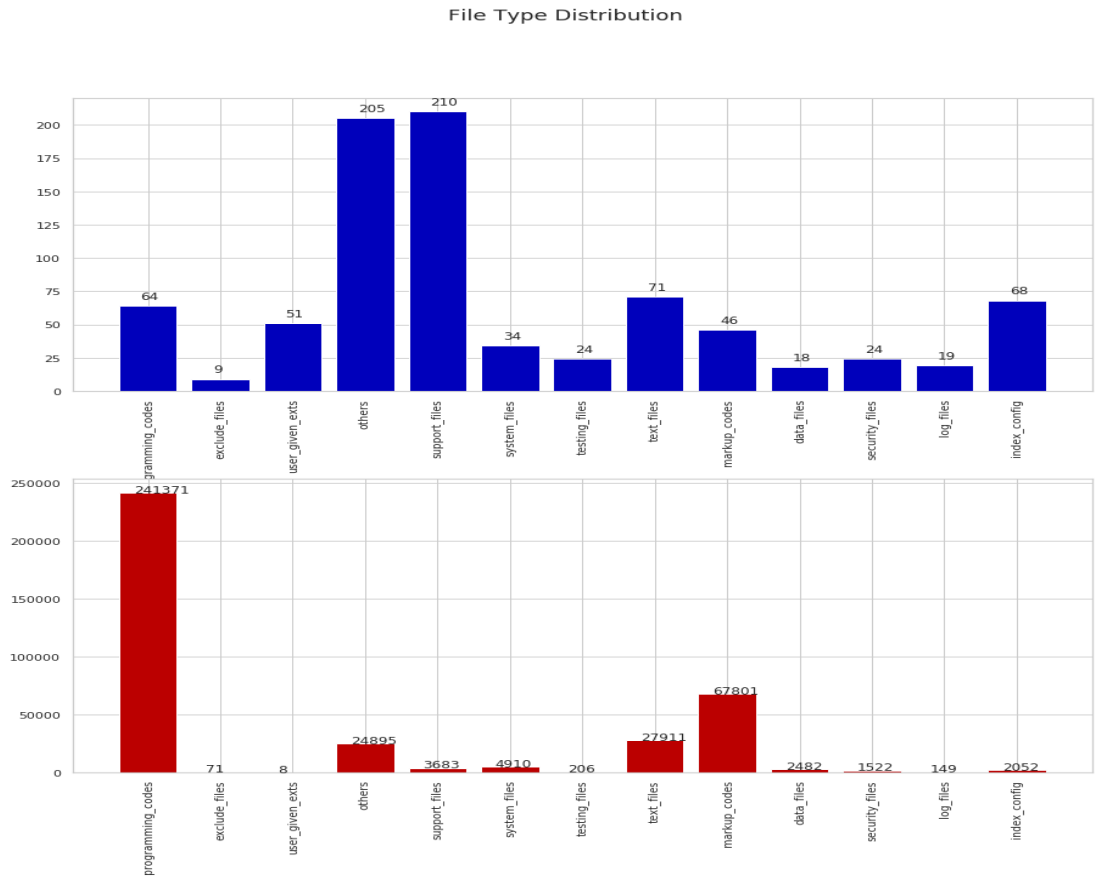

Figure 3: Distribution of file types in the corpus

codes and text files are the major file types that TNA receives regularly. So for the initial experimentation, we started with a small subset (22,292 files) consisting of five file types: two programming source code file types (Java, Python), plain text files (.txt) and two delimiter separated file types (.csv and .tsv).

Aside from digital archiving, file type identification is a serious problem in the areas of digital forensics and cybersecurity. The research in digital forensics is mainly focused on the identification of image file types and their metadata. While most of the research is targeted for binary file formats, very little work is done in the plain text file category. Though our target file types are different, we could adopt their approaches and methods to some extent.

There are mainly two approaches to work with plain text files. The first approach is to treat the file as a plain text file (no prior knowledge about the file type) and search for specific characteristics for possible file types. The signature of the file type is a combination of characteristics of that file type. This is a generic method and can be extended to any number of file types. However, this approach needs a thorough knowledge of the file type to generate its characteristical features. The second approach is based on prior knowledge about a file. For example, if we have an intuition about a file belonging to a programming language, we could validate the file type by running its compiler(s) or search for specific text patterns corre-

sponding programming language. We followed the first approach as our file corpus consists of a variety of file types. We have developed a flexible methodology (described in section 3) to reflect this approach. Our initial prototype makes use of machine learning algorithms and can identify five formats: .py, .java, .txt, .csv, and .tsv.

1.4 Problem Statement

A digital document is often created with the help of several supporting files in an integrated development environment. A corrupted or misidentified support file fails to reconstruct the document correctly. Hence it is necessary to identify files correctly. Several tools are available to identify file formats given the file extension and file's signature (such as magic bytes). It is difficult, however, to identify the type of a plain text file without any information. We need to identify the file from its contents. With thousands of documents to handle, a manual inspection of the contents of each file is tedious and time-consuming. We need to automate this process. We can formulate the research question as follows:

Research Question: How to correctly identify the file type of a plain text file from its contents?

To answer this question, we have carried out an extensive literature survey to review existing methodologies, approaches formulated and their adaptability from other fields. The survey of methods and their usability is explained in section 2. Relevant algorithms reconstructed to our task are explained in section 4. Given the nature of the problem, we narrowed our problem belonging to the classification category of supervised learning. A Python-based machine learning prototype was developed to understand the intricacies of different classification models during the 'proof of concept' development phase. The model construction, testing and evaluation are discussed in section 5.

2 Literature Review

Automated file type identification (AFTI) is a highly researched problem in digital forensics and related fields. Researchers have concentrated on the identification of image file types with corrupted metadata and missing chunks from the contents. Methods were developed to reconstruct damaged files from their fragments. File type identification is using metadata is a widely accepted paradigm for AFTI. It includes information about file extensions, header/footer signatures [1, 2], binary information such as magic bytes etc. All these methods work well when the metadata is available and unaltered. However, traditional approaches are not reliable when the integrity of the metadata is not guaranteed. An alternative paradigm is to generate 'fingerprints' of file types based on the set of known input files and using fingerprints to classify the type of the unknown file. Another prominent approach is to calculate the centroid for a given file type from its salient features. Each unknown file is examined for the distance from the known set of centroids to predict the file type. On the other hand, the centroid paradigm uses supervised and unsupervised learning techniques to infer a file (object) type classifier by exploiting unique inherent patterns that describe a file type's common file structure. Alamri et al. [3] have published a taxonomy of file type identification. Their work ranges over 30 different algorithms and approaches. In this section, we review the literature related to predicting file type from fragments and content-based methods.

2.1 File Type Identification from File Fragments

Identification of file type from its fragments is mainly used as a recovery technique. It allows file recovery of the file (or rebuild) without contextual information or metadata. This process is also referred to as 'file carving' in some of the literature. Image type files are mainly targeted by this technique.

Calhoun et al. [4] investigated two algorithms for predicting the type from fragments in computer forensics. They have performed experiments on the fragments that do not contain header information. First algorithm was based on the linear discriminant and the second was based on the longest common sub-sequences of fragments. Their work provided various relevant statistics such as byte frequency, entropy, etc. as features to predict the file type. Ahmed et al. [5, 6] also published two techniques to identify the file types from file fragments. These techniques aim to reduce the time consumed to process the contents. Their first technique selects a subset of features describing the frequency of occurrence of certain fragments. The second technique speeds up classification by randomly sampling file blocks. They have performed experiments on .png, .jpg and .tiff file types. Poisel et al. [7, 8] published a comprehensive survey of file carving research to detect the file types from their fragments. They have also provided a file carving ontology useful for researchers. In a similar work, Evensen et al. [9] explored the use of the naive Bayes classifier combined with n-gram analysis of byte sequences in files to correctly identify the file type. Gopal et al. [10] presented the evaluation and analysis of the robustness of Support Vector Machine (SVM) and k-Nearest Neighbours (kNN) in handling damaged files and file segments. They have restricted their study to the file type identification from the metadata. The evaluation reveals that SVM and kNN learn better than any commercial off-the-shelf tools developed based on file extensions. In his thesis, Wilgenbus [11] presented a combined multi-layer perceptron neural network and linear programming discriminant classifiers for the multiple class file fragment type identification problems. This solution could help our text file format identification problem, as neural networks learn from features of the contents and helps in classification of discrete file types. In their work, Karampidis et al. [12, 13] examine a three-stage methodology for AFTI, using feature selection (Byte Frequency Distribution) and feature selection using genetic algorithm. They have tested with classification models including decision tree, SVM, neural network, logistic regression and kNN. Their methodology showed that artificial neural networks performed with a very high and exceptional accuracy in most cases.

2.2 Content-based File Type Identification

Content-based file type detection methods have proved to be robust and more accurate so far. They are built on the principle of extracting features from the files. Initial work on content-based file type identification [14, 15] was based on three algorithms: byte frequency analysis, byte frequency cross-correlation and File header/trailer analysis.

Li et al. [16] have provided improvements to these algorithms by generating file prints (file signatures) using K-means algorithm with Manhattan distance metric. They produced file prints with the help of the statistical features extracted and selected. The file prints are also mentioned as *'centroid'* in literature. An unknown file is tested against a set of known centroids. The distance between the centroids is compared to predict the possible file type. The Mahalanobis distance metric is deployed for the comparison. The file prints (centroids) are developed using Natural Language Processing (NLP) techniques such as pattern matching of n-gram contiguous sequence models. While their work is a pioneer in its kind, their approach restricts the input file to follow a specific style only. Also, they fail to differentiate files when

the target file types have almost similar structures. For example, Java and C programming source codes. We need to generate file features and classification models in such a way that they describe file types distinctly.

Other improvements in this area include neural networks [17] and Byte Frequency Distribution (BFD) to classify file types [18, 19]. Amirani et al. [20] proposed a content-based file type detection method for files normalised using BFD. Their model uses principal component analysis for feature selection. The model is then fed into an auto-associative unsupervised neural network. Mitlohner et al. [21] published a comprehensive study of characteristics of open data CSV files. Their work analyzes an open data corpus containing resources from a data consumer perspective. Their study provided a deep insight to feature engineering CSV file type.

Predicting the file type from the contents of text files complicates the problem of AFTI. Though several approaches are available, they are highly domain-specific. Hence we could not use them for the identification of file types from their contents. We need to research generic methods to fill this gap based on existing approaches.

3 Methodology

TNA deals with a huge variety of file types for digital archiving. Hence an iterative process model is appropriate to include file features gradually. The methodology should be flexible to add more file types progressively. As and when a new file type is to be included, its features (specific characteristics) should be compared against the existing features of other file types and engineered to add to the list. Hyperparameters for the models should be tuned to get a better performance. The flow graph in Figure 4 shows the methodology developed and used by us.

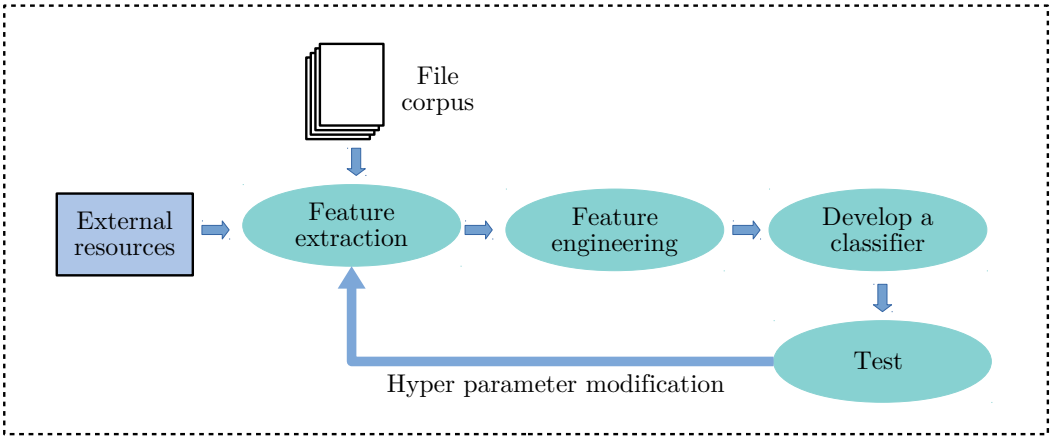

Figure 4: Methodology to include file types progressively

a) *The File Corpus* is the set of files that serve as the dataset to the file type identification task. As TNA is set to receive digital documents from various government departments, it was decided to use data files from the Github repositories of the Government Digital Service. We have also collected file samples from TNA's Github repositories to compile the data file corpus for this prototype.

b) *External resources* comprise published methodologies, approaches and algorithms used

in various environments. For example, DROID - the binary file format identifier tool is used to eliminate known file types as a first step.

c) *Feature Extraction* Features of a file play an important role in the identification of their types. Feature extraction is the process of extracting features from the files in the corpus. Characteristic features that determine the styles and nature of the file type will be extracted during this step.

d) *Feature Engineering* is the process of using domain knowledge of the data to create features that make machine learning algorithms work. It helps to fine tune the machine learning models by reducing the computational processing overhead.

e) *Classifier Development and Test* Machine learning (ML) is chosen to develop a classifier. ML algorithms are used to understand and extract the patterns from the data and help to predict the outcome.

4 Data Pre-processing

Text file format identification is a non-linear learning problem given the correlation between features of different file types. For example, a very high correlation between the Java and Python programming file structures lead their features to be interdependent. Similarly, .csv and .tsv files share their file features. Many times a comma separated files (.csv) may contain unformatted textual lines, leaving a very small difference between .txt and .csv file formats to differentiate. The following phases in the life-cycle of the development of the model mirror these facts.

4.1 Feature Extraction

Our data consists of unstructured text files. So the first phase was to recognise features that describe Python and Java source codes and .txt, .csv and .tsv files correctly. We automated the process of feature extraction from each file to make the process uniform across all file types. The feature extraction process scans through each token (word) in the file and creates a quantifiable feature set. A total of 45 features were identified for extraction from each file.

4.2 Feature Engineering

Some of the facts that help to find relevant features during the feature engineering phase are as given below.

- a Python source code file differs from a Java source code file by its commenting style, strict indentation requirement at the beginning of each line of the code, use of specific keywords etc.
- the Java programming source code needs to follow a pre-defined structure to be able to compile successfully.
- while every line in the Java code needs to be ended with a ';' (semi-colon), python does not need any.
- even though .csv and .tsv files are largely categorised as text-based, they can be recognised by the use of the number of commas (or other delimiter characters). Hence a

comparison of the number of commas between files could become a deciding factor to classify a file between .csv and .txt.

- in general, a .txt file got no restrictions on how to create one compared to .csv or .py. It is difficult to extract a pattern from a normal .txt file. Hence we have used the count of 'stopwords' (common words in English) from the NLTK library. We started with the assumption that the usage of stopwords is more in normal text files than programming codes or data files.

- another significant characteristic is the 'word-combination' proximity. It differs from n-gram contiguous model. For example, the combination of words such as <def-return>, <if-then-else> etc. are used more closely in the programming codes than a .txt file. So, we derived a threshold for the word-combination word sets.

After feature engineering, 33 features were selected for classification. Features extracted and used for classification are listed in the Appendix.

4.3 Selection of Classification Models

There are four prominent categories of classification algorithms. They are (i) Linear models, (ii) Tree-based algorithms, (iii) k-nearest algorithms and (iv) Neural network based. Since our problem has discrete outputs and non-linear inputs, the linear models are omitted. All models were trained and hyperparameters were tuned to improve the accuracy over many iterations. A detailed explanation of the training and testing is explained in section 5.

5 Classification Models & Evaluation

The Jupyter notebooks developed as a proof of concept is available here[4].

5.1 The Decision Tree Classifier

A decision tree [22] is a flowchart-like tree structure where an internal node represents a feature (or attribute). The branch represents a decision rule, and each leaf node represents the outcome. Each parent node learns to partition the data based on the attribute value. It partitions the tree recursively until all the data in the partition belongs to a single class. Decision trees can handle high dimensional data with good accuracy. The decision tree is implemented with the help of the Python scikit-learn library. The decision tree algorithm is explained in Algorithm 1.

Algorithm 1 Decision Tree Algorithm

1: Place the best attribute of our dataset at the root of the tree.
2: Split the training set into subsets. Subsets should be made in such a way that each subset contains data with the same value for an attribute.
3: Repeat step 1 and step 2 on each subset until you find leaf nodes in all the branches of the tree.

[4]https://github.com/nationalarchives/Text-File-Format-Identification

Results:

The evaluation of the decision tree classification is shown in the confusion matrix in Table 1. The train-to-test ratio is set ideally as 80:20 to achieve better accuracy. Though the accuracy of classification is very high, we consider the precision metric more, given the non-uniform distribution of file types in the file corpus.

Table 1: Accuracy and precision metrics for Decision Tree classification model

Accuracy	98.58%
Precision score	86%

5.2 The k-Nearest Neighbour Classifier

The k-Nearest Neighbour [22] (kNN) is based on feature similarity that determines how we classify a given data point. The output is a class membership (predicts a class — a discrete value). An object is classified by a majority vote of its neighbours, with the object being assigned to the class most common among its k-nearest neighbours. The kNN algorithm[5] is explained in Algorithm 2.

Algorithm 2 k-Nearest Neighbour Algorithm

1: Load the data
2: Initialize k to be a chosen number of neighbours
3: For each data point a) Calculate the distance between the data point and the example in the train data
4: Sort the calculated distances in ascending order and choose top k rows
5: Get the most frequent class as the predicted class

Results:

The kNN classification evaluation is shown in a confusion matrix in Table 2. The train-to-test ratio is set ideally to 80:20 to achieve better accuracy. The 'minkowski' distance metric[6] is used to establish the distance between classes. The value for k is set to 3. Due to the uneven distribution of the file types in the file corpus, though the accuracy is 94%, the precision score should be considered more.

Table 2: Accuracy and precision metrics k-nearest neighbour classification model

Accuracy	94.03%
Precision score	80%

5.3 The Multilayer Perceptron based Classifier

A Multilayer perceptron (MLP) is a deep, artificial neural network. It is a non-linear classification model. It is composed of more than one perceptron [23, 24]. Each perceptron is composed

[5]https://www.sciencedirect.com/topics/biochemistry-genetics-and-molecular-biology/k-nearest-neighbor
[6]https://www.sciencedirect.com/topics/computer-science/minkowski-distance

of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and between these two, an arbitrary number of hidden layers that are the true computational engine. MLPs with one hidden layer are capable of approximating any continuous function. They train on a set of input-output pairs and learn to model the correlation between those inputs and outputs. Training involves adjusting the parameters, or the weights and biases, of the model, in order to minimize error. Backpropagation is used to make those weight and bias adjustments relative to the error, and the error itself can be measured in a variety of ways. A neural network executes in two phases: Feed-forward and Backpropagation.

Feed-forward These are the steps performed during the feed-forward phase:

1. The values received in the input layer are multiplied with the weights. A bias is added to the summation of the inputs and weights.

2. Each neuron in the first hidden layer receives different values from the input layer depending upon the weights and bias. Neurons have an activation function that operates upon the value received from the input layer.

3. The outputs from the first hidden layer of neurons are multiplied with the weights of the second hidden layer; the results are summed together and passed to the neurons of the proceeding layers. This process continues until the outer layer is reached. The values calculated at the outer layer are the actual outputs of the algorithm.

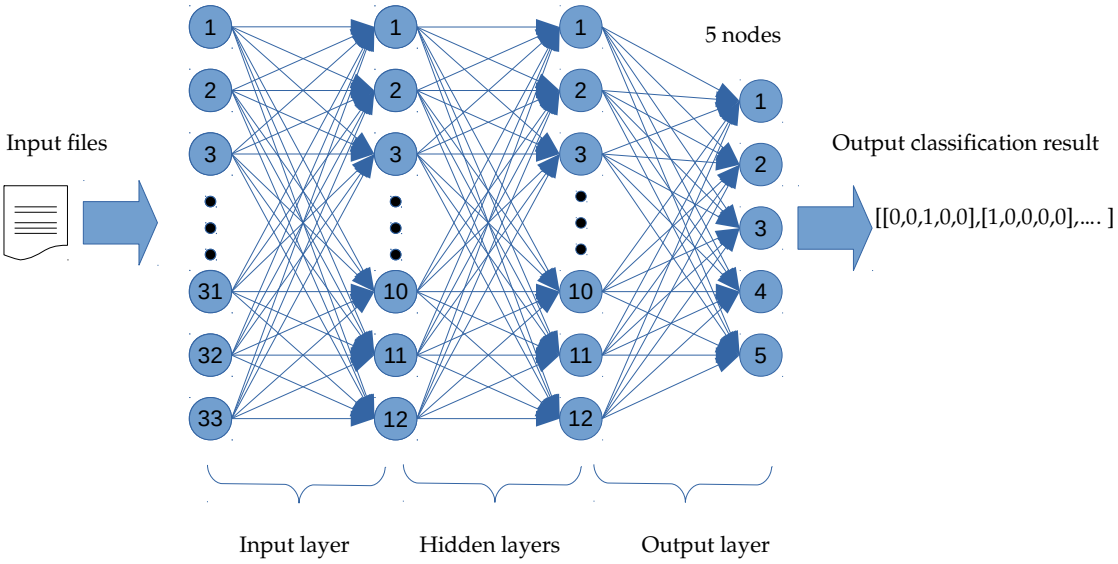

Figure 5: Multilayer perceptron neural network used for classification

Back Propagation The predicted output is not necessarily correct right away after the feed-forward phase. To improve these predicted results, a neural network will go through a back-propagation phase. During backpropagation, the weights of neurons are updated in a way that the difference between the desired and predicted output is as small as possible.

The MLP model generated for our classification problem was designed as a 3-layer fully connected neural network with 33 nodes in the input layer, 12 nodes each in the hidden layers

and 5 nodes (one for each of the output class) in the output layer. The number of nodes in each of the layers was decided on a trial and error basis. Our MLP model is shown in Figure 5. The parameters set for the MLP is given in Table 3.

Table 3: The values set for MLP parameters

Parameter	**Value set to**
activation (input layer/hidden layer 1)	relu
activation (hidden layer 1/hidden layer 2)	relu
activation (hidden layer 2/output layer)	softmax
optimizer	adam
loss type	categorical_crossentropy
metrics	accuracy
# epochs	30
batch_size	20

Results:

The MLP neural network classification results are presented in a confusion matrix in Table 4. The train-to-test ratio is set ideally as 80:20 to achieve better accuracy. The train-test set is kept the same across all classification methods. The test accuracy 97.80% is almost as good as the decision tree.

Table 4: Accuracy MLP Neural network classification model

test - Accuracy	97.37%
test - loss	0.15
train - Accuracy	97.80%
train - loss	0.13

6 Conclusion & Scope

TNA has initiated the project:'Text File Format Identification' to identify file formats for plain text files. A prototype was developed using computational intelligence models. The GitHub repositories of the Government Digital Service and The National Archives were used for testing and training purposes. This prototype has achieved 98.58% accuracy (with a decision tree classifier model) in identifying five file formats. Python and Java programming code file types were identified with higher accuracy compared to text file formats. This leads to the assumption that the inherent structure of programming files has been captured better than .txt/.csv/.tsv files with no fixed structure. In comparison, .tsv file identification has not fared well due to the low representation of .tsv files in the input dataset. The decision tree model has performed better than the other two models. We have successfully established a methodology which is flexible enough to work with more file formats in future.

In future, we could focus on revising the dominant feature identification for .csv and .tsv file types. In this project, we assumed that each .csv file contained only one table. However, it is possible that multiple tables exist within a single .csv file. This issue should be investigated. Even though the current prototype works well for the five file types, a revision of feature engineering will be necessary whenever a new file type is included.

Acknowledgements

To Paul Young and Ian Henderson, for their deep insights and support at every stage.

References

[1] DROID. http://droid.sourceforge.net/, 2013.

[2] TrID. http://mark0.net/soft-trid-e.html.

[3] Nasser S. Alamri and William H. Allen. A taxonomy of file-type identification techniques. In *Proceedings of the 2014 ACM Southeast Regional Conference*, ACM SE '14, page 49:1–49:4, New York, NY, USA, 2014. ACM.

[4] William C. Calhoun and Drue Coles. Predicting the types of file fragments. *Digit. Investig.*, 5:S14–S20, September 2008.

[5] Irfan Ahmed, Kyung suk Lhee, Hyunjung Shin, and ManPyo Hong. Content-based file-type identification using cosine similarity and a divide-and-conquer approach. *IETE Technical Review*, 27(6):465, 2010.

[6] Irfan Ahmed, Kyung-Suk Lhee, Hyun-Jung Shin, and Man-Pyo Hong. Fast content-based file type identification. In *Advances in Digital Forensics VII*, page 65–75. Springer Berlin Heidelberg, 2011.

[7] Rainer Poisel and Simon Tjoa. A comprehensive literature review of file carving. In *2013 International Conference on Availability, Reliability and Security*. IEEE, sep 2013.

[8] Rainer Poisel, Marlies Rybnicek, and Simon Tjoa. Taxonomy of data fragment classification techniques. In *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, page 67–85. Springer International Publishing, 2014.

[9] John Daniel Evensen, Sindre Lindahl, and Morten Goodwin. File-type detection using naive bayes and n-gram analysis. In *2014: NISK 2014*, 2014.

[10] Siddharth Gopal, Yiming Yang, Konstantin Salomatin, and Jaime Carbonell. Statistical learning for file-type identification. In *2011 10th International Conference on Machine Learning and Applications and Workshops*. IEEE, dec 2011.

[11] Erich Feodor Wilgenbus. The file fragment classification problem : a combined neural network and linear programming discriminant model approach. Master's thesis, N, 2013.

[12] Konstantinos Karampidis, Ergina Kavallieratou, and George Papadourakis. Comparison of classification algorithms for file type detection a digital forensics perspective. *Polibits*, 56:15–20, 2017.

[13] Konstantinos Karampidis and Giorgos Papadourakis. File type identification - computational intelligence for digital forensics. *The Journal of Digital Forensics, Security and Law*, 2017.

[14] Mason McDaniel and M.Hossain Heydari. Content based file type detection algorithms. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. IEEE, 2003.

[15] M. McDaniel. Automatic file type detection algorithm. Master's thesis, 2001.

[16] W. J. Li, S. J. Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, page 64–71, June 2005.

[17] J. G. Dunham and J. C. R. Tseng. Classifying file type of stream ciphers in depth using neural networks. In *The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005.*, page 97–, Jan 2005.

[18] M. Karresand and N. Shahmehri. File type identification of data fragments by their binary structure. In *2006 IEEE Information Assurance Workshop*, page 140–147, June 2006.

[19] L. Zhang and G. B. White. An approach to detect executable content for anomaly based network intrusion detection. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 1–8, March 2007.

[20] Mehdi Chehel Amirani, Mohsen Toorani, and Sara Mihandoost. Feature-based type identification of file fragments. *Security and Communication Networks*, 6(1):115–128, apr 2012.

[21] J. Mitlöhner, S. Neumaier, J. Umbrich, and A. Polleres. Characteristics of open data csv files. In *2016 2nd International Conference on Open and Big Data (OBD)*, page 72–79, Aug 2016.

[22] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison Wesley, us ed edition, May 2005.

[23] H. Ramchoun, M. A. Janati Idrissi, Y. Ghanou, and M. Ettaouil. Multilayer perceptron: Architecture optimization and training with mixed activation functions. In *Proceedings of the 2Nd International Conference on Big Data, Cloud and Applications*, BDCA'17, page 71:1–71:6, New York, NY, USA, 2017. ACM.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

A Features generated from files

A.1 Original features generated

Table 5: Features extracted

Feature	Description
file_name	Name of the file along with its complete path
file_extension	File extension if available
num_lines	Number of lines in the file separated by newline character
header_info	File header information if available
trailer_info	Trailer information, if available
indentation	Number of spaces used for indentation (specific to Python)
eol_marker	End-of-line markers, if any (specific to Java)
sol_marker	Start-of-line markers, if any
isLowercaseMethods	Whether methods/functions start with lower case alphabets
num_stopwords	Number of stop words used (specific to text files)
num_Python_keywords	Number of Python key words within the file
num_Java_keywords	Number of Java key words used in the file
Python_comments	Number of Python style of comments
Java_comments	Number of Java style of comments
angular_brackets	Number of angular brackets used
curly_brackets	Number of curly brackets used
round_brackets	Number of round brackets used
square_brackets	Number of square brackets used
num_def	Number of 'def' used (specific to Python
num_returns	Number of times the key word 'return' used
if_else_proximity	Number of words between if and else (specific to programming codes)
num_carat	Number of times the carat symbol used (specific to csv and tsv)
num_comma	Number of times the comma symbol used (specific to csv and tsv)
num_fullstop	Number of times the fullstop symbol used (specific to csv and tsv)
num_tab	Number of times the tab used (specific to csv and tsv)
num_semicolon	Number of times the semi colon symbol used (specific to csv and tsv)
num_colon	Number of times the colon symbol used (specific to csv and tsv)
num_pipe	Number of times the pipe symbol used (specific to csv and tsv)
num_hash	Number of times the hash symbol used (specific to csv and tsv)
average_line_length	Average length of a line (in characters)
description	File description in short, if available
programming	Whether the file is a programming code, if known
stopwords_normalised	Normalised stop words across Java and Python
file_type	File Type information

A.2 Features Used

Out of the above features, following features are engineered for classification.

'num_lines', 'header_info', 'trailer_info', 'indentation', 'eol_marker', 'sol_marker', 'isLowercaseMethods', 'Python_comments', 'Java_comments', 'num_def', 'num_returns', 'if_else_proximity', 'num_carat', 'num_comma', 'num_fullstop', 'num_tab', 'num_semicolon', 'num_colon', 'num_pipe', 'num_hash', 'average_line_length', 'file_type', 'programming', 'stopwords_normalised', 'Python_keywords', 'Java_keywords', 'num_Python_comments_normalised', 'num_Java_comments_normalised',
'round_normalised', 'curly_normalised', 'square_normalised', 'angular_normalised', 'def_return_balance'