# GP-Fileprints: File Types Detection Using Genetic Programming

Ahmed Kattan[1], Edgar Galván-López[2], Riccardo Poli[1], and Michael O'Neill[2]

[1] School of Computer Science and Electronic Engineering, University of Essex, Colchester, UK
[2] Natural Computing Research & Applications Group, University College Dublin, Ireland
akatta@essex.ac.uk, edgar.galvan@ucd.ie, rpoli@essex.ac.uk, m.oneill@ucd.ie

**Abstract.** We propose a novel application of Genetic Programming (GP): the identification of file types via the analysis of raw binary streams (i.e., without the use of meta data). GP evolves programs with multiple components. One component analyses statistical features extracted from the raw byte-series to divide the data into blocks. These blocks are then analysed via another component to obtain a signature for each file in a training set. These signatures are then projected onto a two-dimensional Euclidean space via two further (evolved) program components. K-means clustering is applied to group similar signatures. Each cluster is then labelled according to the dominant label for its members. Once a program that achieves good classification is evolved it can be used on unseen data without requiring any further evolution. Experimental results show that GP compares very well with established file classification algorithms (i.e., Neural Networks, Bayes Networks and J48 Decision Trees).

## 1 Introduction

From the point of view of an operating system or standard high-level programming languages, a file is normally treated as a sequence of elementary data units, typically bytes. File format is a particular way to encode information for storage in a computer so that the file can be correctly interpreted by the operating system. Unfortunately, there are no universal standards for file types and there are hundreds of file types. This makes file type identification a difficult but increasingly significant problem. Different operating systems have traditionally used different approaches to solve this problem (i.e., file extensions and magic numbers). This, however, is very unreliable given that any user or application can easily change the extension of a file or change the file's meta data. A method is required to identify file's contents. This is useful for applications such as email spam filter, virus detection, forensic analysis and network security.

This paper proposes an application based on the use of Genetic Programming (GP) [7,12] to identify the file contents by analysing the raw binary streams. The question that we investigate is whether it is possible for GP to extract certain regularities from the raw byte-series of files and correlate them with particular data types without the need of any other meta data. The paper is organised as follows. In Section 2 we review previous work related to this research. Section 3 presents the details of the proposed method for file type detection. In Section 4, the experimental setup used to conduct our experiments is presented and in Section 5 we present and discuss our results. Finally, Section 6 draws some conclusions.

A.I. Esparcia-Alcazar et al. (Eds.): EuroGP 2010, LNCS 6021, pp. 134–145, 2010.

## 2 Previous Work

In [9], McDaniel and Heydari proposed an approach for automatically generating "fingerprints" for files. These fingerprints were then used to recognise the true type of unknown files based on their content instead of using the metadata associated with them. The authors used three algorithms to build these fingerprints: Byte Frequency Analysis (BFA), Byte Frequency Cross-Correlation (BFC) and the File Header/Trailer (FHT) algorithm. The BFA algorithm works by drawing a frequency distribution of the number of occurrences with which each byte value occurs in a file. This frequency distribution is useful in determining the type of a file because many file types have consistent patterns in their frequency distribution. The BFA algorithm presents some limitations related to the fact that it compares overall byte-frequency distributions. This issue is addressed by the BFC algorithm by taking the relationship between byte value frequencies into account. In BFC, two values are calculated: the average difference in frequency between all byte pairs and a correlation strength. Finally, the FHT algorithm works by using file headers and file trailers. These are pattern of bytes that appear in a fixed location of the file: at the beginning and the end of the file. The authors tested the described algorithms and constructed thirty file-type fingerprints using four test files for each file type (i.e., a total library of 120 files). They reported that BFA and BFC showed poor performance (i.e., an accuracy in the range of 27.5% and 45.83%) compared to FHT algorithm (which had an accuracy of 95.83%).

Later, this work has been criticised in [8], by Li *et al.*, who claimed that a single fingerprint is insufficient to represent whole classes of files. Li *et al.* proposed to analyse the data using n-grams to identify multiple centroids – fingerprints – for each file type. They applied three different techniques: i) Truncation, where part of the file header is analysed and compared with single representative fingerprints; ii) Multi-centroids, where a group of fingerprints is used to form clusters with K-means, each cluster representing a particular file type, then unseen data is classified according to minimal distance; iii) Exemplar files, where unseen data is compared to all fingerprints from all trained data types and classified based on the closest. The authors reported some problems when classifying similar data types such as GIF and JPG. Also, some difficulties appeared when classifying PDF and MS office file types, as some embedded images and figures mislead the algorithms.

Karresand and Shahmehri [6] proposed a method called Oscar that allows classification of data fragments based on their structures without the need of any other meta data (e.g., header information). For this purpose, they used the Byte Frequency Distribution (BFD) of data fragments and calculated the mean and the standard deviation for each byte value. When these measures are put together, they form a model which is used to identify unknown data fragments. In [5], the same authors extended this approach by calculating the rate of change (RoC) (i.e., the absolute value of the difference between two consecutive byte values in a data fragment). RoC allows incorporating the ordering of the bytes into the identification process. The authors reported that their approach, tested using only JPEG files, gave a 99.2% detection rate. The slowest implementation of the algorithm scans a 72.2MB in approximately 2.5 seconds and this scales linearly.

Hall and Wilbon [3] used a sliding window of fixed size and measured the entropy and the data compressibility with LZW compression to identify file types. For each

file type, these measurements were averaged from training examples and the standard deviation calculated for each corresponding point. Later, unseen data was compared with these models to predict their contents. The authors reported that entropy was not successful at associating the correct file type with unseen data. Also, this work reported that some file types such as BMP can vary greatly and it is very difficult to correctly classify them based on the proposed method.

Erbacher and Mullholland [1] focused their attention on the location and identification of data types embedded within a file, to offer analysts a technique to more efficiently locate relevant data on a hard drive. For this purpose, the authors used a range of statistical analyses with a variety of file types and were able to identify the types of data embedded within a file. The authors were able to identify five statistics that gave sufficient information to differentiate the different types of data: average, standard deviation, kurtosis, distribution of averages and distribution of standard deviations.

None of the previous methods used evolutionary algorithms, including GP, to solve the problem of identifying file types from their raw binary streams.

## 3 Approach

An ideal security system would be one that non-invasively reads data from files and detects their contents without the need for human intervention. In this paper we propose a system based on the use of the GP that takes one further step towards the ideal security system. The use of GP to identify file types from their raw binary streams has not been explored thus far. Here, we investigate how to evolve programs that detect the contents of files and inform the user of any suspicious contents according to predefined settings.

Our approach, broadly outlined in Fig. 1, works as follows. We try to spot regularities within the raw byte series and to associate them to different file types (e.g., TXT, PDF, JPG). Each class indicates the contents of the data. The system works in two main stages: *i) Training*, where the system learns to match different byte-series characteristics with different classes, and *ii) Testing*, where the system classifies unseen data. The system processes raw byte-series signals and performs three major functions: *a) Segmentation* of the byte-series based on their statistical features, *b) Fileprint creation*, and *c) Classification* of the identified fileprints into their types (e.g., TXT, PDF, JPG). For these tasks, GP has been supplied with a language that allows it to extract statistical features from byte-series. The selection of the primitives of the language was carefully made to avoid unnecessary growth in the search space, while at the same time ensuring that it is rich enough to express the solution. Table 1 reports the primitive set of the system.

The system starts by randomly initialising a population of individuals using ramped half-and-half [7]. As exemplified in Fig. 2, each individual has a multi-tree representation comprising one splitter tree, one fileprint tree and two feature-extraction trees. Multi-tree representations of this kind are common in GP, and have been used, for example, for data classification in [2] and [10].

We used a representation similar to the one proposed by Haynes in [4]. The population is stored into a 2D vector (vector of trees). Each individual is assigned to a fixed position. Thus, individuals components are co-evolved as they are always selected simultaneously.

In the next sections we describe the role of each of the trees in an individual in detail.

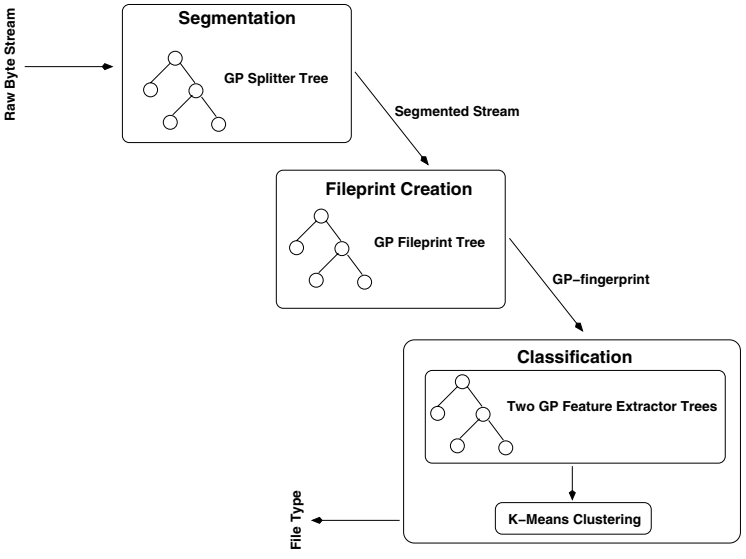


**Fig. 1.** Outline of our file-type detection process

**Table 1.** Primitive set used in our experiments

| *Function* | *Input* |
|---|---|
| Median, Mean, Average deviation Standard deviation, Variance, Skew, Kurtosis, Entropy, Geometric Mean | Vector of Integers (0-255) |
| $+, -, /, *, Sin, Cos, Sqrt, log$ | Real number |

**Table 2.** Parameters used to conduct our experiments

| *Parameter* | *Value* |
|---|---|
| Population Size | 100 |
| Generations | 30 |
| Crossover Rate | 50% |
| Mutation Rate | 50% |
| Elitism | 20% |
| Tournament Size | 5 |

### 3.1 Splitter Tree

It is very difficult to extract statistical features from the raw byte-series and directly correlate them with a particular data type. Furthermore, over time there has been an increase in the use of files with complex structures that store data of different types simultaneously. For example, a single game file might contain executable code, text, pictures and background music. Also, many file types, e.g., OpenOffice's ODT, Microsoft's DOCX or a ZIP file, are in fact archives containing inhomogeneous data. This makes the task of recognising file types is today even more difficult and traditional methods unreliable. It is, therefore, necessary to properly handle the fact files may contain multiple data types. The main job of the splitter trees is to split the given raw byte-series into smaller segments based on their statistical features in such a way that each segment is composed of statistically uniform data.

The system moves a sliding window of size $L$ over the given byte-series with steps of $S$ bytes, where, naturally, $S < L$.

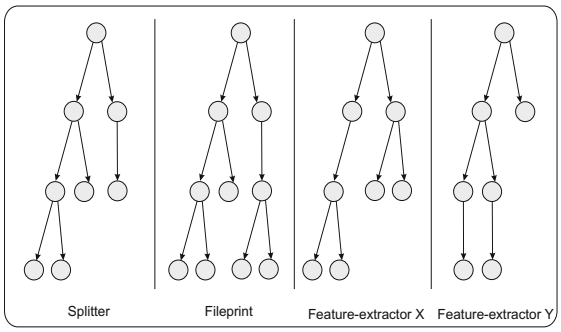


**Fig. 2.** A typical individual representation

At each step the splitter tree is evaluated. This corresponds to applying a function, $f_{\text{splitter}}$, to the data in the window. The output of the program tree is a single number, $\lambda$, which is an abstract representation of the homogeneity of the data in the window. The system then splits the byte-series at a particular position if the difference between the $\lambda$'s in two consecutive windows is greater than a predefined threshold, $\theta$. The threshold $\theta$ has been set arbitrarily ($\theta = 10$ in our implementation). In preliminary experiments we found that small changes in $\theta$ did not affect the performance of splitter trees. This is because evolution is free to change the magnitude of the outputs produced by splitter trees to adapt to the threshold. Also, after trying different settings we found that the size of the sliding window $L$ should be large enough to allow the splitter tree to capture useful information regarding the homogeneity of the data and small enough to not conceal statistical differences in the fragments of the data. In our implementation $L = 100$ and $S = 50$.

An effective splitter tree would be able to detect the statistical differences within the data and divide the file into different segments based on the contents of each segment. For example, if the given data was a document file that contains text and graphical charts, a good splitter tree would notice the change in the byte-series values from the text to the pictures and *vice versa*. Moreover, an ideal splitter tree might even detect different fragments within the same data type (e.g., a page full of blank lines within the text or white area in a picture).

### 3.2 Fileprint Tree

Unlike other techniques where files are processed as single units, our approach attempts to divide the file into smaller segments via the splitter tree and understand the type of each segment separately via the *fileprint* tree before making a final determination about the file type. The main job of the *fileprint* tree is to identify a unique signature for each file. These signatures are meant to be similar for files of the same type and different for files of different types. Hence, the outputs of the fileprint tree are easier to classify into different classes.

As illustrated in Fig. 3, the fileprint tree receives the segments identified by the splitter tree for each file and processes each segment individually. This corresponds to applying a function, $f_{\text{fileprint}}(S_i)$, to the data within each segment, $S_i$. The output

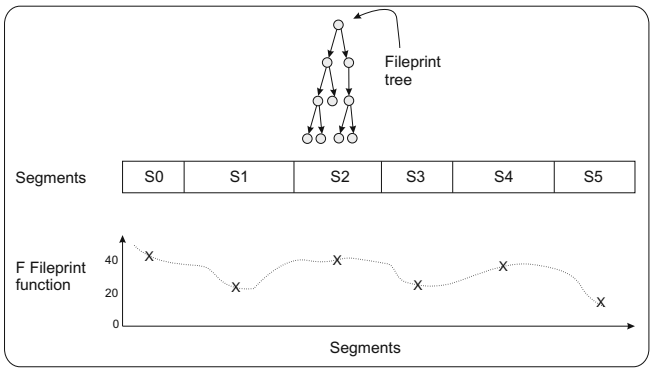


**Fig. 3.** The fileprint tree processes the segments identified by splitter tree (top). Its output produces a GP-fingerprint for the file (bottom).

of the program is a single number, $\alpha$, which is an abstract representation of the features of the data within the segments. Thus, the fileprint tree will produce a vector $v = (\alpha_1, \alpha_2, \cdots, \alpha_i)$. Each $v$ contains a series of abstracted numbers that describe the contents of a particular file. Each vector $v$ is referred to as a *GP-fingerprint*. A collection of vectors, $V = (v_1, v_2, \cdots, v_n)$, is obtained after execution of the fileprint tree with all the files in the training set.

### 3.3 Feature-Extraction Trees

The main job of the feature-extraction trees in our GP representation is to extract features (using the primitives in Table 1) from the GP-fingerprints identified by the fileprint tree and to project them onto a two-dimensional Euclidian space. Here we used two feature extraction trees. In future research we will study the benefits and drawbacks of using a different number.

Each feature-extraction tree represents a transformation formula which maps the original feature set (or more precisely the subset of the features used as terminals in the tree) into a single value output, which can be considered as a composite, higher-level feature. We use an unsupervised pattern classification approach on the outputs produced by the two feature-extraction trees to discover regularities in the training data files. In particular, we used K-means clustering to organise blocks (as represented by their two composite features) into groups. With this algorithm objects within a cluster are similar to each other but dissimilar from objects in other clusters. The advantage here is that the approach does not impose any constrain on the shape of the clusters.

Once the training set is clustered, we can then use the clusters found by K-means to perform classification of unseen data. Naturally, while we can tell K-means to group items in exactly $k$ clusters, being unsupervised, K-means has no way of knowing what each cluster is meant to represent. So, it might produce results that are not useful to classify the files in the training set. For example, at least in principle, K-means might find that text files naturally form two separate groups (judging from their two composite features).

So, how do we convince K-means to group things differently? Simple: we do not act on the K-means algorithm; we act on the composite features. That is, by using evolution,

we ask GP to come up with two feature-extraction trees that lead K-means to cluster the file fingerprints in the training set in such a way that all fingerprints in a cluster belong to the same file types and that different file types are associated to different clusters. At the end of evolution, K-means is able to distinguish file types based on their contents. The advantage of this approach is that it greatly simplifies classification. This is because evolution pushes feature-extraction trees to represent the data in such a way to optimise the performance of the classification algorithm. Here, we used K-means for its simplicity of implementation and its execution speed, but other techniques might work equally well.

### 3.4 Fitness Function

The performance of each individual is evaluated by measuring the classification accuracy of the training examples. Although the system uses a multi-tree representation where each tree has a particular function, this form of performance evaluation is sufficient to encourage each component of a program to perform the particular sub-task assigned to it to its best to achieve good performance in the difficult task of recognising file types.

Fitness is evaluated after performing the clustering of the outputs of the feature-detection trees using K-means. Our fitness evaluation is based on the quality of the clustering in terms of cluster homogeneity and cluster separation.

The homogeneity of the clusters is calculated as follows. As exemplified in Fig. 4, we count the members of each cluster, each data point in the cluster representing the GP-fingerprint of one file in the training set. Since we already know the content type for each fingerprint, we label the clusters according to the dominant data type. The fitness function rates the homogeneity of clusters in terms of the proportion of data points – GP-fingerprints – that are labelled as the file type that labels the cluster.

The system prevents the labelling of different clusters with the same file type even in the cases where the proportions in two or more clusters are equal. Using the information about the GP-fingerprints and their labels we can easily find the total number of data points that belong to the same data type. Any deviations from this optimal value due to clusters containing extra members should be discouraged. Thus, we use a penalty term in the fitness function to penalise extra members in the clusters.

More formally, the clusters homogeneity can be expressed as follows. Let $H$ be a function that calculates the homogeneity of a cluster and $CL_i$ be the $i^{th}$ cluster. Furthermore, let $k$ be the total number of clusters and $\lambda$ the penalty term. Then,

$$f_{\text{homogeneity}} = \frac{\sum_{i=1}^{k} H(CL_i) - \lambda}{k}$$

The homogeneity of the clusters is not the only measurement of the quality of the classification performed by K-means. Homogenous clusters with objects far apart within a cluster will extend the clusters boundary and may lead to inaccurate classification of unseen objects. Also, clusters that overlap are not suitable. Ideal clusters are separated from each other and densely grouped near their centroids. Therefore, we also measure and reward the separation of the clusters.

The formulation of the Davis Bouldin Index (DBI) proposed in [13] was used to measure cluster quality. DBI is a measure of the nearness of the clusters members to their centroids and the distance between clusters centroids. DBI can be expressed as follows.

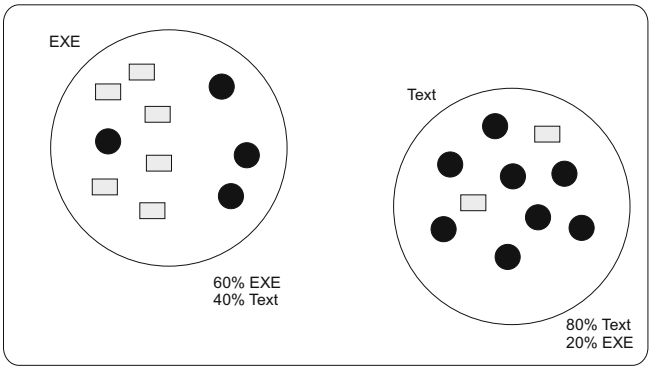


**Fig. 4.** Homogeneity measure of the clusters

Let $C_i$ be the centroid of the $i^{th}$ cluster and $d$ the $n^{th}$ data member that belongs to the $i^{th}$ cluster. In addition, let the Euclidian distance between $d^n$ and $C_i$ expressed by the function be $dis(d^n, C_i)$. Furthermore, let $k$ be the total number of clusters. Finally, let the standard deviation be denoted as $std()$.Then,

$$DBI = \frac{\sum_{i=1}^{k} std[dis(C_i, d^0), \cdots, dis(C_i, d^n)]}{dis(C_0, C_1, \cdots, C_k)}$$

A small DBI index indicates well separated and grouped clusters. Therefore, we add the negation of the DBI index to the total fitness in order to push evolution to separate clusters (i.e., minimise the DBI). So, the DBI is treated as a penalty value, the lower the DBI the lower penalty applied to the fitness. Thus, the fitness function is as follows:

$$fitness = f_{\text{homogeneity}} - DBI \tag{1}$$

A significant advantage with our method is that the approach does not impose any constraint on the shape of the clusters. Once the training set is clustered, we can then use the clusters found by K-means to perform classification of unseen data.

In the testing phase, unseen data goes through the three components of the evolved solution: blocks are produced by the splitter tree, the GP-fingerprint is obtained by the fileprint tree, and, finally, GP-fingerprints are projected onto a two-dimensional Euclidean space by the two feature-extraction trees. Then, these are classified based on the majority class labels of their K-nearest neighbours. We use a weighted majority voting, where each nearest neighbour is weighted based on its distance from the newly projected data point. More specifically the weight is $w = 1/distance(x_i, z_i,)$, where $x_i$ is the nearest neighbour and $z_i$ is the newly projected data point.

### 3.5 Search Operators

There are several options for applying genetic operators to a multi-tree representation. For example, we could apply a particular operator that has been selected (based on a predefined probability of application) to all trees within an individual. Alternatively, we

could iterate over the trees in an individual and, for each, select a potentially different operator. Another possibility would be to constrain crossover to happen only between trees at the same position in the two parents or we could let evolution freely crossover different trees within the representation.

It is unclear what technique is best. In [10] the authors argued that crossing over trees at different positions might result in swapping useless genetic material resulting in weaker offspring. On the contrary, in [2] suggested that restricting the crossover positions is misleading for evolution as the clusters are indistinguishable during evolution.

In preliminary experiments we tried all of these approaches and we learnt that a good way to guide evolution in our system is as follows. Let the $i^{th}$ individual of the population be denoted as $I_i$ and let $T_c^i$ be the $c^{th}$ tree of individual $i$, where $c \in \{splitter, fileprint, feature-extraction_x, feature-extraction_y\}$. The system selects an operator with a predefined probability for each $T_c^i$ . In the crossover, a restriction is applied so that splitter and fileprint trees can only be crossed over with their equivalent tree type. However, the system is able to freely crossover feature-extractions trees at any position.

## 4 Experimental Setup

Experiments were performed on various file types. The main aim of the experiments was to evaluate the performance of the algorithm and to assess the algorithms behaviour under a variety of circumstances.

The results presented in the following section were obtained by using the parameter settings illustrated in Table 2. Evolution halts when 30 generations have elapsed.

Experiments have been divided into four sets. Each set involved 10 independent runs (40 runs in total). In the first set we trained the system to distinguish between two different file types. We increased the number of file types to three in the second set of experiments, to four in the third set, while the last set included five file types. The files types that have been included in the experiments were selected because they are among the most commonly used files types. During the experiments, the algorithm has been trained to distinguish between similar files types (JPG and GIF or TXT and PDF). This allowed us to study the algorithm's ability in distinguishing the files based on their types rather than their contents. The corresponding training sets included 10 different files of each type. Several considerations were taken into account when designing the training set. The training set is processed many times by each individual in each generation. Thus, it has to be small enough to avoid over-fitting and yet big enough to contain enough examples to aid the learning process. Table 3 presents the contents of the training sets for each set of experiments.

To assess the system learning and generalisation we evaluated the accuracy of the evolved programs with a test set. The test set is composed of 30 different files for each type. Table 3 presents the contents of the testing cases for each set of experiments. The test sets are completely independent of the training sets. It should be noticed that the size of test set is bigger than the training set. Also, the test set included complex files such as EXE games, and large PDFs that contain figures and charts. This is to allow us to probe the generalisation capabilities of the evolved solutions.

**Table 3.** Training and test sets for the experiments

| | | Training set | | Test set | |
|---|---|---|---|---|---|
| *Function* | *File types* | *Total size* | *Number of files* | *Total size* | *Number of files* |
| 2 file types | JPG, GIF | 618 KB | 20 | 5.44MB | 60 |
| 3 file types | JPG, GIF, TXT | 987 KB | 30 | 7.9MB | 90 |
| 4 file types | JPG, GIF, TXT, PDF | 1.55 MB | 40 | 16 MB | 120 |
| 5 file types | JPG, GIF, TXT, EXE, PDF | 2 MB | 50 | 17.7MB | 150 |

**Table 4.** Test-set performance results. Numbers in **boldface** represent the best performance achieved.

| *Method* | *2 file types* | *3 file types* | *4 file types* | *5 file types* |
|---|---|---|---|---|
| Neural Networks | 58.33% | 66.67% | 74.17% | 39.33% |
| Bayes Nettworks | 50.00% | 66.67% | 83.33% | 48.67% |
| J48 | 51.67% | 51.67% | 74.17% | 48.67% |
| GP-fingerprint | **85.00%** | **88.90%** | **85.00%** | **70.77%** |

In order to evaluate our approach against other state of the art classification techniques we compared our results with standard Neural Networks [14], Bayes Network [14], and J48 decision trees [14] (a variant of C4.5). For these classification algorithms we used the implementation provided by WEKA [11]. We provided to these algorithms the same training sets and the same primitive sets as our GP system in order to obtain fair comparisons.[1] For each of the Neural Networks and Bayes Network systems we performed 10 different runs for each data set, as we did for our GP system. J48, being a deterministic algorithm, was only executed once for each data set.

## 5 Results and Analysis

Table 4 summarises the results of our experiments comparing GP with other techniques. For each non-deterministic algorithm we report the best results obtained within independent 10 runs. GP appears to outperform the other classification methods considered by a considerable margin. We believe that the good classification accuracy of our approach is largely attributable to the segmenting the data into smaller parts and obtaining fingerprints.

For all algorithms in we see that performance increases as the number of file types increases from 2 to 4. One might wonder why this happens: recognising two file types would appear to be an easier task than recognising three or four. To understand this, we need to consider that the data in Table 4 represent test-set performance. The reason why performance is lower in the two-file case than in the three- or four-file cases is over-fitting. As one can see in Table 3, the fewer the file types in a data set, the smaller

[1] Neural Networks and Bayes Network use deterministic learning models but initial networks are random.

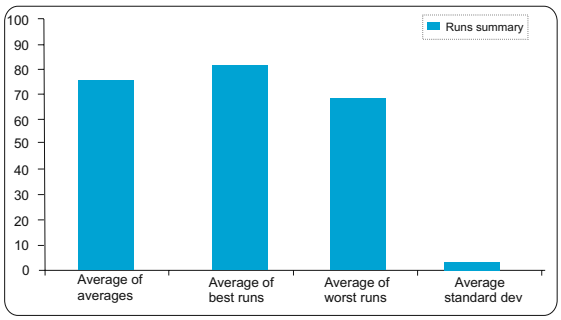


**Fig. 5.** Summary of results of 40 runs. Ordinates represent percentages.

the data set. Thus, it is easier for a learning algorithm to over fit training sets with 2 and 3 file types than the set with 4. As a result, the fewer the file types the worse the generalisation performance of the corresponding learning systems.

It should be noticed that the lowest performance for all classification algorithms shown in Table 4 occurred when classifying 5 file types. This is due to the fact that the complexity of the problem increases with the number of classes. However, GP performance degraded less than for other approaches indicating that our system may be more robust. The disadvantage of our approach, however, is training time: our GP system entails a learning process of several hours, while other classification techniques only consume a few seconds for the whole learning process. On the other hand, it has to be pointed out that the solution evolved by our system take only a few seconds to predict the files contents. So, they are not only accurate, but also entirely practical.

Although in Table 4 we reported best performance for all systems, including our GP one, in fact our system is remarkably reliable across runs. Fig. 5 summarises the results obtained in our 40 runs. We measured the quality of each experiment set (four sets, each set included 10 runs) by calculating the following statistics: the average classification accuracy across the test files, the corresponding standard deviation, and the best and worst classification accuracies in all runs. The first bar in Fig. 5 shows the average of the resulting four averages. The second bar in the figure is the average of the best evolved program in each set. The third bar represents the quality of the worst evolved program in each set. Finally, the last bar shows the average standard deviation. Note that the average *worst* performance of our GP system is better than the average *best* performance of the other systems in Table 4.

The low standard deviation and reasonably high average performance of our system suggests that one is likely to obtain accurate file-type prediction models within very few GP runs.

## 6 Conclusions

In this paper we have proposed a system based on genetic programming to evolve programs that can identify file contents without making use of any meta data. This is a novel application of GP.

The classification accuracies obtained by GP were far superior to those obtained by a number of classical algorithms from WEKA [11], namely artificial neural networks, Bayesian networks and J48 decision trees. While evolution is relatively slow with the large training sets required by this application, the resulting programs are entirely practical, being able to process tens of megabytes of data in seconds.

In the future we intend to extend the work to larger and more varied data sets and, hopefully, to turn the best solutions evolved by GP in public-domain stand-alone programs, which could perhaps be integrated in spam filters and anti-virus software.

## References

1. Erbacher, R.F., Mulholland, J.: Identification and localization of data types within large-scale file systems. In: SADFE 2007: Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering, Washington, DC, USA, pp. 55–70. IEEE Computer Society, Los Alamitos (2007)
2. Boric, N., Estevez, P.A.: Genetic programming-based clustering using an information theoretic fitness measure. In: Proceedings of the IEEE Congress on Evolutionary Computation CEC 2006, pp. 31–38. IEEE, Los Alamitos (2007)
3. Hall, G.A., Davis, W.P.: Sliding window measurement for file type identification. Technical report, Computer Forensics and Intrusion Analysis Group, ManTech. Security and Mission Assurance, Rexas (2006)
4. Haynes, T., Sen, S., Sen, I., Schoenefeld, D., Wainwright, R.: Evolving a team. In: Working Notes of the AAAI 1995 Fall Symposium on Genetic Programming, pp. 23–30. AAAI, Menlo Park (1995)
5. Karresand, M., Shahmehri, N.: File type identification of data fragments by their binary structure. In: Proceedings of the 2006 IEEE Workshop on Information Assurance, NY, pp. 140–147. IEEE Computer Society, Los Alamitos (2006)
6. Karresand, M., Shahmehri, N.: Oscar – file type identification of binary data in disk clusters and ram pages. In: Security and Privacy in Dynamic Environments, pp. 413–424. Springer, Boston (2006)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge (1992)
8. Li, W.-J., Stolfo, S.J., Herzog, B.: Fileprints: Identifying file types by n-gram analysis. In: Proceedings of the 2005 IEEE Workshop on Information Assurance, pp. 64–71 (2005)
9. McDaniel, M., Heydari, M.H.: Content based file type detection algorithms. In: HICSS 2003: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS 2003) - Track 9, Washington, DC, USA, p. 332.1. IEEE Computer Society, Los Alamitos (2003)
10. Muni, D.P., Pal, N.R., Das, J.: A novel approach to design classifiers using genetic programming. IEEE Transactions on Evolutionary Computation 8(2), 183–196 (2004)
11. U. of Waikato. Weka (July 2009), `http://www.cs.waikato.ac.nz/ml/weka/`
12. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (With contributions by J. R. Koza) (2008), `http://lulu.com`, `http://www.gp-field-guide.org.uk`
13. Sepulveda, F., Meckes, M., Conway, B.: Cluster separation index suggests usefulness of non-motor eeg channels in detecting wrist movement direction intention. In: IEEE Conference on Cybernetics and Intelligent Systems, pp. 943–947. IEEE Press, Los Alamitos (2004)
14. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, San Francisco (2005)