Genetic-Programming based Prediction of Data Compression Saving

Ahmed Kattan[1] and Riccado Poli[1],

[1] School of Computer Science and Electronic Engineering, University of Essex. Wivenhoe, United Kindom

Abstract. We use Genetic Programming (GP) to generate programs that predict the data compression ratio for compression algorithms. GP evolves programs with multiple components. One component analyses statistical features extracted from the files' byte frequency distribution to come up with a compression ratio prediction. Another component does the same but by analysing statistical features extracted from the files' raw ASCII representation. A further (evolved) component acts as a decision tree to determine the overall output (compression ratio estimation) returned by an individual. The decision tree produces its result based on a series of comparisons among statistical features extracted from the files and the outputs of the two prediction components. The evolved decision tree has the choice to select either the outputs of the two compression prediction trees or alternatively, to integrate them into an evolved mathematical formula. Experiments with the proposed approach show that GP is able to accurately estimate the compression ratio of unseen files thereby avoiding the need to run multiple compressions on a file to decide which one provide best results.

Keywords: Genetic Programming, Compression, Byte frequency distribution, Decision tree.

1. Introduction

Researchers in the compression field tend to develop algorithms that work with specific data types, taking advantage of any available knowledge regarding the data. Evidently, applying different compression algorithms to the same file will result in different compression ratios. However, with prior knowledge it is possible to match the data with the proper compression model. This is difficult to do if the nature and regularities of a given data file are not known as is the case for heterogeneous archives. Testing alternative compression algorithms to determine the best one to use is extremely time consuming when the given data is large (e.g., > 1GB). Applying a random compression model in this case might result in loss of efficiency with regards to storage space or it might even cause increase. Consequently, estimating the compression ratio when applying different compression models could be very advantageous in saving both computational resource and the time required to perform the compression.

Researchers have attempted to estimate data compression ratios for compression algorithms without the need to run the algorithms in question. Hus, [1], proposed an automatic synthesis of compression techniques for heterogeneous files. His approach focused on pigeon holing file types and forwarding each type to the proper compression model. The proposed approach modified the UNIX 'file' command and applied it on every 5KB of a file to determine the type of information it contains (e.g., text, graphics, executable).

In [2] Culhane proposed three measurements to predict the compression ratio of files when applying the Huffman coding or LZW. The three measures includes; i) standard deviation of the bytes, ii) standard deviation of the difference of consecutive bytes, and iii) standard deviation of the XORed value of consecutive bytes.

Recently we presented a lossless GP data compression system called GP-zip* in [3]. There we used Genetic Programming to learn the compressibility of different patterns in the data and match them with different compression models in such a way as to minimise the total size of the file. Although GP-zip* successfully matched unseen data with different compression models, it was not designed to give an estimation of the compression ratio.

In this paper we propose a new approach based on GP to generate programs that can predict data compression efficiency for different compression models. The aim is allow a rapid analysis of the data to determine which compression model is to be used and, hence, save the resources and time needed to run multiple algorithms.

2. The Methodology

Our system works in two stages: *i) Training*, where the system evolves mathematical formulas to predict the compression ratio of a particular compression model when applied to different training files, and *ii) Testing*, where the system is applied to unseen data.

From the point of view of an operating system or standard high-level programming language, the data to be compressed is normally treated as a sequence of elementary data units, typically bytes. What each unit represents depends on the file type. If the file is plain ASCII text, each unit will represent a character. If a file is an executable program, each unit may either represent an instruction (or most likely a fragment of an instruction) or some numeric or textual data. In files containing recordings of signals (e.g., sound) a unit (say a byte) will either represent a sample or part of a sample. In any case, the interpretation of the sequence of units contained in a data file entirely depends on what we know about that file and what our expectations are regarding the contents of that file. In most situations these are determined by the file's name and extension (although further information may also be available). Naturally, one can use such knowledge about a file to decide how to compress it, which is what most off-the-shelf compression algorithms do. However, when presented with unknown data (e.g., an archive that has been encrypted or an archive in a format unknown to the operating system) one cannot exploit this information.

Our system processes each file via two different representations. Firstly, each file is represented as a series of byte values (i.e., 0 - 255). Secondly, each file is represented as a byte frequency distribution (BFD). Sections 2.1 and 2.2 describe each representation in detail.

GP evolves programs with multiple component trees (see Figure 1). The system analyses each representation of the data independently via the two compression prediction trees. Then, it integrates them into a single evolved decision tree. The output of the decision tree is the estimated compression ratio. The decision tree produces its result based on a series of comparisons among statistical features extracted from the files and the outputs of the other evolved components. The evolved decision tree has the choice to either select the outputs of the compression prediction trees or alternatively, to integrate them into an evolved mathematical formula in an effort to improve the prediction. Section 2.3 will describe the decision trees' structure in detail.

GP has been supplied with a language that allows the extraction of statistical features out of the two data representations and then combines them into a single decision tree. Table 1 illustrates the primitive set of the system. Note that all three trees in the representation of an individual use the same primitives. The only exception in the decision tree (see section 2.3).

Table 1. Pprimitives set

Function	*Arity*	*Input*	*Output*
Median, Mean, Average deviation, Standard deviation, Variance, Skew, Kurtosis Entropy	1	List	Real Number
+, -, /, *,	2	Real Number	Real Number
Sin, Cos, Sqrt, log	1	Real Number	Real Number
<,<=,>,>=	4	Real Number	Real Number
List	0	N/A	Vector of Integers (0-255)

The system starts by randomly initializing a population of individuals using the ramped half and the half method [4]. The three standard genetic operators (crossover, mutation, and reproduction) have been used to guide evolution through the search space.

We let evolution optimise the three components during the training phase. The objective of the system is to build two statistical models and a decision tree that approximate the compression ratio for the files in the training set when applying a particular compression model. After evolution, we test the performance of the evolved components on unseen data.

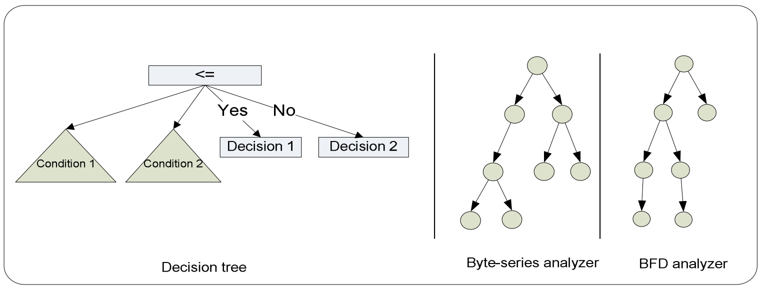

Fig 1: Individual within the system population

2.1. Analyzing the Byte-series

Each file is stored, within a computer, as a series of unsigned bytes. In our system we use this series as a reference interpretation for the data. In particular, we treat the stream of data as a signal, digitised using an 8-bit quantisation. Hence, each byte in a data file is treated as an integer between 0 and 255. Preliminary tests involving plotting such signals for different file types revealed that different data types often correspond to signals with very different characteristics, while similar data types share similar features.

The task is to evolve a non-linear function that extracts features out of the given byte-series that spot regularities and redundancy in the data stream. Naturally, in practice spotting such characteristics is not always straightforward. This depends on the nature of the given data stream. For example, it is easy to spot a regular pattern in an English text (e.g., ‘th’, ‘qu’), but complex in an executable file. Also, large byte-series might conceal some useful features only in some parts of the file.

2.2. Analyzing the Byte-Frequency Distribution

Preliminary experimentations showed that analyzing files’ byte-series alone does not provide enough information to build a generic compression estimation model. So, we also look at the Byte Frequency Distribution (BFD). BFD is defined as a histogram of the number of times that each character appeared divided by the total number of characters.

The basic concept of any compression model is to identify and remove the redundancy during the compression process. BFD contains features about the amount of available information in the data ‘entropy’ and also the symmetry in the data (from the point of view of characters frequencies). Thus, the BDF allows GP to reveal these characteristics and spot commonalities among different characters in the data stream.

The task of the second component of each GP individual is to evolve a function that extracts features from the BFD. The advantage of this representation is that it is easy to process, since it is a list of 256 values only, and contains valuable information regarding the data stream. A disadvantage is that it ignores the order of the data in the stream.

2.3. Decision Tree

A decision tree is a model that maps from the attributes of an item to a conclusion about its value [5]. Decision trees, leaves represent classifications and branches represent conjunctions of features that lead to classifications. Learning decision trees can be represented as an if-else-if series for human readability [5].

For the purpose of our for compression prediction system we customized the decision tree representation to fit our objective. As displayed in the decision tree on Figure 1, each comparison node (i.e., a node that contains a comparison condition such as, <, >, <= or >=) has four children. The first two represent conditions while the other two represent decisions. There are two types of condition trees and three types of decisions. The details of each type are as follows:

Condition types:

1. *Byte-series* condition tree: this is a component tree (see middle of Figure 1) that extracts features from the given Byte-series representation and abstracts them to a single number.

2. *BFD/Byte-series Outputs* condition tree: this is a component tree that integrates the outputs of the Byte-series analyzer tree and the BFD analyzer tree into a mathematical formula.

Decision types:

1. *BFD* Decision: is the output of the BFD analyzer tree.
2. *Byte-series* Decision: is the output of the Byte-series analyzer tree.

3. *BFD/Byte-series* Decision: is similar to *BFD/Byte-series Outputs* condition. It is an evolved tree that integrates the outputs of the Byte-series analyzer tree and/or BFD analyzer tree into single mathematical formula.

The output of the decision tree is the estimated compression ratio. The decision tree produces its result based on a series of comparisons among statistical features extracted from the files and the outputs of the Byte-series analyzer tree and/or the BFD analyzer tree. The evolved decision tree has three choices: i) select the output of the Byte-series analyzer tree, ii) select the output of the BFD analyzer tree or iii) integrate both the Byte-series analyzer and the BFD analyzer trees into a mathematical formula and use that to produce the output.

2.4. Genetic Operators

We used crossover, mutation and reproduction. Naturally, the genetic operators take the multi-tree representation of the individuals into account.

There are several options in applying genetic operators to a multi-tree representation. Firstly, we could either apply a particular operator that has been selected to all trees within an individual or select a potentially different operator for each component. Secondly, we could constrain crossover to occur only between homologous component trees or not. It is unclear what technique is best. In [6] the authors argued that crossing over trees at different positions might result in the swapping of useless genetic material resulting in weaker offspring. On the contrary, in [7] the authors argued that restricting the crossover positions is misleading for evolution as the features are indistinguishable during evolution.

After experimenting with a variety of approaches we settled for the following. Let the T_c^i be the *c*-th tree of individual *i,* where $c\epsilon\{$*Byte-series analyzer*, *BFD analyzer*, *Decision tree*$\}$. The system selects an operator with a predefined probability for each T_c^i. Thus, offspring can be generated by using more than one operator.

In the crossover, as each component has a particular task, only homologous components are allowed to cross. Also, the system has to take the structural constrains of the Decision tree into consideration and ensure its syntax is maintained. The system crosses condition branches with condition branches from corresponding trees and decision branches with decision branches from corresponding trees.

2.5. Fitness Function

As mentioned previously, each individual has a multi-tree representation, where one tree is used to analyze the byte-series, another tree is used to analyze the BFD and a third tree is used to decide which is the most accurate prediction of the compression ratio. Thus, there are three different objectives for the system. The first is to optimize the performance of the byte-series analyzer tree; the second is to optimize the performance of the BFD analyzer tree, and the third is to optimize the decision maker's performance.

We look at this as a multi-objective problem with three fitness functions. Each fitness function is measuring the quality of one component. The system randomly selects a fitness measurement each time it produce a new individual. In this way evolution is forced to jointly optimise all objectives.

The fitness function for each component is simply the average of the absolute difference between the estimated compression ratio and actual achieved compression for all files in the training set as we explain below.

The fitness of the BFD analyzer tree can be expressed as follows: let the output of the BFD-tree be denoted as *BFD(file$_n$)*, where *file$_n$* is the nth file in the training set. Furthermore, let *C(y, file$_n$)* be the compression saving of *file$_n$* when applying the compression model *y*. Thus,

$$\text{BFD-tree Fitness} = \frac{\sum_{i=1}^{n} |BFD(file_i) - C(y, file_i)|}{n}. \quad \textbf{(1).}$$

The fitness of the byte-series analyzer tree can be expressed as follows: let the output of the byte-series tree be denoted as *BS(file$_n$)*. Thus,

$$\text{Byte-series tree Fitness} = \frac{\sum_{i=1}^{n} |BS(file_i) - C(y, file_i)|}{n}. \quad \textbf{(2).}$$

Finally, the fitness of the Decision tree can be expressed as follows: let the output of the Decision-tree be denoted as *DT(bs ,bfd, file$_n$)*, where *bs* is the output of the byte-series analyzer tree, and *bfd* is the output of the BFD analyzer tree. Thus,

$$\text{Decision-tree Fitness} = \frac{\sum_{i=1}^{n} |DT(bs, bfd, file_i) - C(y, file_i)|}{n}. \quad \textbf{(3).}$$

Thus, a GP individual's quality is defined by its ability to identify statistical features for the data stream and predict their compressibility when applying a particular compression model.

2.6. Training and Testing

The system extracts knowledge concerning the features of the data and their relationships with the performance of a particular compression algorithm during a training phase (a GP run).

Several factors have been considered while designing the training set. Firstly, the training set has to contain enough diversity of data types in order to ensure the generality of the system. Secondly, the system will process the training set many times for each individual in each generation. Thus, the size of the training set should be small enough to maintain time-efficient training, but should also be large enough to be a representative set of data types the compression ratio of which is likely to be estimated by the users of the system. In addition, it is essential to avoid over-fitting the training set.

Table 2 presents the data types within our training set. The training set contains 15 different file types within 26 files for a total of 5.14MB. It should be noticed that the training set is completely independent of the test set.

GP's output at the end of the evolution consists of a byte-series analyzer tree, a BFD analyzer tree and a decision tree estimating compression ratios based on the other components. Because GP is stochastic, the user needs to run the system several times until it achieves adequate performance on the training set. Testing involves processing unseen files using these trees.

Table 2. Training files.

Files' types	*Total Size*
pdf, exe, C++ code, gif, jpg, xls, ppt, mp3, mp4, txt, xml, xlsx, doc, ps, ram	5.14 MB
Total number of files	**26**

Table 3 presents a list of file types that have been used to measure the algorithm performance. The test set contains 19 different data types within 27 files. It contains some file types similar to those used in the training set, while others are different data types to which the algorithm has not been exposed during training. Details regarding the algorithm's performance are given in the next section.

Table 3. Testing files.

Files' types	*Total Size*
Tif, jpg, bmp, accdb, xml, c++ code, txt, mht, doc, docx, ppt, pdf, exe, msi, wmv, flv, mp4, mp3, ram	67.9 MB
Total number of files	**27**

3. Experiments

The main aim of the experiments was to investigate the performance of the algorithm and to assess the algorithm's behaviour under a variety of circumstances.

The approach has been tested to predict the compression ratio for the files in the test set for the following compression algorithms: Prediction by Partial Matching (PPMD) [8], Arithmetic coding (AC) [9] and Boolean Minimisation [10]. These algorithms were chosen because they belong to different categories of compression algorithms (i.e., AC is a statistical based coding, BooleanM is a dictionary based coding and PPMD is an adaptive statistical coding).

The experiments presented here were performed using the following parameter settings: a population of 200 individuals, 40 generations, a crossover probability of 90%, a mutation probability of 5%, tournament selection with tournaments of size 5 and a maximum tree depth of 10.

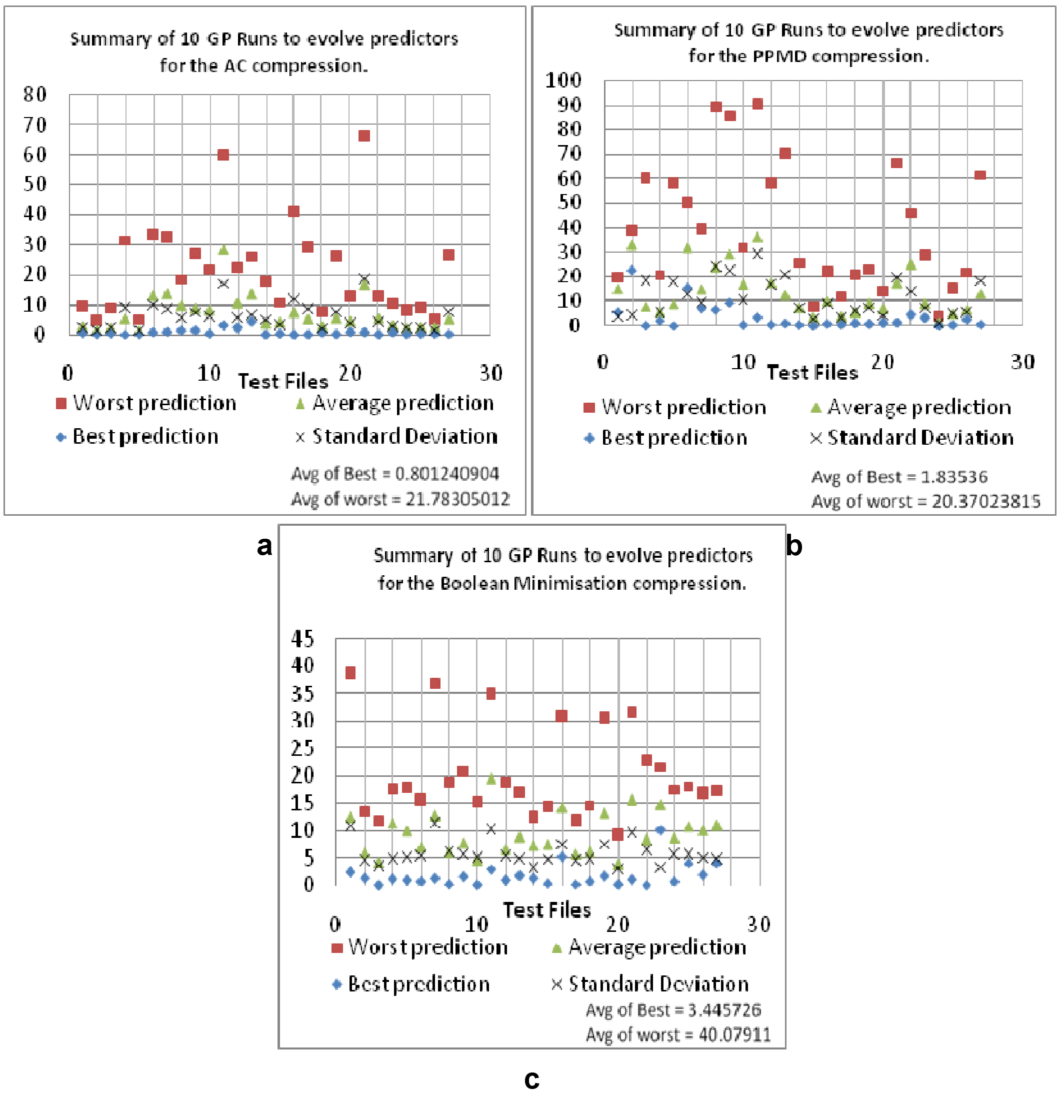

Fig 2: Summary of GP runs.

a) Summarize 10 GP runs to evolve predictors for the AC compression. b) Summarize 10 GP runs to evolve predictors for the PPMD compression c) Summarize 10 GP runs to evolve predictors for the Boolean Minimisation compression.

Table 4. Performance Comparison (Seen file types vs. Unseen file types)

Compression Model	*Avg .prediction error for trained files types in all runs*	*Avg .prediction error for untrained files types in all runs*
AC	5.95	7.31
PPMD	11.96	16.98
BooleanM	9.40	9.77

Figures 4a-c summarise the results of the experiments when evolving predictors for AC, PPMD and Boolean Minimisation (we performed 10 independent GP runs for each compression model). The graphs plot the best, worst and average prediction error for each file. Prediction error is measured as the absolute difference between the actual compression ratio and the estimated compression ratio (expressed as a percentage). Also, the standard deviation of the achieved predictions for each file in all runs is recorded. Each figure shows also the average of the best and worst predictions.

As one can see the average of the best achieved prediction errors is very small (with values ranging from 0.8% to 3.4%). The small standard deviation with the reasonable predictions average indicates that our system is likely to produce accurate models within a few runs.

As we mentioned before, the test set contained some files that consist of data types, to which the algorithm has not been exposed during training. Table 4 illustrates the average achieved prediction errors for the seen file types and the unseen file types.

Generally, the algorithm gives slightly less accurate results for those types which it had no prior experience. Nonetheless, the achieved predictions when the algorithm deals with new files types are satisfactory. So, the algorithm must have learnt some general knowledge which can be used in different situations.

Each component in an individual has a particular task, as explained previously. The final output is produced by the decision tree using the estimates constructed by the two other components. Hence, it is interesting to study how the decision tree integrates such information.

As we mentioned before the evolved decision tree has three choices: select the output of the Byte-series analyzer tree, select the output of BFD analyzer tree or integrate both Byte-series analyzer and BFD analyzer trees into a mathematical formula. Thus, if the decision tree selected the closest estimate to the actual compression ratio then we count that as a right decision. If it chose to return the estimate of the less precise component, we count it as a wrong decision. If the decision tree decided to integrate the Byte-series and BFD analyzers into a mathematical formula producing a more accurate estimate we count that as an improved decision.

Table 5 shows the proportion of *correct/wrong/improved* decisions for the decision tree in all 30 runs. Decision trees were able to select the correct compression estimation in most of the cases. Table 6 shows how often the BFD and Byte series trees were used to produce the right estimation for the compression ratio. Both components have been used to estimate the compression ratio.

Table 5. Decision tree performance

Decision	*Percentage*
Improved decision	7.05%
Right decision	61.78%
Wrong decision	31.17%

Table 6. Decision tree- Right decisions statistics

Decision	*Percentage*
BFD tree	73.87%
Byte series tree	26.13%

We mentioned previously that the Byte-series analyzer tree and the BFD analyzer tree are used as primitives in the decision tree. One might ask: why don't we evolve a single decision tree with all its branches instead of separating its components and evolving them individually? We tested also this alternative approach (using the same parameter settings for both systems). For space limitations we are unable to report full results. However, results with a single-tree representation were much less satisfactory.

As mentioned previously, our aim is to estimate the compression ratio via different compression modules to save both computational resource and the time required to perform a compression. Table 7 reports a comparison between the times needed to compress all files in the test set (67.9 MB) against the time needed to predict their compressions. It is clear that our approach is faster than performing the actual compression. This is no surprise, as compression algorithms involve a lot of I/O operations while our approach only scans the file and extract statistical features that correlate with the compression ratio.

Table 7. Compression time vs. Prediction time

Compression algorithm	*Compression time*	*Prediction time*
PPMD	148 seconds	62 seconds
AC	52 seconds	45 seconds
BooleanM	3003 seconds	65 seconds

4. Conclusions

In this paper we proposed to use GP to evolve predictors of the compression ratio achievable with three well-known compression models when applied to new files without the need to run the actual compression model.

The proposed approach attempts to predict the compression saving for the data via two different interpretations; i) looking at a file's byte-series and ii) considering a file's bytes frequency distribution. Each interpretation is used by a separate tree within our representation in conjunction with a collection of statistical measures. We require the two trees to predict as best as possible the compression ratio achievable when applying a particular compression model. A third component of the representation, a decision tree attempts to distinguish the most accurate of the two predictions and if necessary integrates them into an even better prediction. Evolution is guided by a single fitness measure (prediction error), which is applied randomly to one of the tree components in the representation. This forces GP to evolve accurate predictors in the first two components but in such a way that the third component can easily understand which predictor is more accurate, thereby effectively performing a kind of multi-objective optimisation.

Results are very encouraging, in the sense that a good prediction accuracy has been achieved on a large test set, both on seen and unseen data types, and with three compression models. We also found that separating the components of the decision tree and evolving them individually was advantageous in comparison with the standard method of evolving single decision trees.

Although the proposed technique has achieved good results, the performance depends on the knowledge GP acquires during evolution. Thus, users need to select the training set according to their needs. For example, a user interested in DVD production might want to train the algorithm to predict the compression saving for different types of videos or audios.

This research can be extended in many different ways. In the future we will investigate the use of further interpretations for the raw data in files (e.g., 2- and 4-bytes). Also, using primitives implementing High Order Statistics functions is likely to provide further improvements to the system's performance.

The disadvantage of the current realisation is that the evolved prediction models work only with particular compression algorithms. Thus, the user has to evolve a prediction model for each compression algorithm of interest. In future research we will try to create a single prediction model that works well with different compression algorithms.

References

[1] William H Hsu and Emy E. Zwarico, "Automatic Synthesis of Compression Techniques for Heterogeneous Files," *SOFTPREX: Software–Practice and Experience*, vol. 25, 1995.

[2] William Chlhane, *Statistical Measures as Predictors of Compression Savings*, The Ohio State University, Department of Computer Science and Engineering, Honors Theses, 2008.

[3] Ahmed Kattan and Riccardo Poli, "Evolutionary lossless compression with GP-ZIP*," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA, 2008, 2008, pp. 1211-1218.

[4] Riccardo Poli, William B. Langdon, and Nicholas McPhee, *A field guide to genetic programming.*: http://lulu.com, 2008.

[5] Tom M Mitchell,. McGRAW-HILL International Editions, 1997, ch. 3.

[6] Durga Prasad Muni, Nikhil R. Pal, and Jyotirmoy Das, "A novel approach to design classifiers using genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 2, pp. 183- 196, 2004.

[7] Neven Boric and Pablo A Estevez , "Genetic Programming-Based Clustering Using an Information Theoretic Fitness Measure," in *IEEE Congress on Evolutionary Computation*, 25-28 September 2007, pp. 31-38.

[8] John G. Cleary and I.H Witten, "Unbounded length contexts for PPM," in *Data Compression Conference*, Snowbird, UT, USA, 28-30 Mar 1995, pp. 52-61.

[9] Ian H. Witten, Radford M. Neal, and John G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520-541, 1987.

[10] Ahmed Kattan, *Universal Lossless Data Compression with built in Encryption*, Master Thesis, Ed.: University of Essex, 2006.