2012 IEEE 27-th Convention of Electrical and Electronics Engineers in Israel

FPGA-based data compressor based on Prediction by Partial Matching

Joel Ratsaby*, Vadim Sirota†
Department of Electrical and Electronics Engineering
Ariel University Center of Samaria
Ariel 40700, ISRAEL
Email: vadim.sirota@gmail.com * Email: ratsaby@ariel.ac.il †

***Abstract*—We design and develop a data compression engine on a single FPGA chip that is used as part of a text-classification application. The implementation of the prediction by partial matching algorithm and arithmetic coding data compression is totally in hardware without any software code. Our design implements a dynamic data structure to store the symbol frequency counts up to maximal order of 2. The computation of the tag-interval that encodes the data sequence in arithmetic coding is done in a parallel architecture that achieves a high speedup factor. Even with a relatively slow 50 Mhz clock our hardware engine performs more than 70 times faster than a software-based implementation in C on a CPU running on a 3 Ghz clock.**

Keywords: Data compression, FPGA, parallel architecture

I. Introduction

Text or document classification is a problem studied in the area of information retrieval [1], text-mining [2], pattern recognition [3] and machine learning [4, 5]. In a typical scenario there is a corpus of documents each labeled according to one of M different categories, for instance, email messages in the PC's in-box divided into the categories 'work', 'personal', 'research', 'spam'. Given a new document (for instance, a new incoming email message) the problem is to classify it into one of the existing categories automatically by the PC. A main challenge is to automatically learn to classify accurately from a given training data which is typically noisy. For instance, suppose a document is represented as a 'bag-of-words' from some fixed dictionary with no reference to their specific order in the document and no representation for grammatical rules. The importance of a word in a document is represented by the TFIDF formula [1]. The document's context is represented according to the relative word frequencies and the frequencies with respect to the whole document corpus. As ubiquitous and common as it is in practice, this simple representation can lead to classification errors. For instance, if a document contains the word 'war' then it is more likely to be in the category of 'military' than in 'electronics'. But there is still a non-zero chance that a document that contains the word 'war' is in category 'electronics'. Hence the Bayes' optimal classification error is non-zero and this makes the problem of learning to classify more challenging. The majority of research in this area focuses on improving algorithms and representations such that the accuracy of classification remains close to Bayes' error. The more we enrich documents' representation by introducing features such as words that are more relevant in the domain or finding ways of coding the grammatical structure that appears in the written language (available from knowledge of the specific domain of the problem) the more likely that the classifier will achieve lower errors.

Recently there has been work in the direction of feature-free representation. This means that no feature extraction is necessary but rather a document is used in its original raw format (for instance, the original file containing the text). The idea is based on the so-called information-distance [6]. It is a distance function between two entities x, y which measures their dissimilarity based on their Kolmogorov complexity [7]. In its more practical version it is called the Normalized Compression Distance (NCD) and is defined as follows [8]: given two binary strings x and y which are not necessarily of the same length, denote by xy their concatenation. Let $C(x)$ denote the length of the compressed string x using some compression algorithm [9] such as Lempel-Ziv [10] or PPM [11]. The distance between x and y is defined as

$$NCD(x,y) \quad = \quad \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}. \qquad (1)$$

The main advantage of using this distance is that it is parameter-free and model-independent. It requires no prior domain knowledge. One just needs to compress the two document files (viewed as binary strings) x and y and their concatenation and then evaluate their distance $NCD(x,y)$. There is no need for any kind of feature extraction such as word-frequencies in a bag-of-words representation or extracting grammatical features or semantic knowledge from the documents corpus. A similar approach based on compression has been shown to succeed in classification problems of a wide spectrum of data types [12, 13, 14, 15, 16], for instance, media content (music, video, voice), human sketches [12], image[14, 16], genome categorization, remote sensing data [13], prediction of binary sequences [15]. The computationally intensive part of evaluating this distance is doing the compression. In general, running a compression algorithm in software may take several seconds even on a state-of-the art computer which for some real-time applications is unacceptable. For the NCD one only needs to use a compressor and there is no need for decompressing.

In [17] a parallel hardware architecture was designed specifically for doing arithmetic coding compression in order to compute the NCD. The results there indicate that there is a significant advantage in computational time for evaluating this distance on a single FPGA chip relative to doing it in software.

In the current paper we expand on [17] by adding another layer of coding (on top of arithmetic coding) that is based on *prediction by partial matching* (PPM). This method achieves among the best compression rates on text-based data. It is based on parametric estimation of the context-based conditional probability of appearance of symbols. We describe it in the next section.

II. Context based compression

In context-based compression [9] the idea is to code the probability of the next symbol conditioned on the previously-seen values of

symbols in the text stream. The more skewed this conditional probability, the higher the compression rate. The most common way to represent this probability is to examine the history of the symbol sequence and estimate the probability distribution of the next symbol given a fixed number of previous symbols. This number is called the *context order* and is typically fixed in advance. For instance, in the word 'probability' the first-order context for the letter a is b since it is the single symbol before a. The second-order context for a is ob since it is the two symbols preceding a. In general, aßs we increase the order the probability of the occurence of a will become more skewed, that is, further from the uniform probability. This in turn reduces the number of bits required to encode a. Thus ideally one would like to use as high order as possible for representing the probability of occurence of a symbol. The disadvantage is in the amount of memory needed since the number of possible contexts grows at an exponential rate with respect to the order.

The PPM algorithm [9, 11] solves this exponential storage problem by keeping only the counts of those contexts that actually appeared in the data. This is a much smaller number than the number of all possible contexts. PPM incrementally updates these counts as it reads the text sequence that is to be compressed and acts in a greedy manner, always trying to use the conditional probability for a symbol with respect to the largest order context (the largest order is fixed in advance). If the symbol to be encoded did not appear in the data within a context of that size then the algorithm encodes an 'escape' symbol and attempts to use the next smaller context. This process continues until either we find a context of some size (greater than or equal to zero) within which the symbol appeared or that the symbol did not appear in any context. In the latter, the algorithm uses a uniform probability ($1/M$ probability where M is the number of possible symbols in the alphabet) and we refer to it as context of order -1 (written Order_-1). Concerning the encoding of the escape symbol, there are several implementations of the algorithm. We choose the variant known as PPM-C which means that the count assigned to the escape symbol is the number of distinct symbols that have occured in that context (denoted by *define_bytes)*. The probability of a symbol is the normalized value of its count. In the next section we describe arithmetic coding which is used to encode each symbol based on its probability.

III. Arithmetic coding

Arithmetic coding is a procedure which encodes a given sequence of symbols, for instance a text document, as a binary sequence whose length is shorter than the number of bits used in the original symbol representation. The idea is to represent the symbol sequence by a unique fraction which is denoted as *tag* (it is contained in the unit interval $[0, 1]$). The binary representation of this fraction becomes the binary codeword of the sequence. The cumulative probability distribution (cdf) of the source of the symbol-sequence plays a major role in coding. The basic version of the algorithm (which we have implemented) uses just the zeroth-order cdf. Each symbol is uniquely associated with a subinterval corresponding to one of the discontinuous steps of the cdf. We start with the complete unit interval which by definition contains the tag. When a new symbol in the sequence is received, its unique interval is used to rescale the current interval that contains the tag. In general, this procedure can be described by a recursive algorithm for computing the lower and upper boundaries of the tag interval at time n (the time when symbol

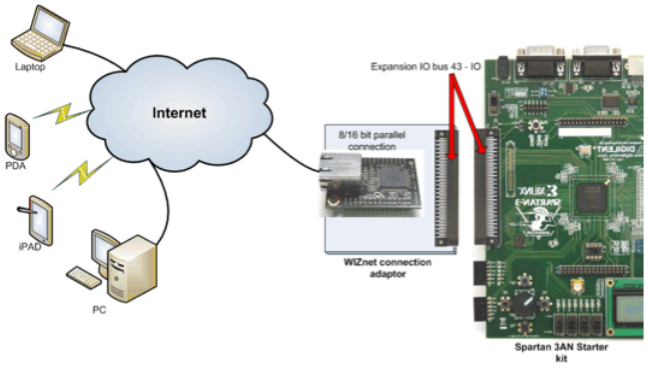

Fig. 1. Ethernet connection

x_n arrives) :

$$\begin{aligned} l^{(n)} &= l^{(n-1)} + \left(u^{(n-1)} - l^{(n-1)}\right) F(x_n - 1) \\ u^{(n)} &= l^{(n-1)} + \left(u^{(n-1)} - l^{(n-1)}\right) F(x_n) \end{aligned} \quad (2)$$

where $F(x_n)$ denotes the value of the cdf at x_n and $F(x_n - 1)$ denotes its value at the symbol to its left (in the order of the alphabet). As the text sequence becomes long the interval length decreases to a very small fraction. To avoid the need for infinite precision there is a scaling procedure which keeps the current interval at time n always at a minimum size. Moreover, in an integer-precision implementation it is necessary to estimate the cdf values by a fixed number of 2^m discrete levels. A finite table of cumulative counts are computed based on the symbol-sequence and is used as an estimate of the true cdf. A new pair of equations based on this discrete estimate replaces (2) where now $l^{(n)}$ and $u^{(n)}$ are m-bit binary words. An incremental procedure exists for generating the binary encoding that forms the compressed version of the original symbol-sequence. It also does the scaling. This way at each time n, based on the to most significant bits of $l^{(n)}$ and $u^{(n)}$ one determines the output bit. Repeating this incremental procedure from beginning to end produces the full codeword for the input text.

IV. System implementation

We developed a special-purpose system that acts as a high-speed compression server. A client sends a file (or any character stream) to the server who compresses the file and reports back the length (number of bits) of the compressed encoding (Figure 1) . Figure 2 shows the system overview. The micro-controller (PicoBlaze) is used only for transfer of the text stream from the ethernet interface into the compressor and for reporting the compressed size in the end of the compression. The computation of the NCD formula is done at the client side based on the reported compressed length from the server.

A. PPMU

The compressor itself is programmed all in hardware in VHDL with two main components: a PPM unit (PPMU) and an arithmetic coding unit (ACU). The text sequence to be compressed consists of consecutive bytes each an eight bit binary ASCII code which are transfered to the card via an ethernet interface. The counts of symbols are kept in data structures for each of the orders of context where maximal order is 2. We denote them as Order_2, Order_1, Order_0 and Order_-1. Order_-1 is the uniform distribution (described above). Order_0 means that the probability function of a symbol is unconditional. Order_1 means that it is conditioned on a

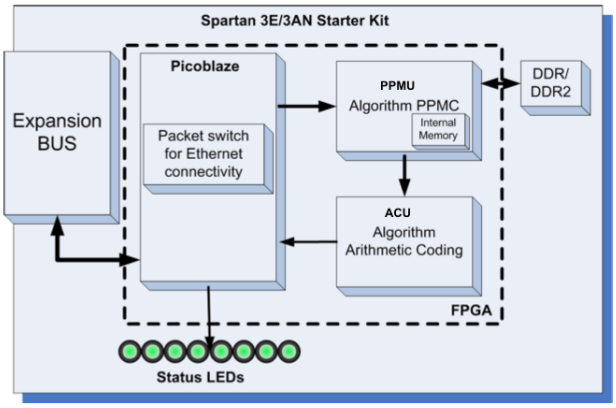

Fig. 2. System architecture

single preceeding symbol and Order_2 means it is conditioned on two preceding symbols. As an example, consider the following text sequence $bacabac$ then

Order0 (symbol,count)	Order1 (symbol,context,count)	Order2 (symbol,context,count)
b,2	a, b,2	c,ba,2
a,3	c,a,2	a,ac,1
c,2	b,a,1	b,ca,1
		a,ab,1

From (2) the cummulative probability $F(x)$ of a symbol x is needed in order to compute the lower and upper boundaries of the tag interval. Denote by n_i the number of times that symbol i occurs in a sequence of length $TotalCount$ then for a symbol k we can express $F(k) = \frac{1}{TotalCount} \sum_{i=1}^{k} n_i$. Define by $cumCount(k) = \sum_{i=1}^{k} n_i$ then (2) can be expressed in terms of these cummulative counts as follows,

$$\begin{aligned} l^{(n)} &= l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times cumCount(x_n - 1)}{TotalCount} \right\rfloor \\ u^{(n)} &= l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times cumCount(x_n)}{TotalCount} \right\rfloor - 1 \end{aligned} \quad (3)$$

The value of $cumCount(x_n - 1) = cumCount(x_n) - count(x_n)$ hence the only information needed by the ACU to compute the interval for the n^{th} symbol x_n in the sequence is $cumCount(x_n)$, $count(x_n)$ and $TotalCount$. All three values depend on the specific context in which the symbol x_n was found to occur. For instance, in Order_1, if the current symbol x_n follows the symbol z which did not occur up until time n then for this context z the $TotalCount = 0$.

According to the PPM algorithm we need only keep the counts of symbols that occured in the text sequence. We keep these counts in tables that are distributed in various memory units (internal and external to the FPGA). We only store the $count(k)$ of a symbol k so in order to get $cumCount(k)$ we sum all counts from the first symbol $(0)_H$ up until symbol k. This way we do not need to store the cumCounts. We store the $TotalCount$ in order to save time by not needing to sum up all the counts each time a new symbol gets encoded. Note that $TotalCount$ includes the count of the escape symbol which as mentioned above is *Defined_bytes*.

For Order_0 we have a single counter that holds the $TotalCount$. For Order_1 we have 256 counters each holding the $TotalCount$ that corresponds to one of the 256 possible contexts in order 1. These 256 counters are stored in a separate dedicated memory unit in order to parallelize memory access. In Order_2, for each context we keep the $TotalCount$ in a memory address in a hashtable entry that also points to a linked list that contains symbols that have already occured in the sequence in this particualr context. Since there are too many possible contexts of order 2 we only keep those contexts that occured in the sequence.

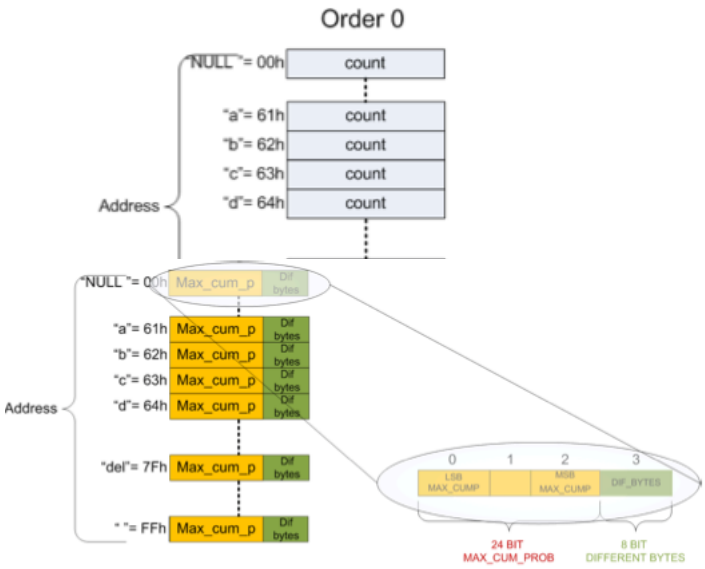

Fig. 4. Order 1, $TotalCount$ and *Def_Bytes* (FPGA internal RAM)

Let us describe the data structures used to store the counts for each order. We start with the simplest order *Order_-1.*

1) Order_-1: Order -1 is used when the current symbol did not appear up until the current symbol. The probability for encoding the symbol is according to the uniform distribution. Since there are 256 possible symbols this probability is $1/256$. We represent this by keeping a count of 1 for each symbol and a cummulative count $cumCount(k) = k$, for each symbol $k \in \{0, \ldots, 255\}$. Since this information never changes there is no need to store it in memory. Given a symbol x_n we pass to the ACU the following values $count(x_n) = 1$, $cumCount(x_n) = x_n$, and the $TotalCount = 256$.

2) Order_0: Order 0 is used when the symbol to be encoded appeared at least once before in the sequence but occurs in a context which has not appeared in the sequence in neither order 1 nor 2. In this case the probability of the symbol is read from a one dimensional table with 256 entries where each entry is a 32 bit number representing the count of a symbol as displayed in Figure 3. The count for the escape symbol is called *Define_bytes* which holds the number of distinct symbols that appeared in the sequence. Its inverse value is used as the probability of the escape symbol. The $TotalCount$ has 24 bits and holds the total number of symbols (not just distinct) that appeared.

3) Order_1: In Order 1 there are 256 possible contexts so the data structure is implemented as a two-dimensional table based on $256^2 = 65536$ cells each 16 bits wide for holding the count as shown in Figure 5

Instead of using the internal block RAM we implement this table in DDR RAM on the Xilinx Spartan-3AN board which has four DDR banks. We use a Micron DDR 512Mb chip organized as

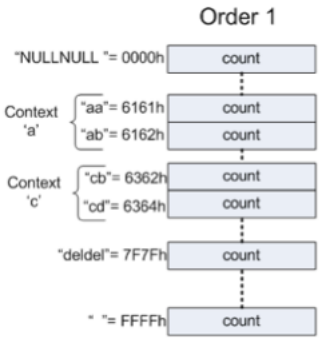

Fig. 5. Order 1, symbol counts for all contexts (external RAM bank #1)

Fig. 6. Order 2, entry in Hashtable (external RAM bank #2)

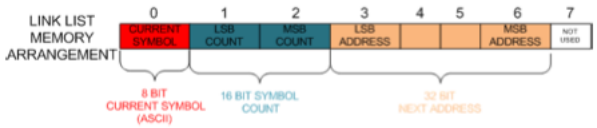

Fig. 7. Order 2, a cell in the linked list (external RAM bank #3)

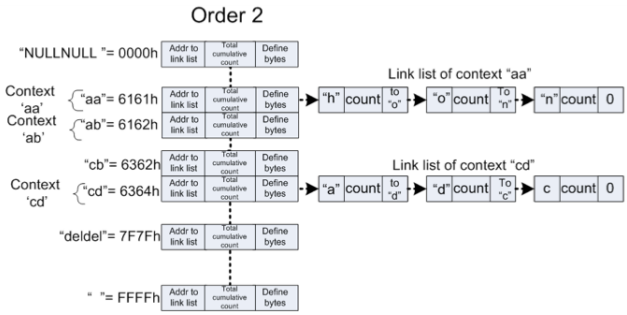

Fig. 8. Order 2, combined data structure

32Mx16. The $TotalCount$ (which is referred to also as *Max_cum*) and *Define_bytes* (referred to also as *Dif_bytes*) for each context are stored in 256 cells in a separate RAM block internal to the FPGA (Figure 4). Each of the 256 cells has 24 bits for the $TotalCount$ and 8 bits for the *Define_bytes*. We use bank# 1 for the table (Figure 5). This combination of internal and external RAM allows concurrent access to both memory modules and speeds up the Order 1 data processing.

4) Order 2: The number of possible contexts in Order_2 is $256^2 = 65536$ and each such context can contain 256 symbol counts hence reaching up to 16Mbyte which is more than available in the internal RAM of the FPGA. To overcome this we use a dynamic memory structure in the form of a linked list fully implemented in VHDL. This way we only store counts of symbols that occured and only for contexts that occured in the sequence. Hence the data structure for Order_2 needs to reside in RAM external to the FPGA. We devote two external banks, bank #2 and bank #3 of the DDR RAM. Bank #2 implements a hash table with 256^2 cells each of which contains an entry of 64 bits as shown in Figure 6. The address of an entry in this table is obtained by mapping a key into an index. The key consists of the concatenation of two symbols (the value of the context). For instance, if the current symbol is c and the last two symbols are ab then the key is the 16 bit code $0x6162$ (in hex) which is the concatenation of the ASCII codes of a and b. We use the following formula to obtain the index (hash) for a key which is the concatenation of the ASCII codes of two symbols k and j,

$$PPMC\_hash(k, j) := ((j \ll 8) + k) \,\&\, 0x0000FFFF$$

where $j \ll 8$ means shift left the value of j by 8 bits. The first element of an entry (Figure 6) is 32-bit address of the start of a linked list (in Bank #3) that contains the symbol counts for that context. The second celement is *Max_CUM_PROB* which is a 24-bit value of the $TotalCount$. The *Dif_Bytes* is an 8-bit representing the *Define_bytes* for this context.

Bank #3 holds the actual linked lists of the contexts that occured. A linked list holds the symbol counts that occured in this context. It may grow each time a new symbol arrives. In each cell of the linked list we store a symbol k (8-bit ASCII code) which occured in the present context, its 16-bit $count(k)$, and a 32-bit pointer to the next symbol in this context as displayed in Figure 7. Allocation of space is done in

Figure 8 shows the combined view of the two data structures of Order_2.

5) Updating counts: In the previous sections we described the data structures that hold the statistics of symbols. In addition to supplying the ACU with these counts for encoding a symbol using (3) one neds to update them each time a new symbol is read in the text sequence. Everything described here is done using special-purpose state-machines programmed in VHDL. For Order_-1 no updates are done. For Order_0 the updating is simple and entails incrementing by one the count of the current symbol and incrementing by one the $TotalCount$. For Order_1, we locate the count of the current symbol that corresponds to the current context and increment it by one. We also increment $TotalCount$ and update *Def_bytes* in the entry of the table (Figure IV-A3) that corresponds to the current context of order 1. For every order, when any of the symbol counts of the current context reaches half the maximal value, we renormalize all the symbol counts in this context by dividing them by 2. If a symbo count has a value of 1 we do not divide it by 2. This enables us to maintain count statistics for (and hence compress) texts of arbitrarily large length.

The operation on Order_2 is somewhat more involved. Given the current symbol we first locate the current context by finding its entry (Figure 6) in the hashtable and obtain the 32-bit address of the start of the corresponding list. We then scan the list, comparing the symbol in each entry (Figure 7) to the current symbol. If we find a match then we read the symbol count and encode the symbol in the ACU. Otherwise we read the 32-bit address pointer (located to the right of the entry) of the next entry in the list and repeat the comparison on the next entry. If we reached the end of the list (indicated by a pointer value of 0) then we encode an escape symbol. The $cumCount$ of a symbol is also computed by summing all the symbol counts as we read the entries in the list. If we did not the current symbol then after encoding an escape we append a new element to the list for this symbol with a count initialized to 1 and a pointer (to the next entry) initialized to 0. We update the *Defined_bytes* and $TotalCount$

counts in the hashtable (Figure 6).

B. ACU

The ACU operates according to the arithmetic coding algorithm described in Section III. It is based on the following components:

- ARITH_CODING - state machine which controls the add and subtract modules, computes the $l^{(n)}$ and $u^{(n)}$ in (3), counts the number of bits of the encoding of the text sequence and controls a multiplexer that selects to obtain the count statistics from one of the three order components (Order_0, Order_1, Order_2)
- DIVIDER - Two components, each performs integer division and produces the quotient and remainder.
- MULTIPLIER - Two components, each performs integer multiplication and produces porduct.

The time needed to complete an operation by the DIVIDER and MULTIPLIER is proportional to the number of bits of the operands. Having two dividers and two multipliers means that the calculation of $l^{(n)}$ and $u^{(n)}$ are executed in parallel.

V. Conclusion

We implemented in VHDL an FPGA-based data compressor that uses the PPM algorithm with arithmetic coding. It is implemented on a single FPGA chip on an inexpensive evaluation board (Xilinx Sparatan 3A) with a 50 Mhz clock. The board has an ethernet interface and acts as a server on a local area network that gets a file, compresses it and returns the number of bits of the compressed encoding. Using this server a client can do text classification by computing the NCD measure on pairs of files.

The advantage of implementing this in hardware versus software on a PC is that we can tailor the hardware architecture to the specific algorithmic requirements and make it more efficient than a general purpose CPU. Our compressor executes the algorithmic stages in parallel. The PPM unit updates multiple data structures, some of which are dynamically and incrementally allocated, that keep the symbol count statistics. The arithmetic coding unit employs multiple components that compute the boundary of the tag interval in parallel. Based on our performance tests, the compressor is more than 70 times faster than a C based software implementation running on a 3Ghz PC. One direction to extend the work is to add a pipeline in order to process multiple sequential symbols and encode them simultaneously.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] R. Feldman and J. Sanger. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2006.

[3] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience Publication, 2000.

[4] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[6] C. H. Bennett, P. Gacs, M. Li, P. Vitanyi, and W. Zurek. Information distance. *IEEE Trans. Info. Theory*, 44, 1998.

[7] V. V. Vyugin. Algorithmic complexity and stochastic properties of finite binary sequences. *The Computer Journal*, 42:294–317, 1999.

[8] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[9] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, second edition, 2000.

[10] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[11] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[12] K. Sugawara T. Watanabe and H. Sugihata. A new pattern representation scheme using data compression. *IEEE Trans Pattern Analysis Machine Intelligence*, 24:579–590, 2002.

[13] D. Cerra and M. Datcu. A model conditioned data compression based similarity measure. In *Proceedings of Data Compression Conference (DCC'08)*, pages 509 –509, 2008.

[14] D. Cerra and M. Datcu. Image classification and indexing using data compression based techniques. In *Proceedings of IEEE International Symposium on Geoscience and Remote Sensing (IGARSS'08)*, volume 1, pages I–237 –I–240, 2008.

[15] J. Ratsaby. Prediction by compression. In *Proc. of the Eighth IASTED International Conference on Signal Processing, Pattern Recognition and Applications (SPPRA'11)*, pages 282–288, 2011.

[16] U. Chester and J. Ratsaby. Universal distance measure for images. In *Proc. of* 27^{th} *IEEE Convention of Electrical and Electronics Engineers in Israel*, Eilat, Nov. 14-17, 2012.

[17] J. Ratsaby and D. Zavielov. An fpga-based pattern classifier using data compression. In *Proc. of* 26^{th} *IEEE Convention of Electrical and Electronics Engineers in Israel*, Eilat, Nov. 17-20, pages 320–324, 2010.