Painsky A, Rosset S. Lossless compression of random forests. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(2): 494–506 Mar. 2019. DOI 10.1007/s11390-019-1921-0

Lossless Compression of Random Forests

Amichai Painsky[1] and Saharon Rosset[2]

[1] *School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem 91904, Israel*
[2] *Department of Statistics and Operations Research, Tel Aviv University, Tel Aviv 69978, Israel*

E-mail: amichai.painsky@mail.huji.ac.il; saharon@post.tau.ac.il

Received February 8, 2018; revised January 3, 2019.

Abstract Ensemble methods are among the state-of-the-art predictive modeling approaches. Applied to modern big data, these methods often require a large number of sub-learners, where the complexity of each learner typically grows with the size of the dataset. This phenomenon results in an increasing demand for storage space, which may be very costly. This problem mostly manifests in a subscriber-based environment, where a user-specific ensemble needs to be stored on a personal device with strict storage limitations (such as a cellular device). In this work we introduce a novel method for lossless compression of tree-based ensemble methods, focusing on random forests. Our suggested method is based on probabilistic modeling of the ensemble's trees, followed by model clustering via Bregman divergence. This allows us to find a minimal set of models that provides an accurate description of the trees, and at the same time is small enough to store and maintain. Our compression scheme demonstrates high compression rates on a variety of modern datasets. Importantly, our scheme enables predictions from the compressed format and a perfect reconstruction of the original ensemble. In addition, we introduce a theoretically sound lossy compression scheme, which allows us to control the trade-off between the distortion and the coding rate.

Keywords entropy coding, lossless compression, lossy compression, random forest

1 Introduction

An ensemble method is a collection of sub-learners, usually decision trees like CART[1] or C4.5/C5.0[2]. The ensemble takes advantage of the favorable properties of its sub-learners, while mitigating their low accuracy by averaging or adaptively adding together many trees. Widely used ensemble methods include bagging[3], boosting[4], random forests[5] and others. During the past decades ensemble methods have gained a wide reputation of being among the most powerful off-the-shelf predictive modeling tools[6].

In order to attain their favorable predictive performance, ensemble methods usually require a large number of sub-learners, which tends to grow with the size of the problem. An increasing dataset size also results in deeper and more complex models. This most clearly manifests in random forest, where the trees are typically grown to a maximal size and are not pruned[5]. Consequently, the size of the forest strongly depends on the number of observations. For example, training a random forest of 1 000 trees (using Matlab's treeBagger routine) on a modern big dataset such as Liberty Mutual Group's Property Inspection Prediction① (which consists of 50 999 observations and 32 features), results in an average tree depth of 40 levels. Storing these trees requires 733.7 MB with the best standard solution (that is, using the compact(tree) MATLAB routine, followed by a gzip② compression).

In this work we present an extended version of [7], which focuses on lossless compression method for large tree-based ensembles. The fundamental observation underlying our method is that the random forest's trees are independent and identically distributed random en-

Regular Paper

A preliminary version of the paper was published in the Proceedings of ICDM 2016.

This work was supported by Israel Science Foundation under Grant No. 1487/12 and a Returning Scientist Fellowship from the Israeli Ministry of Immigration to Amichai Painsky.

① https://www.kaggle.com, Jan. 2019.

② File format specification version 4.3. https://tools.ietf.org/html/rfc1952, Dec. 2018.

tities, given the training data. This allows us to infer their probabilistic structure and construct an entropy code with a corresponding dictionary. As later discussed, more complicated models better describe the true probabilistic structure of the trees and therefore result in better compression rates. However, such complicated models also result in codes which require a greater number of dictionaries (and henceforth increase the overall compressed data description). Therefore, the main challenge is finding the ideal trade-off between an accurate description of the model and the total dictionary size. Our compression approach is lossless in its essence. This means we allow complete recovery of the original trees without any loss of information. Moreover, with a careful implementation, our suggested approach allows prediction straight from the compressed format.

In this extended version, we further introduce a novel lossy compression scheme which demonstrates a greater coding rate at the cost of a distortion in the reconstruction. Our lossy compression is based on subsampling and quantization of the ensemble trees, followed by lossless compression. This allows us to introduce a fundamental trade-off between distortion and coding rate in i.i.d. (independent identically distributed) ensemble methods.

A MATLAB implementation of our suggested compression scheme is publicly available at the first author's web-page③.

1.1 Related Work

The problem of storing large ensembles has gained an increasing interest in recent years, due to these methods' popularity and the emergence of extremely large datasets.

One line of work focuses on "pruning" techniques for tree ensembles. Here, the idea is to reduce the size of the ensemble by removing redundant components (features/trees, etc.), while maintaining the predictive performance. In [8], the author proposed to extend the classical cost-complexity pruning of individual trees to ensembles. On the other hand, [9, 10] propose to prune and improve the model's interpretability by selecting optimal rule subsets from tree-ensembles. Another way to reduce the complexity and/or improve the accuracy of the tree-ensembles is to merely select an optimal subset of trees from a very large ensemble generated in a random fashion (see, e.g., [11]). An additional pruning-based approach[12] is to reformulate the tree-ensemble as a linear model in terms of node indicator functions, while adding an L_1-norm regularization term (LASSO) to encourage sparsity in the features. The idea behind this approach is to select a minimal subset of indicator functions while maintaining predictive accuracy. Notice that all of these "compression" schemes are lossy and result in a pruned ensemble which is significantly different from the original ensemble. Moreover, there are no guarantees on the combination of compression rate and the difference between the pruned and the original ensemble. In other words, some ensemble may be successfully pruned while others may not.

In a different line of work, Bucelia *et al.*[13] suggested to "compress" an ensemble model by training an artificial neural network that mimics the functionality of the ensemble. This results in a significantly faster and more compact approximation of the original model. Despite these favorable properties, approximating an ensemble by a neural network is again both lossy and irreversible. In other words, the neural network predictions are not identical to the predictions of the original ensemble and in some cases may deviate quite notably in terms of root mean square error. Moreover, given the approximated neural network, it is not possible to recover the original ensemble. This means that once an approximation network replaces the ensemble, one cannot make further use or modifications to the ensemble (for example, adding more trees to improve performance). In addition, notice that for a modern big data, the trained random forest would usually consist of complex and deep (un-pruned) decision trees. It is well known that training a neural network to accurately approximate such a complicated function is not a trivial task. In fact, it typically requires an exponentially increasing number of neurons to achieve a prescribed accuracy[14].

There also exists a large body of work on the compression of different data structures in the source coding community. This includes the compression of a single or multiple tree structures[15,16]. However, this line of work focuses on more general settings — usually arbitrary or randomly constructed trees. These trees hold different probabilistic characteristics than our data-driven decision trees, which are all built on a single dataset and with only the randomness infused by the random forest algorithm differentiating them.

To the best of our knowledge, our contribution provides the first lossless compression approach for large tree ensembles.

③https://sites.google.com/site/amichaipainsky/software, Jan. 2019.

2 Basics

2.1 Random Forests

A random forest is an ensemble learning method, usually used for classification or regression problems[5]. It operates by constructing multiple decision trees at the training phase, followed by aggregating their results through a majority vote (classification) or averaging (regression). This overcomes the well-known drawback of a single decision tree, which tends to have low accuracy and high variance due to its greedy model building approach. In a random forest, each tree is constructed according to a randomly sampled subset of observations (usually with replacement), and a randomly sampled set of variables. This allows a diverse set of learners which is then averaged, thus reduces the variance associated with a single tree, and decreases the generalization error.

Random forest's trees are usually constructed by widely-used tree fitting methods like CART[1] or C4.5/C5.0④[2]. These methods are greedy recursive partitioning algorithms. In each iteration, a set of observations is split into disjoint subsets, such that a loss criterion[17,18] is minimized, in a greedy, non-regret manner (for example, [1, 19, 20]). A regression or classification tree is a tree data structure in which each internal (non-leaf) node is labeled with a variable name and a corresponding split value, while a leaf is labeled with a fitted value (a class for classification problems, or a numerical value for regression problems).

2.2 Entropy Coding

A compressed representation of a dataset involves two components — the compressed data itself and an overhead redundancy. Encoding a sequence of a length n requires at least n times its empirical entropy. This is attained through entropy coding according to the sequence's empirical distribution. The redundancy, on the other hand, may be quantified in several ways. One simple way is through a dictionary. Assume we encounter $n_0 \leqslant n$ unique symbols. Then a dictionary is simply a one-to-one mapping of each unique symbol and its corresponding codeword. An alternative way to quantify the redundancy is through a reference distribution. Assume we encode the source sequence according to a fixed (and predefined) distribution Q while the empirical distribution is P. Then, the Kullback Leibler divergence of Q from P, denoted as $D_{KL}(P||Q) = \sum P_i \log \frac{P_i}{Q_i}$, is the amount of information lost when Q is used to approximate P. In other words, $nD_{kl}(P||Q)$ is the expected number of extra bits required to encode the n samples from P using a code optimized for Q rather than the code optimized for P. Hence, the trade-off is between having efficient codes with large overhead (when using a detailed dictionary) and having inefficient codes with no overhead (when using a predefined reference distribution).

There exist several popular entropy coding schemes. The most widely used ones are Huffman and arithmetic coding[21]. The Huffman algorithm is an iterative construction of a variable-length code table for encoding the source symbols. The algorithm derives this table from the probability of occurrence of each source symbol. It can be shown that the average codeword length, achieved by the Huffman algorithm, R, satisfies $\hat{H}(X) \leqslant R \leqslant \hat{H}(X) + 1$. In arithmetic coding, instead of using a sequence of bits to represent each symbol, we represent it by a subinterval of the unit interval[21]. This means that the code for a sequence of samples is an interval whose length decreases as we add more samples to the sequence. Assuming that the empirical distribution of the sequence is known, the arithmetic coding procedure achieves an average codeword length which is within 2 bits of the empirical entropy. Although this is not necessarily optimal for any fixed sequence length (as the Huffman code), this procedure is incremental and can be used for any sequence-length. One of the major challenges of entropy coding occurs when the source is over a large alphabet size. Then, the coding redundancy becomes quite significant[22] and alternative compression methods should be considered[23−27].

In addition to the entropy coders discussed above, it is important to mention the Lempel-Ziv (LZ)-based family of coders[21]. LZ-based algorithms replace repeated occurrences of source sequences with references to a single copy of that sequence existing earlier in the uncompressed stream. The main advantage of this scheme is that it requires neither to transmit a dictionary, nor a predefined reference distribution. Yet, the LZ-based algorithms' compression rate asymptotically approaches the empirical entropy of the sequence.

3 Compression Methodology

A tree-based ensemble is a collection of decision trees, usually like CART or C4.5/C.5. Tree building algorithms can handle both numerical and categorical

④Data mining tools See5 and C5.0. https://www.rulequest.com/see5-info.html, Dec. 2018.

features and build models for regression, two-class classification, and multi-class classification. The splitting decisions in these algorithms are based on optimizing a splitting criterion over all possible splits on all variables. This means that each node in the constructed tree is defined by both a splitting variable and a corresponding split value. The fits of the tree are minimizers of the objective function for the resulting sets of leaf observations. For example, the fit of the observations in a certain leaf of a regression tree is simply the average value of these observations. A single tree structure may hold many additional characteristics and parameters (such as various summary statistics at each node). Since we are interested in compression for prediction purposes, we limit our attention to the following relevant attributes:

1) the structure of the tree,

2) the splits of the nodes (variable name and a corresponding selected split value),

3) the values of the leaves (fits),

where the structure of the tree is simply a data-structure which distinguishes between nodes and leaves (for example, Fig.1). In this work we focus on the compression of random forests, in which the trees are constructed independently and are identically distributed, given the training data. In order to apply entropy-based compression methods (such as Huffman or arithmetic coding), we first need to define a probabilistic setup for the entity we are to compress. We have that

$$\begin{aligned} P(tree) = {} & P(tree\ structure) \times \\ & P(nodes|tree\ structure) \times \\ & P(leaves|nodes,\ tree\ structure). \end{aligned}$$

This decomposition allows us to compress each of the components separately, while benefiting from a reduced algorithmic complexity.

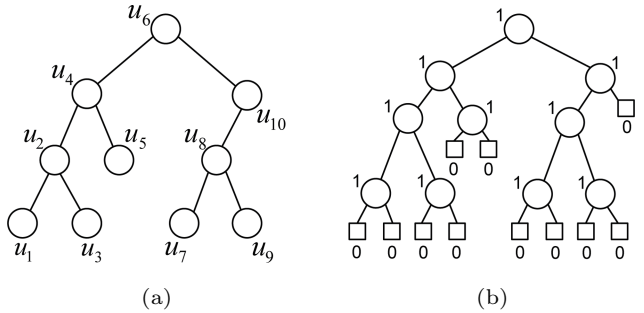

Fig.1. Zaks' tree[28] binary representation. (a) A decision tree. (b) The numbering of the nodes and the leaves related to Zaks' sequence.

3.1 Tree Structure Compression

The problem of compressing a generalized tree-based data structure has received a considerable amount attention throughout the years[15]. Here we introduce an encoding method presented by Zaks[28]. However, there exist many other compact representation formats for the structure of a tree, as later described.

Consider the tree in Fig.1(a). Label all the nodes by 1 and all the leaves (missing subtrees) by 0 as in Fig.1(b). We obtain the code sequence, called Zaks' sequence, by reading the labels in preorder (first visiting the root, then recursively traversing the left subtree in preorder, and then the right subtree in preorder). Hence, the Zaks' sequence related to the tree in Fig.1 is 11110010010011100 1000.

We have the following characterization for feasible Zaks' sequences. A bit string is a Zaks' sequence if and only if the following three conditions hold:

1) the string begins with 1;

2) the number of 0's is one greater than the number of 1's;

3) no proper prefix of the string has the property 2.

Hence, the length of a Zaks' sequence is $2n + 1$ for a tree with n nodes and it is uniquely decodable[28].

There exist several other tree structure encoding schemes[15], such as children pattern sequence (of length $2n$) and balanced parentheses (again, of length $2n$) or others.

As shown in the experiments in Section 6, the structure of the tree holds a relatively small size, compared with the other compressed components. Therefore, we choose to represent the structure of each tree with a Zaks sequence, concatenate all sequences, and apply a simple LZ-based encoder[21] to the concatenated sequence. Notice we may have treated each Zaks' sequence as an independent realization from $P(tree\ structure)$ and encode them accordingly. However, this approach would treat each sequence as a single symbol, drawn from a very large alphabet (of all possible sequences), and ignore the internal structure of the sequences. Therefore, inspired by [16], we compress the concatenated sequence using an LZ-based encoder, and take advantage of the structural nature of Zaks' sequences.

3.2 Nodes Compression

In this subsection we focus on the compression of the trees' nodes (specifically, the split selected at each

node). As mentioned above, each node is defined by a name of a variable and a corresponding split value. Notice some variables may be numerical while others categorical, and the range of values of each variable may also be significantly different than the other variables. Therefore, we derive a probabilistic model for each of the variables separately. In addition, we notice that a node only depends on its parents, as a result of the recursive construction of the tree. This means that

$$\begin{aligned} &P(nodes|tree\ structure) \\ =&\prod_{u\in\{nodes\}} P(u\text{'s variable name}|u\text{'s parents})\times \\ &P(u\text{'s split value}|u\text{'s parents}, u\text{'s variable name}). \end{aligned} \quad (1)$$

At this point it becomes quite evident that if we are to define a separate probabilistic model for each term in (1), (for example, $P(u_2$*'s variable name* | *parent name*, *parent split value*)), we would end up with a number of models which are exponential in the depth of the tree. This phenomenon is further demonstrated in Subsection 3.2.2. Moreover, encoding each node's information according to its specific model would result in an exponentially increasing number of dictionaries, as discussed in Subsection 2.2. This means we need to "cluster" models together, in order to reduce the dictionary size overhead, while maintaining a good compression rate.

3.2.1 Model Clustering

Let $s_1,\ldots,s_M$ be M sequences of independent draws, with corresponding empirical distributions $P_1,\ldots,P_M$, all on the same alphabet. We denote the lengths of the sequences as $n_1,\ldots,n_M$, respectively. We would like to encode all of these sequences according to a single codebook (and a single corresponding dictionary). Let Q be the probability distribution according to which the codebook is constructed. Then, the minimal overhead redundancy, where the minimization is with respect to the probability distribution Q, is:

$$\min_Q \sum_{i=1}^{M} n_i D_{kl}\left(P_i||Q\right)+\alpha||Q||_0, \quad (2)$$

where D_{kl} is the Kullback Leibler divergence (previously defined in Subsection 2.2), $||Q||_0$ is the L_0 norm of Q (the number of non-zero elements in Q), and α is the cost of describing a single line in the dictionary (a symbol and its codeword). The L_0 term makes this optimization problem quite involved. Therefore, we may relax it by replacing the L_0 term with L_1 (Lasso-like) or L_2 (Ridge-like) penalties, to achieve a convex optimization problem. Alternatively, assume that the alphabet size (from which each of the sequences is drawn) is finite and equals B. Then $||Q||_0 \leqslant B$ and the minimal value of (2) is bounded from above by $\sum_{i=1}^{M} n_i D_{kl}\left(P_i||Q^*\right)+\alpha B$, where Q^* is the minimizer of $\sum_{i=1}^{M} n_k D_{kl}\left(P_i||Q\right)$. Further, let us assume that B is fixed, while the lengths of the sequences $(n_1,\ldots,n_M)$ increase. In this case, the first term becomes dominant, compared with the penalty term, $\alpha B \ll \sum_{i=1}^{M} n_i D_{kl}\left(P_i||Q^*\right)$. This means that for a fixed B, and as the lengths of the sequences, $(n_1,\ldots,n_M)$ increase, we may approximate the penalty term as a constant and replace (2) with

$$\min_Q \sum_{i=1}^{M} n_k D_{kl}\left(P_i||Q\right)+\alpha B.$$

Let us now extend this problem and assume that the M sequences are to be clustered according to K different codebooks. For a fixed K, the corresponding optimization problem is

$$\min_{\underline{C},\underline{Q}} \sum_{k=1}^{K}\sum_{i=1}^{M} \mathbb{1}_{\{P_i\in C_k\}} n_i D_{kl}\left(P_i||Q_k\right)+\alpha||Q_k||_0, \quad (3)$$

where $\underline{C}=\{C_i\}_{i=1}^{K}$ and $\underline{Q}=\{Q_i\}_{i=1}^{K}$ are the clusters and corresponding codebook probability distributions respectively, and $\mathbb{1}\{\cdot\}$ is the indicator function. As before, the penalty term may be bounded from above by αB, which leads to

$$\min_{\underline{C},\underline{Q}} \sum_{k=1}^{K}\sum_{i=1}^{M} \mathbb{1}_{\{P_i\in C_k\}} n_i D_{kl}\left(P_i||Q_k\right)+\alpha BK. \quad (4)$$

This means that for sufficiently large n's and a fixed B, we may bound (3) from above, to achieve a simple clustering problem (4). Notice this clustering problem is very well studied[29] with many algorithms (mostly K-means like) and applications.

3.2.2 Clustering of Node Models

As mentioned above, we would like to cluster models together, to find the ideal trade-off between a minimal number of dictionaries and a minimal loss of bits, which results from encoding the models according to the cluster's codebook. As demonstrated in (1), we distinguish between modeling the variables' names and modeling the split values, given the variable name.

Let us first focus on the modeling of variable names. We would like to assign a designated probability distribution for a variable name, for each node in the tree,

and then cluster the distributions as in (4). We begin by defining an empirical distribution which describes the variable name in the root. Then, we may define an empirical distribution of the root's children given the root, and so forth. Obviously, the number of distributions quickly becomes intractable as we go deeper in the tree, even before we apply the clustering. Therefore, we relax the exhaustive construction of all possible models and focus on a simpler form of dependencies in the tree, in which we assume a node only depends on its depth and the variable name of its father. Therefore, assuming a forest with a maximal tree depth T, the number of possible models for the variable name is $d \times T$, where d is the number of variables in the problem.

Once we have established the list of possible models for variable names, we are ready to cluster the models according to (4), for different values of K, and choose the one which minimizes the objective. We then compress the data which corresponds to each model with a Huffman code, according to the cluster's empirical probability distribution.

Notice that the cost of describing a single dictionary line, defined as α in (4), depends on the nature of the data we are to compress and the encoder we use. Here, we may achieve a reduced dictionary size by holding a single dictionary which maps the actual name of the variable to its numeric representation and use the numeric representation in all the dictionaries we construct (e.g., instead of using the variable names "height", "weight" and "eye color", we use "00", "01", "10"). Since we do not know the codeword used for each symbol in the dictionaries, we may bound it by the maximal length of a codeword, which is d bits (the worst-case Huffman codeword for an alphabet size d). Therefore, we have that $\alpha = \log_2(d) + d$ for the variable names.

In the same manner we would like to model the split value, given the name of the variable. We use the same modeling relaxation and construct a model according to the same dependencies described above. This leads to a total of $d^2 \times T$ candidate models for clustering, since we need a different model of split values for each of the models that describe the variable names. Assuming that a variable's split values take over C different values, then the maximal codeword length is C bits and $\alpha = \log_2(C) + C$. Obviously, C may be quite large for numerical variables. However, in most decision trees (such as CART or C4.5/C5.0), a numerical split is specified by a single observation's value. This means that the numerical split value may be represented by an index of an observation, which takes $\log_2(n)$ bits. This naive representation may be further improved by applying entropy coding to these split values, as previously demonstrated. Therefore, we have that for numerical split values, $\alpha = \log_2(n) + C$.

At this point it is important to emphasize an additional difference between the split values of numerical and categorical variables. The split values of a numerical variable are numeric values. Therefore, the distribution of these values is continuous, and there is a natural order between every two different values. On the other hand, splitting a categorical variable corresponds to a partition of its categories into two disjoint sets. This means that there is no natural ordering and the distribution of the values is discrete (taking over a finite set). In other words, designing an entropy encoder (which is designated for a finite set of unordered symbols) is much more natural for categorical split values than for numerical ones. However, notice that for large datasets, variables' split values tend to take over a limited set of values as we are closer to the root, for both numerical and categorical variables. This means we can regard the numerical values as categories in this sense. As we go deeper in the tree, the split values become more uniform (and sparse) for both categorical and numerical variables; therefore most coding techniques are ineffective. These phenomena are discussed in more details detail in Section 6.

3.3 Fits Compression

We now turn to consider the compression of the tree's fits. As in the nodes' compression, we may model the (conditional) probability of fit values in each leaf of the tree. However, this requires an exponentially increasing number of models, as demonstrated in Subsection 3.2.2. Therefore, we define a simplified model in which the distribution of the fits in a leaf depends on its depth and its father's variable name. This leads to a set of probability distributions which we cluster according to (4), in the same manner mentioned above. As before, it is important to distinguish between compressing numerical and categorical values. In a classification problem, the fits are categorical and take over a finite set of values. This makes the use of entropy coders very suitable. However, regression problems result in numerical fits which may take over a continuous (and ordered) set of values. This means that we may either ignore this property and treat the continuous fits as categorical ones, or quantize the fits (through simple rounding, or in a more complicated manner using a

frequency-based quantization technique, such as Lloyd-max algorithm[30]).

Notice that by quantizing the fits we introduce an error from the original tree. Therefore we can no longer regard our method as lossless. However, such quantization results in a very regularized distortion, in the sense that we can directly set the distortion level to achieve a required compression rate (as opposed to most other lossy compression techniques mentioned in Subsection 1.1). A detailed discussion regarding fits' quantization is presented in Section 7.

Notice that while it is customary to consider the leafs as the position of the fits in a tree, in many popular decision tree implementations (such as Matlab's fitrtree, fitctree, treebagger), each node of the tree holds a fit, in case of missing values during prediction. This means that the compression rate of the fits takes a significant part in the compressed forest.

4 Our Suggested Algorithm

As described in Section 3, our suggested compression technique decomposes a tree into three components, which are the structure of the tree, the nodes of the tree, and the fits of the tree. Since the trees are independent and identically distributed (as a result of the random forest construction), we may compress the trees as memoryless draws from a complex random source, as described in Section 3. Our suggested algorithm works as follows. We first extract the Zaks sequences which describe the structure of the trees. As mentioned in Subsection 3.1, we compress each of these sequences with an LZ-based encoder. We then extract the empirical probability distributions for the nodes' names and split values. Specifically, we go over all the nodes in the trees and for each node we record its variable name and split value, its depth in the tree, and its father's variable name. We then aggregate this information into a set of conditional empirical probability distributions:

$$\begin{aligned} P_{vn} &= P(\textit{variable names}|\textit{node depth}, \\ &\quad \textit{father's variable name}), \\ P_{cv} &= P(\textit{split value}|\textit{node depth}, \textit{variable name}, \\ &\quad \textit{father's variable name}). \end{aligned}$$

Once we have gathered these sets of conditional distributions, we apply our clustering technique (4) on P_{vn} and P_{cv} (separately), to find the ideal trade-off between a minimal cost of dictionaries' description and minimal averaged redundancy, resulting in using unified dictionaries. We repeat the clustering process for different values of K to find the minimizer of (4) over all possible K's. Once we have established the chosen clustering and the mean of each cluster (which is a probability distribution Q_k), we construct a Huffman code according to Q_k and compress all the clusters' sequences accordingly. Lastly, we repeat the same construction of conditional probability distributions to the fits in the tree. We again apply our clustering technique and compress the fits accordingly. Notice that for two-class classification problems we would usually prefer to use an arithmetic encoder, which tends to out-perform the Huffman encoder for binary alphabets with skewed probability distributions. Algorithm 1 summarizes our suggested method.

5 Predictions from the Compressed Forest

As mentioned above, our suggested approach allows making predictions straight from the compressed representation of the forest. This is possible due to the prefix property of the Huffman code. Specifically, given a sequence of symbols that are coded by a Huffman code, we may decode a symbol in the sequence without decoding the entire sequence. In this way, we may access (and decode) only the required information, to make a prediction for a given future observation. Let us demonstrate our prediction scheme. First, we extract the Zaks' sequence of the first tree. This requires storing $2n+1$ bits in the random access memory (RAM) of the system for a tree of n nodes (see Subsection 3.1). Then, for every node that we encounter, we access its compressed variable name and split value in the compressed data, and decode them according to their corresponding Huffman code, as described in Subsection 3.2.2. Notice that this operation only requires the location of both the compressed information and the corresponding dictionaries in our stored data, which is directly due to the prefix property. Finally, we decode the fit of the leaf, using its corresponding Huffman dictionary, in the same manner as above (Subsection 3.3). We repeat this process for each tree in the forest. Notice that the described scheme may also be used to decode the entire forest, and not just to predict from it.

It is important to emphasize that Huffman code guarantees lossless compression, even if the data is not encoded according to its true underlaying probability distribution[31]. This property allows us to reduce the number of Huffman dictionaries that are used in our

compression scheme, while still allowing a perfect reconstruction and identical predictions to the original random forest.

Algorithm 1 . Lossless Compression of Random Forests

Require: a set of A random forest trees, $\{t_1, \ldots, t_A\}$, v = variables names, d = number of variables, $C(v_i)$ = set of split values for each $v_i \in v$, and T = maximal depth among all the trees $\{t_1, \ldots, t_A\}$

1: Extract a set of A Zaks' sequences, $\{z_1, \ldots, z_A\}$, from the given trees $\{t_1, \ldots, t_A\}$
2: Concatenate $\{z_1, \ldots, z_A\}$ to a single sequence, z_{all}
3: Compress z_{all} using an LZ encoder to achieve z_{comp}
4: Set sequences of variable names $vars(dp, fa) = \{\}$ and corresponding counters $P_{vars}(vn, dp, fa) = 0$ for all $dp \in \{1, \ldots, T\}$ and $vn, fa \in v$
5: Set sequences of split values $splits(vn, dp, fa) = \{\}$ and corresponding counters $P_{spt}(sp, vn, dp, fa) = 0$
6: Set sequences of fits $fits(dp, fa) = \{\}$ and corresponding counters $P_{fits}(vn, dp, fa) = 0$
7: **for all** $t_i \in \{t_1, \ldots, t_A\}$ **do**
8: **for all** $node_j \in t_i$ **do**
9: Set dp = the depth of $node_j$'s in t_i
10: Set fa = the variable name of $node_j$'s father
11: Set vn = the variable name of $node_j$
12: Set sp, ft = the split and fit values of $node_j$ respectively
13: Set $vars(dp, fa) = vars(dp, fa)||vn$
14: Set $P_{vars}(vn, dp, fa) = P_{vars}(vn, dp, fa) + 1$
15: Set $splits(vn, dp, fa) = splits(vn, dp, fa)||sp$
16: Set $P_{spt}(sp, vn, dp, fa) = P_{spt}(sp, vn, dp, fa) + 1$
17: Set $fits(dp, fa) = fits(dp, fa)||ft$
18: Set $P_{fits}(ft, dp, fa) = P_{fits}(ft, dp, fa) + 1$
19: **end for**
20: **end for**
21: Normalize all P's by their sums
22: **for all** $k \in \{1, \ldots, K\}$ **do**
23: Apply the clustering algorithm (4) with k clusters on the set P_{vars}
24: Set obj = the objective attained in line 23
25: **if** $obj < min\_obj$ **then**
26: Set $min\_obj = obj$, $k\_opt = k$
27: Set C_{cl} = the set of clusters attained in line 23
28: Set P_{cl} = the cluster centers attained in line 23
29: **end if**
30: **end for**
31: set $vars_{comp} = \{\}$
32: **for all** $k \in \{1, \ldots, k\_opt\}$ **do**
33: Construct a Huffman encoder $HF_{vars}(k)$ to $P_{cl}(k)$
34: **for all** $P_{vars} \in C_{cl}(k)$ **do**
35: Encode the corresponding $vars$ sequence according to $HF_{vars}(k)$, to attain $vars\_seq_{comp}$
36: Set $vars_{comp} = \{vars_{comp}, vars\_seq_{comp}\}$
37: **end for**
38: **end for**
39: Repeat steps 22–38 for $\{P_{splits}\}_{j=1}^{d}$ to attain the sets of compressed sequences $\{splits_{comp}\}_{j=1}^{d}$ and corresponding sets of Huffman encoders $\{HF_{splits}\}_{j=1}^{d}$
40: Repeat steps 22–38 for P_{fits} with an arithmetic encoder to attain the set of compressed fits $fits_{comp}$ and a corresponding set of P_{fits_cl} for decompression purpose
41: **return** z_{comp}, $vars_{comp}$, HF_{vars}, $\{splits_{comp}\}_{j=1}^{d}$, $\{HF_{splits}\}_{j=1}^{d}$, $fits_{comp}$, P_{fits_cl}

6 Experiments

We now demonstrate our suggested compression scheme on a variety of data-driven random forests, generated from publicly available real-world datasets (UCI repository⑤ and Kaggle⑥). The random forests are trained using Matlab's treeBagger routine with 1 000 trees, while the rest of the parameters are set to their default values. We compare our suggested algorithm with two different lossless compression schemes. The first, denoted as standard compression, begins with applying the compact(tree) routine on the trained forest. This creates a compact version of the random forest by eliminating redundant information and duplications of information. Then, the compact version is compressed using gzip[32]. These steps attain an immediate lossless compression by currently available off-the-shelf tools. However, notice that the compact(tree) routine is not designed solely for prediction purposes and maintains several forest attributes which are unnecessary for our prediction-oriented scheme. Therefore, we further suggest a light compression of a random forest, in which we only keep the information necessary for prediction, as listed in the beginning of Section 3, followed by gzip compression. This gives us a more relevant reference for our suggested scheme. It is important to notice that we do make some elementary adjustments to the trees prior to the gzip compression, such as replacing the alphabetical strings along the trees with short numerical values. This further enhances the compression rate of the light compression scheme.

It is important to mention that in all of our experiments we use a 64-bit representation for every numerical fit value we represent. This may be considered as an overly conservative approach for lossless compression. However, for the purpose of this work, we prefer to follow the most orthodox interpretation of losslessness, and show that we still achieve high compression rates.

We begin the presentation of our results with a case study, in which we compress a random forest trained over Liberty Mutual Group's Property Inspection Prediction dataset⑦. In this dataset, the goal is to predict a count of hazards or pre-existing damages using the property's information. This enables Liberty Mutual to more accurately identify high-risk homes that require additional examination to confirm their insur-

⑤ http://archive.ics.uci.edu/ml, Jan. 2019.
⑥ http://www.kaggle.com/competitions, Jan. 2019.
⑦ https://www.kaggle.com/c/liberty-mutual-group-property-inspection-prediction, Jan. 2019.

ability. Liberty dataset consists of 50 999 observations and 32 confidential variables, of which 16 are numerical and 16 are categorical. We train a random forest according to this dataset, as described above. We then apply the standard compression, to attain a compressed size of 733.7 MB. We further apply the light compression to the same random forest. This results in 215.6 MB, of which 122.1 MB are for the fits. Applying our suggested algorithm achieves a total compression size of 142.7 MB, where 118 MB describe the fits. We immediately notice that in both of these cases the fits hold a very dominant portion of the forest. This is a result of the numerical nature of the fits, as described in Subsection 3.3. Therefore, let us revert Liberty's regression problem into classification by comparing each observation value with the mean of all observation. This means we would now like to classify those homes for which the number of hazards or pre-existing damages is greater than the mean. We train a random forest for the classification problem and again apply the compression schemes described above. The standard compression results in a total of 723.1 MB, almost as before. However, the light compression now takes only 96.5 MB, of which 2.54 MB are for the trees structure, 10.16 MB for the variable names, 2.54 MB for the fits and 81.3 MB describe the split values. Notice that the fits now take the same portion as the tree structure, since each node holds a single binary fit.

Applying our suggested compression scheme, we get a total of 12.43 MB which breaks down to 1.81 MB for the structure, 4.02 MB for the variables names, 4.5 MB for the split values, 1.58 MB for the fits, and the reminder for the dictionaries. These results are summarized in Table 1.

Table 1. Compression Results (in MB) of 1 000 Trees Random Forests, for Liberty Mutual Classification Problem

Method	Tree Structure	Variable Names	Splits	Fits	Dictionary	Total
Light	2.54	10.16	81.3	2.54	–	96.50
Ours	1.81	4.02	4.5	1.58	0.52	12.43

We notice that by reverting the problem into classification, we achieve a reduction of 124.2 MB, due to the finite (binary) alphabet of the fits. In total, our suggested scheme achieves a compression rate of $1:40$ compared with the standard compression, and a rate of $1:5.2$ compared with the light compression.

We further analyze our results and notice that for most variables, the clustering results in three separate models which only depend on the depth of the nodes. This means we usually have a single model for low depth nodes, a single model for middle depth nodes, and a single model for deeper nodes. Moreover, we notice that the low depth model (closer to the root) is usually very sparse while the deeper model is almost uniformly distributed. This is not surprising since for a large number of observations, the splits which are closer to the root are expected to have much resemblance over different trees, while deeper splits are much more "random", due to the greedy construction of the trees. This phenomenon is observed for the variable names models and the split value models. Notice that the number of models also strongly depends on the cost of describing each line in the dictionary (the α term in (4)). Since we choose a 64-bit representation, the cost of a dictionary is relatively large and results in a small number of models. Reducing the representation accuracy to 32 bits shows an increase in the number of clusters to approximately 7.

In addition to Liberty's dataset, we examine our suggested scheme on a variety of classification (marked with $*$) and regression (marked with $+$) problems of different sizes and complexities. Notice that several classification datasets are generated from regression datasets, as in the Liberty example discussed above. The results are summarized in Table 2. All of the datasets are obtained from UCI repository and Kaggle.

As we can see, our suggested scheme achieves an average compression rate of approximately $1:70$ compared with the standard compression, and approximately $1:6$ compared with the light compression, for the classification problems. However, the average compression rates for the regression problem are only $1:4.1$ and $1:1.45$ compared with the two compression methods respectively, as a result of the costly lossless compression of the numerical fits, as discussed above. In most of the datasets, the model clustering results in 2–3 different models, in the same manner as in the Liberty dataset. This further justifies the relaxation of our trees' model, as described in Section 4, so that in practice there is no need for exponentially growing number of models prior to the clustering phase.

7 Lossy Compression

Although the focus of our work is lossless compression of random forests, there are several immediate adjustments which allow a lossy compression with favorable theoretical guarantees. In this section we introduce

Table 2. Compression Results of 1 000 Trees Random Forests, Trained over Different Datasets

Dataset (Method)	Observations, Variables	Standard (MB)	Light (MB)	Our Scheme (MB)
Iris* (3 class)	150, 4	3.730	0.082	0.013
Wages*	534, 11	15.780	1.400	0.160
Airfoil Self Noise+	1 503, 5	1.364	0.490	0.340
Airfoil Self Noise*	1 503, 5	1.260	0.108	0.012
Bike Sharing+	10 886, 11	7.690	3.390	2.380
Naval Plants+	11 934, 16	8.600	3.050	2.150
Naval Plants*	11 934, 16	8.500	2.210	0.810
Shuttle*	14 500, 9	2.162	0.280	0.049
Forests*	15 120, 55	9.136	2.910	0.340
Adults*	48 842, 14	159.100	41.600	7.300
Liberty+	50 999, 32	733.700	215.600	142.700
Liberty*	50 999, 32	723.100	96.500	12.430
Otto*	61 878, 94	209.100	48.300	6.100

two basic lossy modifications, which are tree sampling and fits quantization.

Let A be a set of independent and identically distributed trees, trained by the random forest routine, over a dataset of n observations. Let A_0 be a randomly sampled subset of A. We would like quantify the accuracy loss and the compression gain, caused by the sampling operation.

Notice that while it is customary to regard the observations as random entities (for generalization purposes), in the context of data compression we regard them as fixed. Therefore, the randomness of the ensemble is solely due to the forest construction routine.

For each observation i, we denote the mean random forest prediction for this observation on this specific dataset by $\hat{y}_i^*$. Denote the prediction from a random tree $t \in A$ in the random forest sequence by $\hat{y}_{t,i}$, and the "error" it incurs by $e_t(i) = \hat{y}_{t,i} - \hat{y}_i^*$. Let μ_i and σ_i^2 be the mean and the variance of this error, respectively. Let us now randomly sample a subset $A_0 \subset A$ of the ensemble. Then, the accuracy loss may be bounded from above by

$$D(A, A_0, \sigma^2) = var\left(\frac{\sum_{t\in A_0} e_t}{|A_0|} - \frac{\sum_{t\in A} e_t}{|A|}\right),$$

where e_t is the mean of $e_t(i)$ for all $t \in A$. Notice that the random variables e_t are i.i.d. with a mean $\mu = n^{-1}\sum_{i=1}^{n}\mu_i$ and a variance σ_i^2. We assume for simplicity that $\sigma_i^2 = \sigma^2$ is fixed (or that σ_i^2 is bounded from above by σ^2). Then $var(e_t)$ is between $\frac{\sigma^2}{n}$ and σ^2, depending on the dependence structure between predictions of the same tree. However, since $\text{var}(e_t) < \sigma^2$, we have that $\sigma^2 > \sigma_i^2$, $\forall i \in \{1 \dots n\}$. Simple derivation shows that

$$\begin{aligned} D(A, A_0, \sigma^2) &= var\left(\frac{(|A||A_0|)\sum_{k\in A_0} e_k}{|A||A_0|} - \frac{\sum_{k\in A\ A_0}}{|A|}\right) \\ &= \sigma^2|A_0|\left(\frac{1}{|A_0|} + \frac{1}{|A|}\right)^2 + \sigma^2\frac{|A| - |A_0|}{|A|^2}. \end{aligned}$$

Assuming that $|A_0| \ll |A|$ we have that

$$D(A, A_0, \sigma^2) \approx \frac{\sigma^2}{|A_0|} + \frac{\sigma^2}{|A|}.$$

It is important to mention that even though our derivation considers the ensemble's trees, $t \in A$, as a random entity, in practice they are regarded as fixed data structures to be compressed. This means that the $\frac{\sigma^2}{|A|}$ term is the "ground truth" of our random forest prediction accuracy and the accuracy loss, caused by sampling $|A_0|$ trees (followed by lossless compressing), is simply $\frac{\sigma^2}{|A_0|}$. Assuming that subsampling the ensamble does not effect the compression rate of individual trees, the compression gain we achieve is fairly straight forward and shown to be linear in the sampling ratio, $\frac{|A_0|}{|A|}$, on the average.

On top of subsampling the trees, an additional lossy compression adjustment may be attained through quantizing the (numerical) fits, as discussed in Subsection 3.3. Assume the fits take values over a finite range of size 2^c. Let us quantize the values of the fits with a naive b-bit quantization. This means we define 2^b quantization points and uniformly place them over the range. Assuming that the distortion (quantization error) is uniformly distributed (for example, through dithered quantization[33]) we attain an average accuracy loss of $\frac{2^r}{2^b} = 2^{-(b-r)}$. Further, assuming that each numerical value is represented by 64 bits, the compression gain we achieve is $\frac{b}{64}$, on the average.

Therefore, the average overall accuracy loss (that is, the variance of the difference before and after subsampling $|A_0| \ll |A|$ trees and quantizing the numerical fits) is bounded from above by

$$\frac{\sigma^2}{|A_0|} + \frac{\left(2^{-(b-r)}\right)^2}{12|A_0|},$$

while the average compression gain is a factor of $\frac{b}{64}$ for the compressed fits and an additional factor of

$\frac{|A_0|}{|A|}$ for the entire compressed ensemble. Notice that while there exist more adequate frequency based quantization techniques (for example, Lloyd-max[30]), the naive quantization described above offers simple and favorable theoretical properties. However, in practice, one may achieve better performance by applying those methods.

Let us now illustrate our suggested lossy compression approach. Fig.2 demonstrates the fits quantization (Fig.2(a)) and the tree subsampling (Fig.2(b)), applied to the (regression) Air Self Noise dataset (see Table 2). Here, we split the data-set to 80% training set and 20% testset. We train a random forest (using Matlab's treebagger routine) and evaluate the mean square error (MSE) on the test set. Then, we apply the two lossy compression techniques discussed above. Fig.2(a) demonstrates the effect of the fits quantization. The x-axis is the number of quantization bits used to describe the fits, the dashed curve is the corresponding MSE (on the test set), and the straight (full) line is the compression size. As we can see, we may represent the fits by only 7 bits, with no significant degradation in performance of the random forest. This results in a compression size of approximately 47 KB. Notice that the over-conservative 64-bit representation used in Table 2 allows a compression size of 340 KB. Now, let us subsample the trees in the forest, while maintaining the 7-bit representation for the fits. Fig.2(b) demonstrates the MSE (dashed curve) and the resulting compression size (straight full line) for different numbers of subsampled trees (x-axis). Here we observe that by sampling only 250 trees of the forest, we may reduce the compression size to only 11 KB, while almost maintaining the same performance. Therefore, we conclude that by both quantizing the fits and subsampling the forest, we may reduce the compression size from 340 KB (in the conservative lossless case) to only 11 KB with no significant impact on the generalization performance. In addition, we notice the linear threads of our compression size curves, which illustrate (and justify) our analysis above.

Let us further apply our lossy compression techniques to a larger dataset. Fig.3 demonstrates the MSE and the corresponding compression size of our suggest method, applied to the (regression) Bike Sharing dataset (Table 2). Here, we may reduce the compression size from 2.38 MB to only 300 KB with no significant effect on the generalization performance. This is achieved by representing the fits with 12 bits, while subsamping 600 trees from the forest.

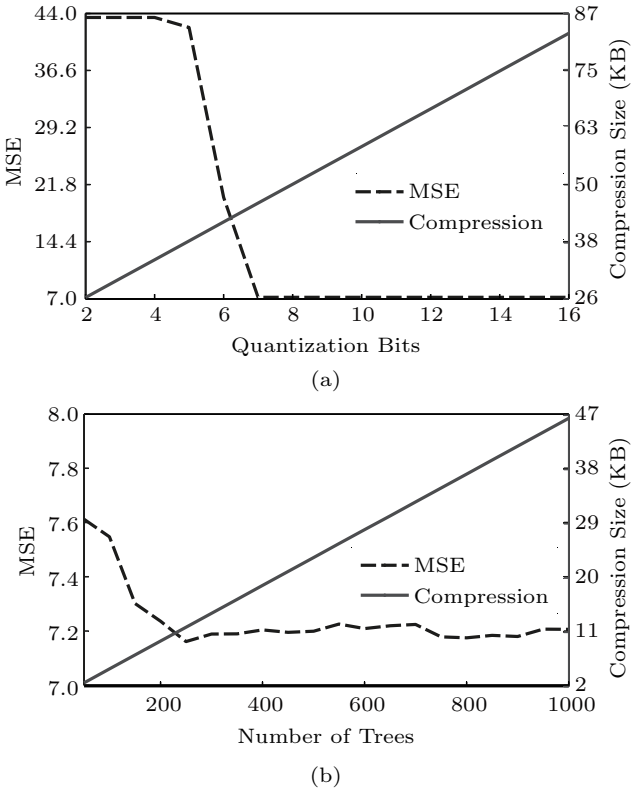

Fig.2. Lossy compression of Air Foil Noise dataset. (a) Fits quantization. (b) Tree subsampling.

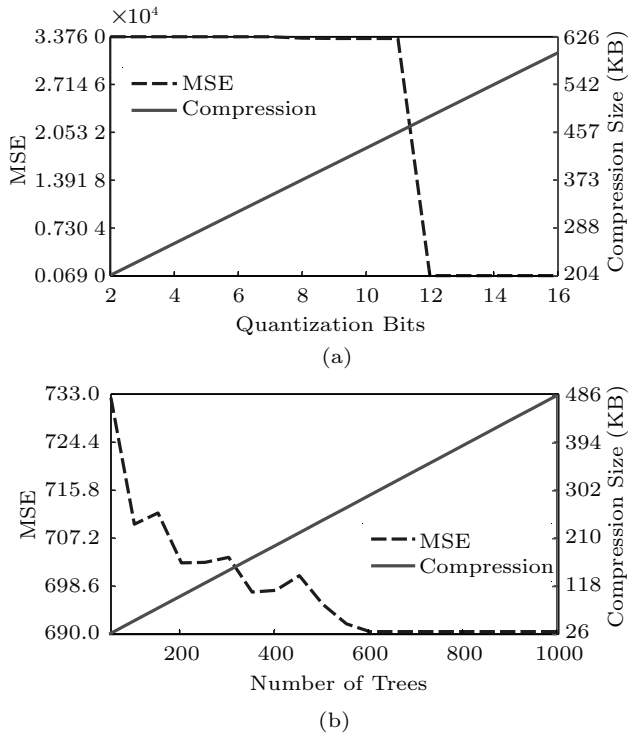

Fig.3. Lossy compression of the Bike Sharing dataset. (a) Fits quantization. (b) Tree subsampling.

It is important to mention that our suggested lossy approach is typically not competitive with some

alternative methods, such as neural-networks based compression[13]. Our suggested lossy compression typically compresses the forest in a factor of up to a 100 (from the uncompressed representation), while neural-based methods compress in factors of 1 000 and more[13]. However, our main advantage lies in the ability to provide a theoretically sound trade-off between distortion and compression rate and to explicitly control the desired performance. In addition, our method allows to further modify the forest (for example, by adding more trees), even after the lossy compression is applied. This serves as a balancing mechanism for coding implementations.

8 Conclusions

In this work we introduced a novel method for lossless compression of random forests. Our suggested method uses the independent and identically distributed nature of the trees to fit probabilistic models and compress the data accordingly. Since the number and the complexity of the models grow with the size of the problem, we applied model clustering according to Bregman divergence. This allows us to find the optimal trade-off between a smaller set of models that accurately describe the data, and corresponding dictionaries for decompression purposes.

While to the best of our knowledge, our suggest approach is unique in its lossless nature. There exists a large body of work on lossy compression of ensemble methods. Most of these lossy compression schemes manipulate the forest (by pruning or mimicking it), with hardly any guarantees on the resulting prediction accuracy. The main advantage of our suggested scheme is that it provides a complete and accurate recovery of the forest. This property ensures the same prediction accuracy as the original forest. In addition, it allows future modification to the forest (such as adding more trees, applying further inference, and so on.). Further, since our method is lossless and directly compresses the trees, a more complex random forest would not necessarily result in a worse compression rate (as demonstrated in Table 2). Notice that the lossy schemes, on the other hand, may result in a severe deterioration of accuracy in order to achieve a prescribed compression rate, as described in Subsection 1.1.

Although the focus of our work is lossless compression, our suggested scheme may also be extended to lossy compression, as described in Section 7. The main advantage of our lossy scheme is that it is easy to implement and provides theoretical guarantees on both the accuracy loss and the achieved compression gain. This allows the user to find the ideal balance between the two without blindly applying a series of lossy compression tasks.

It is important to mention several popular variants of tree ensembles which imply different probabilistic structures. For example, completely randomized trees (CRT)[34,35] utilize a recursive partitioning in which the observations in each node are split according to a randomly chosen feature and a corresponding random split value. Therefore, we expect less resemblance among the trees. Further, it leads to more uniform distributions of the splitting rules in each node, and henceforth, a lower compression rate. On the other hand, there exist more complicated tree-based structures such as deep forest[36], where different random forests are cascaded layer by layer, similar to deep neural networks. This results in more involved probabilistic dependencies, as we consider the collection of all the trees in the system. Nevertheless, we may still cluster and encode different models together, to introduce a compression gain.

All of these properties make our suggested compression framework a favorable methodology, both in theory and in practice.

References

[1] Breiman L, Friedman J, Olshen R A, Stone C J. Classification and Regression Trees (1st edition). Chapman and Hall/CRC, 1984.

[2] Quinlan J R. C4.5: Programs for Machine Learning (1st edition). Morgan Kaufmann Publishers, 1992.

[3] Breiman L. Bagging predictors. *Machine Learning*, 1996, 24(2): 123-140.

[4] Schapire R E. The boosting approach to machine learning: An overview. In *Nonlinear Estimation and Classification*, Denison D D, Hansen M H, Holmes C C, Mallick B, Yu B (eds.), Springer, 2003, pp.149-171.

[5] Breiman L. Random forests. *Machine Learning*, 2001, 45(1): 5-32.

[6] Friedman J, Hastie T, Tibshirani R. The Elements of Statistical Learning: Data Mining, Inference, and Prediction (1st edition). Springer, 2001.

[7] Painsky A, Rosset S. Compressing random forests. In *Proc. the 16th International Conference on Data Mining*, December 2016, pp.1131-1136.

[8] Geurts P. Some enhancements of decision tree bagging. In *Proc. the 4th European Conference Principles of Data Mining and Knowledge Discovery*, Sept. 2000, pp.136-147.

[9] Meinshausen N. Node harvest. *The Annals of Applied Statistics*, 2010, 4(4): 2049-2072.

[10] Friedman J H, Popescu B E. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2008, 2(3): 916-954.

[11] Bernard S, Heutte L, Adam S. On the selection of decision trees in random forests. In *Proc. the 2009 International Joint Conference on Neural Networks*, June 2009, pp.302-307.

[12] Joly A, Schnitzler F, Geurts P, Wehenkel L. L_1-based compression of random forest models. In *Proc. European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, April 2012, pp.375-380.

[13] Buciluǎ C, Caruana R, Niculescu-Mizil A. Model compression. In *Proc. the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 2006, pp.535-541.

[14] Tikk D, Kóczy L T, Gedeon T D. A survey on universal approximation and its limits in soft computing techniques. *International Journal of Approximate Reasoning*, 2003, 33(2): 185-202.

[15] Katajainen J, Mäkinen E. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1990, 1(04): 425-447.

[16] Chen S, Reif J H. Efficient lossless compression of trees and graphs. In *Proc. the 6th Data Compression Conference*, March 1996, pp.428.

[17] Painsky A, Wornell G W. On the universality of the logistic loss function. arXiv:1805.03804, 2018. https://arxiv.org/pdf/1805.03804.pdf, September 2018.

[18] Painsky A, Wornell G W. Bregman divergence bounds and the universality of the logarithmic loss. arXiv:1810.07014, 2018. http://export.arxiv.org/pdf/1810.07014, September 2018.

[19] Hothorn T, Hornik K, Zeileis A. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 2006, 15(3): 651-674.

[20] Painsky A, Rosset S. Cross-validated variable selection in tree-based methods improves predictive performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017, 39(11): 2142-2153.

[21] Sayood K. Introduction to Data Compression (5th Edition). Morgan Kaufmann, 2017.

[22] Szpankowski W, Weinberger M J. Minimax pointwise redundancy for memoryless models over large alphabets. *IEEE Transactions on Information Theory*, 2012, 58(7): 4094-4104.

[23] Orlitsky A, Santhanam N P, Zhang J. Universal compression of memoryless sources over unknown alphabets. *IEEE Transactions on Information Theory*, 2004, 50(7): 1469-1481.

[24] Painsky A, Rosset S, Feder M. Universal compression of memoryless sources over large alphabets via independent component analysis. In *Proc. the 2015 Data Compression Conference*, April 2015, pp.213-222.

[25] Painsky A, Rosset S, Feder M. A simple and efficient approach for adaptive entropy coding over large alphabets. In *Proc. the 2016 Data Compression Conference*, March 2016, pp.369-378.

[26] Painsky A, Rosset S, Feder M. Large alphabet source coding using independent component analysis. *IEEE Transactions on Information Theory*, 2017, 63(10): 6514-6529.

[27] Painsky A, Rosset S, Feder M G. Linear independent component analysis over finite fields: Algorithms and bounds. *IEEE Transactions on Signal Processing*, 2018, 66(22): 5875-5886.

[28] Zaks S. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 1980, 10(1): 63-82.

[29] Banerjee A, Merugu S, Dhillon I S, Ghosh J. Clustering with Bregman divergences. *Journal of Machine Learning Research*, 2005, 6: 1705-1749.

[30] Lloyd S. P. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 1982, 28(2): 129-137.

[31] Cover T M, Thomas J A. Elements of Information Theory (2nd edition, e-book). John Wiley & Sons, 2012.

[32] Deutsch L P. Gzip file format specification version 4.3. 1996. https://www.rfc-editor.org/rfc/rfc1952.txt, Oct. 2018.

[33] Schuchman L. Dither signals and their effect on quantization noise. *IEEE Transactions on Communication Technology*, 1964, 12(4): 162-165.

[34] Geurts P, Ernst D, Wehenkel L. Extremely randomized trees. *Machine Learning*, 2006, 63(1): 3-42.

[35] Liu F T, Ting K M, Yu Y, Zhou Z H. Spectrum of variable-random trees. *Journal of Artificial Intelligence Research*, 2008, 32: 355-384.

[36] Zhou Z H, Feng J. Deep forest: Towards an alternative to deep neural networks. arXiv:1702.08835, 2017. https://arxiv.org/pdf/1702.08835v2.pdf, September 2018.

Amichai Painsky received his B.Sc. in electrical engineering from Tel Aviv University (2007), Tel Avir, his M.Eng. degree in electrical engineering from Princeton University (2009), Princeton, and his Ph.D. degree in statistics from the School of Mathematical Sciences in Tel Aviv University, Tel Aviv. He is currently a post-doctoral fellow, co-affiliated with the Israeli Center of Research Excellence in Algorithms (I-CORE) at the Hebrew University of Jerusalem, Jerusalem, and the Signals, Information and Algorithms (SIA) Lab at MIT (Massachusetts Institute of Technology). His research interests include data mining, machine learning, statistical learning and their connection to information theory.

Saharon Rosset is a professor in the Department of Statistics and Operations Research at Tel Aviv University, Tel Aviv. His research interests are in computational biology and statistical genetics, data mining and statistical learning. Prior to his tenure at Tel Aviv, he received his Ph.D. degree from Stanford University in 2003 and spent four years as a Research Staff Member at IBM Research in New York. He is a five-time winner of major data mining competitions, including KDD Cup (four times) and INFORMS Data Mining Challenge, and winner of the Best Paper Award at KDD (ACM SIGKDD International Conference on Knowledge Discovery and Data Mining) twice.