The Model-Summary Problem and a Solution for Trees

Biswanath Panda [1], Mirek Riedewald [2], Daniel Fink [3]

[1]*Google Inc., USA;* bpanda@google.com

[2]*College of Computer and Information Science, Northeastern University, USA;* mirek@ccs.neu.edu

[3]*Cornell Lab of Ornithology, USA;* df36@cornell.edu

Abstract— **Modern science is collecting massive amounts of data from sensors, instruments, and through computer simulation. It is widely believed that analysis of this data will hold the key for future scientific breakthroughs. Unfortunately, deriving knowledge from large high-dimensional scientific datasets is difficult. One emerging answer is exploratory analysis using data mining; but data mining models that accurately capture natural processes tend to be very complex and are usually not intelligible. Scientists therefore generate model summaries to find the most important patterns learned by the model. We formalize the model-summary problem and introduce it as a novel problem to the database community. Generating model summaries creates serious data management challenges: Scientists usually want to analyze patterns in different "slices" and "dices" of the data space, comparing the effects of various input variables on the output. We propose novel techniques for efficiently generating such summaries for the popular class of tree-based models. Our techniques leverage workload structure on multiple levels. We also propose a scalable implementation of our techniques in MapReduce. For both sequential and parallel implementation, we achieve speedups of one or more orders of magnitude over the naive algorithm, while guaranteeing the exact same results.**

I. Introduction

Across many scientific disciplines, the availability of very large amounts of data is creating a paradigm shift. This is usually referred to as *data-driven science* or *eScience*. The National Science Foundation (NSF) in the US has made data-driven science one of its funding priorities, and there are similar efforts world-wide. In its 2007 report, the NSF Cyberinfrastructure Council stated that "...U.S. international leadership in science and engineering will increasingly depend upon our ability to leverage this reservoir of scientific data captured in digital form, and to transform these data into information and knowledge aided by sophisticated data mining, integration, analysis and visualization tools."

Data management skills are needed to solve many of the grand challenges in data-driven science. However, these problems have not received adequate attention in the database community. In this paper we introduce the model-summary computation problem, a challenging general problem from data-driven science. We show solutions for a popular instance of the problem, and point out various future challenges. To illustrate the problem, consider the following example from bird ecology.

Ornithologists and conservation biologists try to identify large-scale threats to sensitive bird species such as climate change or land use change associated with human population expansion. To do this, they need to explore the complex and highly dynamic ecology of bird populations across huge geographic extents. The traditional approach of having experts come up with a hypothesis and then design an experiment to collect the data for testing this hypothesis requires a sufficient understanding of the studied phenomenon. And due to the high cost of collecting data from a carefully designed experiment, it is limited to the study of few variables at small scale.

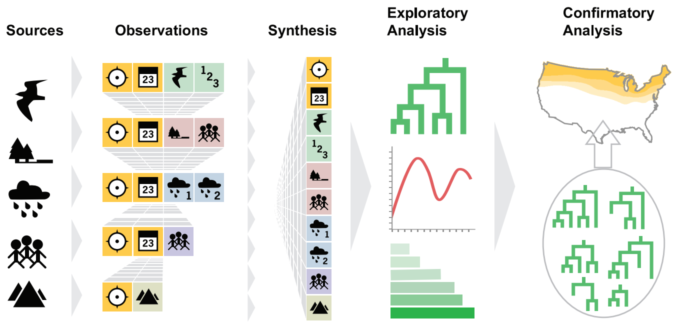


Fig. 1. Typical data-intensive science workflow

Instead, the bird ecology community, like many other domain sciences, is shifting focus to the analysis of non-experimental, or what we call "observational", data that can be more efficiently collected across large spatial and temporal scales [1]. Figure 1 shows the corresponding workflow. For example, the Cornell Lab of Ornithology and dozens of partner organizations are collecting millions of bird sighting reports every year from various protocols (see www.avianknowledge.net). These records are joined based on time and location with other datasets, adding thousands of attributes describing habitat, climate, census, elevation, and other features (synthesis step).

In the next step of the workflow, exploratory analysis techniques are used to identify and describe the input attributes that are most strongly associated with observed distributions of birds. Understanding such *statistical associations* is essential quantitative information, which provides the inspiration for new hypotheses about the true causal relationships. Then more traditional hypothesis-driven science can be carried out through careful collection of additional data or through quasi-experimental design approaches [2].

Exploratory analysis begins by training prediction models. These models are then analyzed as an approximation of the underlying real process that generated the data [3], [4]. Non-parametric data mining models like tree-based ensembles, SVM's, and artificial neural nets (ANNs) are perfectly suited for exploratory analysis. They are flexible enough to model complex interactions between many variables and they can handle large datasets. Even with little understanding of a complex natural process, data mining techniques can generate excellent models that have high predictive accuracy by "letting the data speak for itself" and avoiding prior assumptions as much as possible.

Although non-parametric models have great accuracy, they tend to be very complex for all but the most trivial data sets. This makes it difficult to directly discover associations between input attributes and output. Even decision trees become unintelligible once they have thousands of nodes. In other words, the model behaves like a *blackbox*. Scientists therefore compute low-dimensional summaries to extract and visualize "what the model has learned." All such summaries are based on predictive "experiments", calculated across a series of systematic, structured predictions from the exploratory model. Most often, these analyses begin by investigating how each attribute, in isolation, affects the output. More detailed investigations deal with the joint effect of attribute sets, called "interactions". Importantly, the number of model evaluations required increases dramatically when we investigate high-order interactions. Thus, a thorough investigation of even a relatively small analysis may require a huge number of summaries, and hence model predictions, to be computed to generate useful scientific knowledge. In fact, it is the bottleneck of the workflow as described in Figure 1, dwarfing other costs such as model training.

The final step in the workflow, confirmatory analysis, is used to refine the results from exploratory analyses and strengthen the basis for inference, e.g., by taking into account the estimation error. In practice, error and confidence is estimated with resampling techniques. This exacerbates the computational bottleneck associated with summary generation.

This model-summary problem is not specific to bird ecology. It is relevant to any scientific use of predictive modeling or supervised learning. As programs like NSF's DataNet are poised to create massive repositories with petabytes of data from many scientific disciplines, exploratory analysis as discussed here will become an indispensable tool, and achieving *scalability* will be crucial.

In this paper we make the following contributions:

- We introduce the general *model-summary computation* problem for complex data mining models and identify common structure in the workloads for generating model summaries. (Sections II and III)
- We argue that any solution has to be tailored to the model type and propose algorithms that take advantage of the workload structure for speeding up summary computations in tree-based models, including state of the art ensembles. (Sections IV, V and VI)
- We evaluate our algorithms using models built from real world data and show impressive speedups in computing large sets of summaries. (Section VII)

Section VIII discusses algorithm extensions, Section IX discusses related work, and Section X concludes the paper.

II. Examples

To illustrate the problem, we discuss a toy example and then show how summaries are used in a real-world study. Both examples are from bird ecology, but it is easy to see how they generalize to other domains, e.g., analysis of medical records.

A. Toy Example

Assume a scientist has trained a model $F(\text{Elev}, \text{Year}, \text{Hpop})$, which for a given combination of elevation ($e \in \text{Elev}$), year ($y \in \text{Year}$), and human population density ($h \in \text{Hpop}$) can accurately predict the probability of observing some bird species of interest.[1]

Now the scientist would like to study how bird occurrence is associated with Year by generating a plot like the right one in Figure 2. In general, there is no perfect way of summarizing a high-dimensional function with a lower-dimensional one. Some information will inevitably be lost, no matter which method we choose. However, all accepted methods follow the same fundamental principle of experimental design: *To study the dependency of the output on a set of variables, one varies only the values of these variables, while holding all other variables constant.* The two basic approaches for generating a summary are:

Non-aggregate summaries: The most fine-grained way of studying the effect of Year on bird occurrence is the following. We pick a pair $(e_1, h_1) \in \text{Elev} \times \text{Hpop}$ and then compute $F_{e_1,h_1}(\text{Year}) = F(e_1, \text{Year}, h_1)$. In particular, if we want to visualize the effect for the years 1994 to 2004, then we evaluate the model for points $(e_1, 1994, h_1)$, $(e_1, 1995, h_1)$,..., $(e_1, 2004, h_1)$. Hence the summary consists of the 11 points $(1994, F(e_1, 1994, h_1))$, $(1995, F(e_1, 1995, h_1))$,..., $(2004, F(e_1, 2004, h_1))$. We can do the same for many different pairs $(e_i, h_i) \in \text{Elev} \times \text{Hpop}$.

Aggregate summaries: Looking at Year-summaries for many different elevation-human population pairs will give a very detailed picture of the statistical association between Year and bird occurrence. However, scientists usually prefer to aggregate many of these summaries. Since the data is high-dimensional, it tends to be sparse and hence aggregate summaries are usually trusted more. And aggregating summaries also reduces the amount of information that needs to be examined.

An aggregate summary is produced by averaging multiple non-aggregate summaries. In our example, for a set of elevation-human population pairs $\{(e_i, h_i)\}_{i=1}^{n}$, the aggregate

[1]We slightly abuse notation by using the same symbol for both an attribute name and the set of all values of this attribute, e.g., $y \in \text{Year}$ means that y is a value of attribute Year.

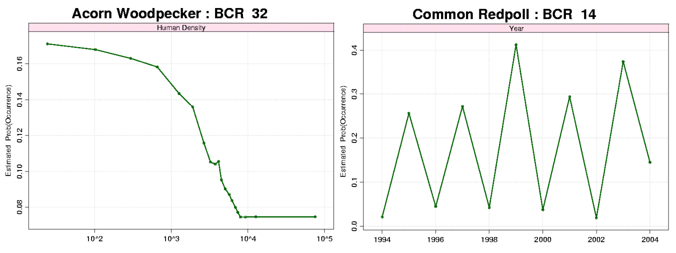


Fig. 2. Summaries of a complex bird occurrence prediction model.

summary is computed as $\frac{1}{n}\sum_{i=1}^{n} F_{e_i,h_i}(\text{Year})$. Stated differently, for the year 1994, the function value in the summary is $\frac{1}{n}\sum_{i=1}^{n} F(e_i, 1994, h_i)$; similar for the other years.

Which type of summary (aggregate versus non-aggregate) and for which pairs (e_i, h_i) to generate the summary on Year is largely a function of the research questions of interest. Visualizations (or *plots*) are a convenient way to present summaries.

B. Real Example

Consider again the example from bird ecology where ornithologists would like to analyze bird observation records. As mentioned earlier, by joining the observations with other datasets about habitat, climate etc, each observation record is described by thousands of attributes. Data mining techniques can produce highly accurate models, but often these models are unintelligible and do not reveal statistical associations directly. To understand what the model has learned, scientists rely on low-dimensional summaries like those discussed for the toy example above. Partial dependence plots are one particularly popular type of aggregate summaries [3], [5], [6], [7].

Figure 2 presents an example of two 1-dimensional partial dependence plots. They show the estimated probability of occurrence of a bird species at feeders in some Bird Conservation Region (BCR)[2], for a selected summary attribute. The left plot is for the Acorn Woodpecker in California, showing a drop in the probability of Acorn Woodpecker occurrence as human population density increases above 1,000 people per square mile. It is hypothesized that habitat competition between the woodpecker, which needs dead or dying branches to store acorns, and humans who remove these branches could be the cause for this decline. The plot on the right shows the biennial winter irruptive migration of Common Redpoll into New England, likely caused by biennial cycles of production of tree seeds in Northern Canada. Interestingly, Purple Finch shows a similar biennial pattern in BCR 14, but its highs and lows are exactly the opposite compared to Common Redpoll. This hints at a biological process driven by availability of certain food sources and competition for similar habitat.

As the example indicates, interesting patterns could be observed for various species, attributes, and regions. Scientists therefore would like to search across many different species, variables (attributes), and regions to see if there are any interesting patterns like those in Figure 2. We are currently working on a search engine for such model summaries. It will enable scientists to express their preferences, e.g., to find summaries showing a strong effect (measured as the difference between max and min value in the summary), and then return a ranked list of summaries according to these preferences. However, to make such a pattern search engine useful, we first have to *create* a large collection of these summaries.

[2]BCRs correspond to large geographical regions in North America.

Creating summaries is an expensive process, even for a small dataset. Assume we have 1000 attributes that are potentially interesting. Hence there are $\binom{1000}{1} + \binom{1000}{2} \approx 500,000$ different 1- and 2-dimensional summaries. To produce a plot like those in Figure 2, we need to evaluate the model for sufficiently many values of the summary attribute (the one on the x axis), at least 10. And each point in an aggregate summary is obtained by appropriately averaging over many combinations of data points, typically 1000 or more, to take the average contribution of other variables into account. To discover regional trends, not only for geographical regions, but also for say certain elevation ranges, human population ranges, or temperature ranges, this analysis is done for many "slices" and "dices" of the data space, i.e., various selections of the original data. At the very least, thousands of such selections are typically explored. This results in a total of at least $500,000 \cdot 10 \cdot 1000 \cdot 1000 = 5 \cdot 10^{12}$ model evaluations. At the optimistic estimate of 1 microsecond per evaluation, this adds up to about 2 months of computation time.

For larger data, more complex models, and more ambitious studies, we experienced that the naive method of creating summaries is computationally infeasible. On top of that, scientists cannot rely on studying a single model. Correlated attributes distort the results and noise affects model structure. Hence during confirmatory analysis, scientists explore how summaries vary when different projections of the data are studied (eliminating some of the correlated attributes), and different samples are used for training the models. In short, without dramatically speeding up summary computation, scientists are limited to small-scale studies or poor approximations.

In theory summaries like those in Figure 2 could also be obtained by training a model directly on the low-dimensional space, i.e., a projection of the dataset on the attributes of interest. However, this usually results in poor models and hence low-quality summaries, because variables that do not appear in the summary can still have a significant influence on the output. Since these would be projected away, their effects cannot be learned by the model.

III. The Model-Summary Problem

A. Terminology

We will use the terms *attribute* and *variable* interchangeably throughout the paper. Like in a database, an attribute describes a property of a data record. At the same time, this attribute corresponds to a variable in a statistical or data mining model.

We will refer to those attributes that the scientist wants to visualize with a model summary as the *summary attributes*. The remaining attributes in the model are the *non-summary* *attributes* of this summary. In the toy example of a summary on Year, Year is the summary attribute, while Elev and Hpop are the non-summary attributes. Similarly, if the scientist wanted to study the combined effect of Year and Hpop on bird occurrence, she would choose these two as the summary attributes, while Elev would be the non-summary attribute. The corresponding summary plot would show a two-dimensional function surface.

We refer to the values of the summary attributes at which the model is evaluated as the *visualization points*. In the right summary in Figure 2 on Year, all year values between 1994 and 2004 were selected as the visualization points.

Let $\mathcal{X} = \{X_1, X_2, \ldots, X_{|\mathcal{X}|}\}$ be a set of $|\mathcal{X}|$ attributes with domains dom_1, dom_2,..., $\mathrm{dom}_{|\mathcal{X}|}$, respectively. With $D = \{\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(|D|)\}$, where all $\mathbf{x}(i) \in \mathrm{dom}_1 \times \mathrm{dom}_2 \times \cdots \times \mathrm{dom}_{|\mathcal{X}|}$, we denote a dataset from the input domain. Let Y be the output with domain dom_y and let $F : \mathrm{dom}_1 \times \mathrm{dom}_2 \times \cdots \times \mathrm{dom}_{|\mathcal{X}|} \rightarrow \mathrm{dom}_y$ be a data mining model. Model F maps $|\mathcal{X}|$-dimensional vectors $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathcal{X}|})$ of input attribute values to the corresponding output value $F(\mathbf{x})$.

B. Problem Definition

We first formalize the notion of a summary and then define the model summary problem.

Definition 1: Let $\mathcal{X}$, F and D be as defined above and let $\mathcal{S} \subset \mathcal{X}$ and $\tilde{\mathcal{S}} = \mathcal{X} - \mathcal{S}$ be the sets of summary and non-summary attributes, respectively. Let $V_{\mathcal{S}} \subseteq \bigotimes_{X_j \in \mathcal{S}} \mathrm{dom}_j$ denote the set of visualization points. The *summary* of F on $\mathcal{S}$ and $V_{\mathcal{S}}$ is defined as

$$\left\{ \left(\mathbf{x}_{\mathcal{S}}, \hat{F}_{\tilde{\mathcal{S}}}(\mathbf{x}_{\mathcal{S}})\right) | \, \mathbf{x}_{\mathcal{S}} \in V_{\mathcal{S}}, \, \hat{F}_{\tilde{\mathcal{S}}}(\mathbf{x}_{\mathcal{S}}) = \frac{1}{|D|} \sum_{i=1}^{|D|} F(\mathbf{x}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}}(i)) \right\} \tag{1}$$

where $\mathbf{x}_{\tilde{\mathcal{S}}}(i) = \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$ is the projection of the i-th data record in D on the attributes in $\tilde{\mathcal{S}}$.

Notice that for $|D| = 1$, we obtain a non-aggregate summary, while for $|D| > 1$ it would be an aggregate summary (see Section II-A). Depending on the choice of points in dataset D, aggregate summaries with different properties can be generated. For example, if D is the set of data points that was used for training model F, then the summary is called a *partial dependence function* [5]. Another popular choice for D is to use points from a *regular grid of non-summary attribute value combinations*.

The different choices of D affect the summary properties, but these are irrelevant from our point of view. Our techniques support *all variations* of summary definitions. We can now define the model-summary problem.

Definition 2: Let $\mathcal{X}$, F and D be as defined above and let $P = \{p_1, p_2, \ldots, p_{|P|}\}$ be a set of summaries ("plots"). Each summary p_i, $1 \leq i \leq |P|$, is defined by its set of summary attributes $p_i.\mathcal{S} \subset \mathcal{X}$ and a set of visualization points $p_i.V_{\mathcal{S}} \subseteq \bigotimes_{X_j \in p_i.\mathcal{S}} \mathrm{dom}_j$. The *model-summary computation problem* is to compute all summaries in P efficiently and to scale to large problems.

Algorithm 1 : Naive Algorithm

```
Input: model $F(\mathcal{X})$, summary attributes $\mathcal{S}$, non-summary attributes $\tilde{\mathcal{S}}$, dataset $D$, visualization points $V_{\mathcal{S}}$
1: for all $\mathbf{v}_{\mathcal{S}} \in V_{\mathcal{S}}$ do
2:     $\mathrm{sum} = 0$
3:     for all $\mathbf{x} \in D$ do
4:         $\mathrm{sum} = \mathrm{sum} + F(\mathbf{v}_{\mathcal{S}}, \pi_{\tilde{\mathcal{S}}}(\mathbf{x}))$
5:     return $(\mathbf{v}_{\mathcal{S}}, \mathrm{sum}/|D|)$
```

C. Summaries for Blackbox Models

The naive algorithm as outlined in Algorithm 1 computes a single summary. It directly implements the summary definition and it is the only algorithm currently available for this problem. (There is also an approximation algorithm for a restricted version of the problem, which we discuss in Section IX.)

For each visualization point $\mathbf{v}_{\mathcal{S}} \in V_{\mathcal{S}}$, the naive algorithm iterates through all data points $\mathbf{x} \in D$ and evaluates the model for each query point obtained by combining $\mathbf{v}_{\mathcal{S}}$ with the appropriate projection of $\mathbf{x}$ on the non-summary attributes. In our toy example for the Year summary with visualization points 1994 to 2004 and data set $\{(e_i, y_i, h_i)\}_{i=1}^{n}$, we query the model with $(e_1, 1994, h_1)$,..., $(e_1, 2004, h_1)$, then $(e_2, 1994, h_2)$,..., $(e_2, 2004, h_2)$, and so on.

In fact, if the model is a true blackbox, then the naive algorithm is the only option: Even if two query points are similar, their model predictions can be very different. Hence one cannot obtain the exact model summary without actually evaluating the model for each individual point. Even an approximate algorithm would be problematic for a blackbox model, because there is no way to establish a bound on the similarity of predictions based on the similarity of the input values, without actually generating the predictions.

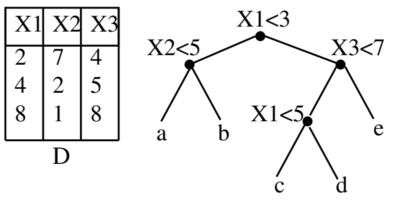
Stated differently, **any improvement over the naive algorithm has to take advantage of the internal structure of the data mining model**.

D. Workload Properties

Since data mining models, though complex, are not true blackboxes, there are opportunities for designing algorithms with lower cost than the naive one. The following workload properties can be exploited:

Repetitive structure among query points: Algorithm 1 evaluates the model for all query points in $V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(D)$, i.e., the cross product of the set of visualization points with the $\tilde{\mathcal{S}}$-projected set of data points. Hence for each $\mathbf{v}_{\mathcal{S}} \in V_{\mathcal{S}}$, there are $|D|$ query points that all have the same value vector $\mathbf{v}_{\mathcal{S}}$ for the summary attributes. Similarly, for each $\mathbf{x} \in D$, there are $|V_{\mathcal{S}}|$ query points that all have the same value vector $\pi_{\tilde{\mathcal{S}}}(\mathbf{x})$ for the non-summary attributes. This creates a potential for sharing computation across query points.

Aggregation: For a given visualization point $\mathbf{v}_{\mathcal{S}}$, its summary output is the average of the model predictions for all query points in $\{\mathbf{v}_{\mathcal{S}}\} \times \pi_{\tilde{\mathcal{S}}}(D)$. Rather than first computing each individual prediction and then averaging them, aggregation could be "pushed into the model".



X1	X2	X3
2	7	4
4	2	5
8	1	8

Fig. 3. Example tree and dataset D

Inter-summary commonality: Multiple summaries for the same model can have non-summary or summary attributes in common. This provides additional opportunities for sharing computation *across* summaries.

IV. Summary Computation in Trees

As discussed in Section III-C, we can only improve over the naive algorithm if we work with the internal structure of a model. In this paper we focus on tree-based models, because they are among the most popular models in practice for several reasons. First, trees can handle all attribute types and missing values. Second, the split predicates in tree nodes provide an explanation why the tree made a certain prediction for a given input. Third, tree models like bagged trees, boosted trees, Random Forests, and Groves are among the very best prediction models for both classification and regression problems [8], [9]. Fourth, they are perfectly suited for explanatory analysis because they work well with fairly little tuning. We briefly introduce trees and show how to take advantage of their structure to exploit the observations discussed in Section III-D. The algorithms will be more formally introduced in Section V.

A. Tree Models

Classification and regression trees are some of the oldest and most popular predictive models [10]. A tree model partitions the data space recursively, attempting to achieve partitions with high purity, low mean squared error, or similar goals. Each non-leaf node in the tree splits the data space on some attribute; the leaves contain predictions for points that fall into the corresponding region of the data space.

Figure 3 shows an example tree for attributes X_1, X_2, and X_3. The root corresponds to the entire data space. It contains a split predicate on X_1 ($X_1 < 3$). The root has two children. The left child corresponds to the "half" of the data space containing all records with $X_1 < 3$, while the right child corresponds to the other "half" of the data space with records satisfying $X_1 \geq 3$. Partitioning continues recursively at the children, who can divide their respective sub-spaces further by similarly splitting on any attribute. The leaf predictions in the example are some constants a, b, c, d, and e. Nodes can have more than two children. In the rest of the paper, we will refer to a node in a tree model as `nde`, and its children as $\texttt{nde.chld}_1$, $\texttt{nde.chld}_2$ and so forth.

When making predictions for a point $\mathbf{x}$, the tree is traversed from the root. At each node, the split predicate is evaluated for $\mathbf{x}$. This evaluation, which we will refer to as `nde`.TestSplit($\mathbf{x}$) in the rest of the paper, returns the appropriate child where the traversal continues recursively until a leaf is reached. For example, in the tree of Figure 3, if $\mathbf{x} = (1, 1, 2)$, then the predicate evaluation at the root results in the traversal of the left child ($X_2 < 5$), after which the prediction a is returned. Since trees are well-known, we omit a detailed discussion and refer the reader to Breiman et al. [10].

Trees work well for all types of prediction problems, but the predictive performance of single tree models usually is not competitive with more recent machine learning techniques. This disadvantage has been eliminated by ensemble methods like bagged trees [11], boosted trees [12], Random Forests [13], and Groves [8]. These ensembles consist of many trees and make predictions by adding and/or averaging predictions of all trees in the ensemble. Our techniques can be applied to *all these tree ensembles.*

Many variations of trees have been proposed, including some with multivariate splits (split predicates on more than one variable) and with non-trivial prediction functions in the leaves (rather than a constant value). With the advent of ensemble methods, these more "exotic" trees are rarely used because they (1) are much harder to train and (2) not necessarily produce better models. Furthermore, even the simple ID3 tree can represent any finite discrete-valued function [14]. For all our algorithms, we will therefore focus on trees with univariate splits and constant predictions in the leaves. In Section VIII we outline how our algorithms can be extended to the more complex tree types.

B. Sharing Computation

We show how to speed up computation by leveraging tree structure together with the workload properties discussed in Section III-D. We will focus on single trees; all ideas extend to ensembles by applying them to each tree in the ensemble individually. For ease of presentation, the techniques are explained for a concrete example. The general algorithms are discussed in Section V.

Short circuiting: Recall that to compute a summary, we have to evaluate the model for all points in $V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(D)$. (Notice that projection here returns a multi-set!) We can rewrite this cross-product as $\bigcup_{\mathbf{x}(i) \in D} V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$. In $V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$, the same set of non-summary attribute values, $\pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$, occurs $|V_{\mathcal{S}}|$ many times. To avoid duplicate computation, we can "compress" the original tree for a given $\pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$ as follows.

Consider a tree node `nde` that splits on a non-summary attribute $\tilde{X} \in \tilde{\mathcal{S}}$. Since all query points in $V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$ have the same value $\pi_{\tilde{X}}(\mathbf{x}(i))$, the result of `nde`.TestSplit($\mathbf{x}$) will be the same for all of them. Stated differently, whenever we reach node `nde` during tree traversal for any point in $V_{\mathcal{S}} \times \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$, the traversal will continue with the same child every time. To avoid repeated split predicate evaluations, we can hard-code this traversal path with a *short-circuit pointer.* This pointer connects the parent of `nde` directly to the appropriate child of `nde`, effectively removing `nde` from the tree and pruning away all other sub-trees of `nde`. Short-circuiting can be applied to all tree nodes that split on non-summary attributes.

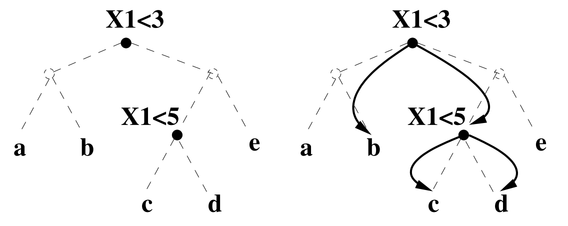


Fig. 4. Retained nodes for summary on X_1 and single-point shortcircuit tree for $(X_2, X_3) = (7, 4)$

To illustrate the idea, consider the example tree in Figure 3 and assume we want to compute a summary on $\mathcal{S} = \{X_1\}$ for dataset D, shown in the same figure. Now consider the set of query points for the first point in D, $(X_1, X_2, X_3) = (2, 7, 4)$. This set of query points is $V_{X_1} \times \{(7, 4)\}$. Predicates $X_2 < 5$ and $X_3 < 7$ will evaluate to false and true, respectively, for each of these query points. Hence we can compress the original tree to obtain the one shown on the right in Figure 4. This compressed tree can now be traversed for every visualization point $x_1 \in V_{X_1}$.

We will refer to such a tree as a *single-point shortcircuit tree*, because it was created based on a single point from D. For the other points in D, we obtain similar trees. For example, for the last point $(8, 1, 8)$, the single-point shortcircuit tree is a stump, consisting of the node with predicate $X_1 < 3$, pointing directly to leaves a (left child) and e (right child).

Instead of short-circuiting a tree on the non-summary attributes, one could alternatively short-circuit on the summary attributes. However, in practice $|\mathcal{S}| \lll |\tilde{\mathcal{S}}|$, because scientists usually care about summaries for visualization ($|\mathcal{S}| = 1$ or $|\mathcal{S}| = 2$). This implies that short-circuiting on non-summary attributes will usually result in better tree compression.

Aggregating shortcircuit trees: Aggregate summaries are computed using terms of the form $\frac{1}{|D|}\sum_{i=1}^{|D|} F(\mathbf{x}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}}(i))$ for each visualization point $\mathbf{x}_{\mathcal{S}}$. With shortcircuit trees as described above, we would run point $\mathbf{x}_{\mathcal{S}}$ through each of the $|D|$ single-point shortcircuit trees obtained for the points in D, then compute the sum of the individual predictions. A much faster algorithm for computing the same value is based on the following observation.

For the sake of simplicity, we will continue the discussion for the concrete example of a summary on X_1. It is easy to see how it generalizes. We can show formally that every single-point shortcircuit tree for the summary on X_1 satisfies the following properties: (1) All nodes with split predicates on non-summary attributes (X_2 and X_3) are effectively eliminated (dotted nodes on the left side in Figure 4). (2) Some leaves and some non-leaf nodes with split predicates on summary attributes (X_1) are retained, each connected directly through a short-circuit pointer to its closest ancestor that splits on a summary attribute. Stated differently, each shortcircuit tree for the summary on X_1 consists of a subset of the bold nodes as marked on the left in Figure 4 and whenever a certain node `nde` is retained in a single-point shortcircuit tree, it is connected to the same ancestor node.

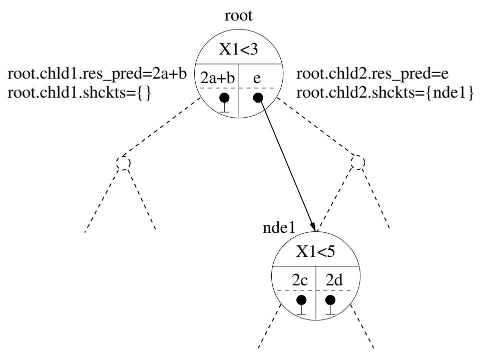


Fig. 5. Multi-point shortcircuit tree for running example

From these observations it follows that all shortcircuit trees for a given summary can be equivalently represented by a *single tree* whose nodes are a subset of the original tree's nodes (in particular its leaves and the nodes that split on summary attributes). Nodes are connected through short-circuit pointers such that each node is directly connected to its closest ancestor that splits on a summary attribute. In addition, for each short-circuit pointer there is a counter that indicates how many of the individual shortcircuit trees contained this pointer. Traversing this tree with a visualization point $\mathbf{x}_{\mathcal{S}} \in V_{\mathcal{S}}$, we directly obtain $\sum_{i=1}^{|D|} F(\mathbf{x}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}}(i))$ by returning the sum of the predictions of all leaves reached, weighted by the count value of the short-circuit pointer pointing to the leaf.

We will refer to this single tree that represents all $|D|$ single-point shortcircuit trees for a given summary as a *multi-point shortcircuit tree* for this summary. As an optimization, we replace a set of short-circuit pointers to leaves in the same sub-tree by the corresponding weighted sum for this sub-tree and store this sum directly in the tree node. Figure 5 shows the corresponding multi-point shortcircuit tree for the running example. For example, the node with predicate $X_1 < 3$ would have two left ("true" branch) short-circuit pointers, one to leaf a with weight 2 (from the single-point trees for points $(4, 2, 5)$ and $(8, 1, 8)$) and one to leaf b with weight 1 (from the single-point tree for point $(2, 7, 4)$). These are replaced by value $2a+b$ in the node.

For convenience, in the rest of the paper whenever we refer to a shortcircuit tree, unless mentioned otherwise, it refers to a multi-point shortcircuit tree.

Inter summary structure: The techniques discussed so far eliminate repetitive work in the computation of a *single* summary. In the model-summary problem as introduced in Section III, a scientist might request a large set of summaries, P, for a given dataset D. In this case, any pair of summaries $\{p_i, p_j\} \in P$ can share computation on attributes in $\mathcal{X} - \{p_i.\mathcal{S} \cup p_j.\mathcal{S}\}$. For example, for $\mathcal{X} = \{X_1, X_2, \ldots, X_{100}\}$, the summaries (X_1, X_2) and (X_1, X_3) can share computation on attributes $X_4, \ldots, X_{100}$. We share computation by generating the multi-point shortcircuit trees for all summaries in a *single* tree traversal.

Algorithm 2 : PointComputeOutput

```
Input: tree node nde, visualization point $\mathbf{v}_{\mathcal{S}}$
1: chld = nde.TestSplit($\mathbf{v}_{\mathcal{S}}$)
2: sum = chld.res_pred
3: for all nde$'$ $\in$ chld.shckts do
4:     sum = sum + PointComputeOutput(nde$'$,$\mathbf{v}_{\mathcal{S}}$)
5: return sum
```

V. Algorithms

We first introduce shortcircuit trees more formally and then present algorithms for creating and querying such trees. For ease of presentation, in the following discussion we will assume that there are no missing values in the set D. Support for missing values will be addressed in Section VIII.

A. Shortcircuit Tree Structure

Let T be a given tree for which we want to compute a summary on $\mathcal{S}$. The corresponding multi-point shortcircuit tree is $\mathrm{T}_{\mathcal{S}}$. Let nde be a node of T that splits on a summary attribute. In $\mathrm{T}_{\mathcal{S}}$ this node maintains two types of information about each subtree rooted at its children. Let chld be a child of node nde. The first type of information is an array of shortcircuit pointers, called *shckts*. Each pointer in this array points to a node in the subtree rooted at chld with the following properties: (1) the node splits on a summary attribute and (2) none of the node's ancestors that are also descendants of node nde splits on a summary attribute. In general, there can be 0 or more pointers in shckts, depending on the tree structure. The second type of information for a subtree rooted at chld is a residual prediction, called *res_pred*, which is the sum of the predictions for all points in $\pi_{\bar{\mathcal{S}}}(D)$ that traversed this subtree and reached a leaf without encountering a node that splits on a summary attribute. (See Figure 5 for an example.)

If the root node of T splits on a non-summary attribute, then $\mathrm{T}_{\bar{\mathcal{S}}}$ also has a new root node with the trivial split predicate "true". It maintains an array of shortcircuit pointers and a residual prediction computed for the entire tree T, as explained above for nodes that split on summary attributes.

For each visualization point $\mathbf{x}_{\mathcal{S}}$, we compute $\sum_{i=1}^{|D|} F(\mathbf{v}_{\mathcal{S}}, \mathbf{x}_{\bar{\mathcal{S}}}(i))$ by traversing the shortcircuit tree using Algorithm 2, starting the traversal at the root. The algorithm determines the appropriate subtree by evaluating the split predicate (line 1); it then recursively traverses all shortcircuit pointers for this subtree. For all accessed nodes, their residual predictions are added.

For a given set of summaries P, the corresponding shortcircuit tree is equivalent to the set of per-summary shortcircuit trees, but all of them merged together into a single structure. More precisely, a node has not just a single (shckts, res_pred) pair. It now has an array of these entries, one array element for every summary that contains the split attribute of the node as a summary attribute.

Algorithm 3 : ShortCircuitTree

```
Input: tree T, summary set $P$, dataset $D$
1: Create new root node new_root
2: for all $\mathbf{x} \in D$ do
3:     new_root.Update( ShortCircuitNode(T,$P$,$\mathbf{x}$) )
4: return new_root
```

Algorithm 4 : ShortCircuitNode

```
Input: tree node nde, set of active summaries $P_{\mathtt{nde}}$, data record $\mathbf{x}$
1: if nde is a leaf then
2:     for all $p \in P_{\mathtt{nde}}$ do
3:         op[$p$] = $\langle$null, nde.prediction$\rangle$
4: else
5:     $P^s_{\mathtt{nde}} = \{p \in P_{\mathtt{nde}} \,|\, \mathtt{nde}.\text{splitAttribute} \in p.\mathcal{S}\}$
6:     for all children chld of node nde do
7:         if chld == nde.TestSplit($\mathbf{x}$) then
8:             /* Pass all summaries to the child for which the
               node predicate is satisfied. */
9:             op = ShortCircuitNode(chld,$P_{\mathtt{nde}}$,$\mathbf{x}$)
10:            for all $p \in P^s_{\mathtt{nde}}$ do
11:                chld[$p$].shckts.add( op[$p$].shckts )
12:                chld[$p$].res_pred.add( op[$p$].res_pred )
13:                op[$p$] = $\langle$nde, null$\rangle$
14:        else /* Pass only summaries that have nde's split attr.
           as a summary attribute to all other children. */
15:            op$^{\mathrm{tmp}}$ = ShortCircuitNode(chld,$P^s_{\mathtt{nde}}$,$\mathbf{x}$)
16:            for all $p \in P^s_{\mathtt{nde}}$ do
17:                chld[$p$].shckts.add( op$^{\mathrm{tmp}}$[$p$].shckts )
18:                chld[$p$].res_pred.add( op$^{\mathrm{tmp}}$[$p$].res_pred )
19: return op
```

B. Generating Shortcircuit Trees

Algorithms 3 and 4 describe the pseudocode for generating the multi-point shortcircuit trees for a set P of summaries. For each record in D, they perform a *single traversal of the tree* (instead of $|P|$ traversals). During this traversal the pointer structure and residual prediction values for *all* summaries in P are generated. We first discuss Algorithm 4 which performs operations at a single node nde in the tree.

To better understand Algorithm 4, let us for now assume that P contains only a single summary on $\mathcal{S}$. In this case the output (called op in the pseudocode) is a single pair consisting of a short-circuit pointer to a node in nde's subtree (possibly nde itself) and a prediction value. Exactly one of them is null. Now let us examine how the output is computed.

If nde is a leaf, then the algorithm returns the node's prediction value and null for the pointer (lines 1–3). If nde is not a leaf, there are two cases. Case 1: If nde splits on a summary attribute, then we need to recursively traverse *all* its children. For each child, this traversal returns either a short-circuit pointer or a residual prediction value for this subtree (if no node splitting on a summary attribute was accessed in the subtree). The returned pointer or prediction value is stored in nde.chld for the corresponding subtree. Since nde splits on a summary attribute, its nearest ancestor that splits on a summary attribute should point to it. Hence the algorithm returns op as a pair containing a pointer to nde and null for the residual prediction value. Case 2: If nde splits on a non-summary attribute, then we only traverse that child for which the split predicate evaluates to true (nde.$\text{TestSplit}(\mathbf{x})$). This recursive call returns either a pointer or a residual prediction as described before, but since nde splits on a non-summary attribute, nde is conceptually deleted from the tree and hence does not store any short-circuit pointers or residual predictions itself. Instead, it returns what it received from its subtree to its ancestor.

Algorithm 4 implements this procedure for an entire set of summaries together during a single tree traversal. The set of active summaries encodes the set of all summaries that reach node nde. Notice also the branches in lines 9–13 and 15–18. For the child that was selected by the split predicate evaluation, all summaries that were active at nde remain active. For the other children, only those summaries for which the split attribute is a summary attribute will be active. Similarly, as lines 10 and 16 indicate, we only update nde's pointers and residual prediction values for those summaries that contain nde's split attribute as a summary attribute. Line 13 ensures that for these summaries, we return a short-circuit pointer to nde to nde's ancestors. For all other summaries, i.e., those for which nde's split attribute is a non-summary attribute, the algorithm simply passes up the call chain the pointer or residual prediction value it received from traversing the subtree rooted at the child that was selected by the split predicate evaluation.

Algorithm 3 calls Algorithm 4 for every point in D on the root node of tree T with the active set of summaries set to P. The return array op from the call to Algorithm 4 is used to update the shortcircuit pointers and residual prediction for the root of the shortcircuit tree for each summary in P (line 3 in Algorithm 3).

C. Algorithm Analysis

Single summary computation. Let $\mathcal{T}$ be an ensemble of trees and $\mathrm{n}(\mathcal{T})$ denote the total number of tree nodes in the ensemble. The naive algorithm queries each tree in the ensemble for each point in $V_{\mathcal{S}} \times \pi_{\bar{\mathcal{S}}}(D)$. Its runtime is $O(|V_{\mathcal{S}}| \cdot |D| \cdot \mathrm{n}(\mathcal{T}))$ and it needs $O(\mathrm{n}(\mathcal{T}))$ space. With single-point shortcircuit trees, for each point in D, we create a short-circuit tree with a single traversal, then query the smaller tree. Hence total computation time is $O(|D| \cdot \mathrm{n}(\mathcal{T}) + |V_{\mathcal{S}}| \cdot |D| \cdot \mathrm{n}(\mathcal{T}'))$, where usually $\mathrm{n}(\mathcal{T}') \ll \mathrm{n}(\mathcal{T})$. If a shortcircuit tree ensemble for a single point is discarded before the next one is generated, space cost is $O(\mathrm{n}(\mathcal{T}) + \mathrm{n}(\mathcal{T}')) = O(\mathrm{n}(\mathcal{T}))$.

A multi-point shortcircuit tree ensemble is constructed by traversing $\mathcal{T}$ for each point in $\pi_{\bar{\mathcal{S}}}(D)$, then predictions are made by running each point in $V_{\mathcal{S}}$ through it. As we discussed earlier, a multi-point shortcircuit tree cannot have more nodes than the original tree, no matter how big D is. This results in a total computation cost of $O(|D| \cdot \mathrm{n}(\mathcal{T}) + |V_{\mathcal{S}}| \cdot \mathrm{n}(\mathcal{T}'))$, where usually $\mathrm{n}(\mathcal{T}') \ll \mathrm{n}(\mathcal{T})$. This is a dramatic improvement over the naive algorithm, essentially reducing cost by a factor in the order of $V_{\mathcal{S}}$ or $|D|$, depending on which term dominates. Space cost is still low at $O(\mathrm{n}(\mathcal{T}) + \mathrm{n}(\mathcal{T}')) = O(\mathrm{n}(\mathcal{T}))$.

Multiple summaries. If $|P|$ summaries are computed one-by-one, the above costs are $|P|$ times higher. When computing all summaries together in a single tree traversal, the asymptotic cost is the same. However, this algorithm evaluates node predicates for each point in D exactly once. And for a point in D, it makes all updates to short-circuit pointers and residual predictions in one visit to a node. This results in significant cost savings in practice.

Desirable properties. Our short-circuiting based algorithms have several important properties. First, as points are added to or deleted from data set D, it is easy to **incrementally maintain** a multi-point shortcircuit tree. Algorithm 3 already computes the tree with a single scan of D, hence when adding a new point to D, we just call line 3 for this point. Similarly, when deleting a point from D, we traverse the *original* tree to determine which short-circuit pointers and nodes are affected. Then we simply decrement the counter values for these short-circuit pointers and reduce the residual predictions in the nodes accordingly. Second, due to their incremental maintainability and fixed space cost, independent of $|D|$, multi-point shortcircuit trees are perfectly suited for **data stream applications**, i.e., where data set D is streaming. However, a change of the original tree model would require a re-computation of the shortcircuit tree.

Third, by decomposing summary computation into multi-point shortcircuit tree construction (does not need visualization points $V_{\mathcal{S}}$) and faster prediction on that more compact tree (only needs $V_{\mathcal{S}}$), our algorithms are ideal for the **online version** of the summary computation problem. In that version, scientists explore a summary interactively, presenting visualization points on-the-fly.

VI. Distributed Computation

To scale model-summary computation to realistic workloads, we have to parallelize both the construction and the evaluation of shortcircuit trees. In this section we propose algorithms that allow us to scale in all important input parameters of summary computation: size of the dataset ($|D|$), number of summaries ($|P|$), number of trees in the ensemble ($|\mathcal{T}|$), and number of visualization points ($|V_{\mathcal{S}}|$) per summary. Following common practice, we say that our algorithm is (linearly) *scalable* in a parameter, if we can achieve the following: With c times the computing resources, we can process a c times larger job (i.e., parameter scaled up by a factor of c) without suffering a significantly higher response time. Based on a careful evaluation of alternatives, we determined that MapReduce [15] would be a perfect fit for parallelizing our approach.

A. MapReduce Overview

The MapReduce framework can be used to implement a two-phase distributed computation on a very large input

Algorithm 5 : GeneralizedMap

```
Input: $P' \subseteq P$, $\mathcal{T}' \subseteq \mathcal{T}$, $D' \subseteq D$
1: for all $\mathrm{T} \in \mathcal{T}'$ do
2:     ShortCircuitTree(T, $P'$, $D'$)
3: for all $p \in P'$ do
4:     for all $\mathbf{v}_\mathcal{S} \in p.V_\mathcal{S}$ do
5:         sum = $\sum_{\mathrm{T}\in\mathcal{T}'}$PointComputeOutput(T, $\mathbf{v}_\mathcal{S}$)
6:         Output(($p$, $\mathbf{v}_\mathcal{S}$), (sum, id($\mathcal{T}'$), $|\mathcal{T}'|$, id($D'$), $|D'|$))
```

Algorithm 6 : Reduce

```
Input: Key = $(p, \mathbf{v}_\mathcal{S})$, Values = $\{(\mathrm{sum}_1, \mathrm{id}_{1,1}, |\mathcal{T}_1'|, \mathrm{id}_{1,2}, |D_1'|)$, $(\mathrm{sum}_2, \mathrm{id}_{2,1}, |\mathcal{T}_2'|, \mathrm{id}_{2,2}, |D_2'|)$,...$\}$
1: AVG = ComputeAVG(($\mathrm{sum}_1, \mathrm{id}_{1,1}, |\mathcal{T}_1'|, \mathrm{id}_{1,2}, |D_1'|$), ($\mathrm{sum}_2, \mathrm{id}_{2,1}, |\mathcal{T}_2'|, \mathrm{id}_{2,2}, |D_2'|$),...)
2: Output(($p$, $\mathbf{v}_\mathcal{S}$), AVG)
```

dataset, which we denote as I. The first phase, *Map*, partitions I into a set of disjoint chunks. A user-specified map function is then applied to each chunk in parallel by a set of machines, called the mappers. The output of map is a set of $<$ key, value $>$ pairs. The second phase, *Reduce*, works on all the key-value pairs produced by the mappers. Conceptually, these pairs are grouped by their key; then each group of values is processed by a single reducer. This happens in parallel on many reducer machines. The output produced by all the reducers is the final output of the distributed computation.

B. Algorithms

There are three major inputs for our algorithm: P (set of plots to be computed, including their visualization points), $\mathcal{T}$ (set of tree models), and D (set of data points). Assume we partition P into two subsets $P_1 \cup P_2 = P$, $P_1 \cap P_2 = \emptyset$; and we similarly partition $\mathcal{T}$ into $\mathcal{T}_1$, $\mathcal{T}_2$ and D into D_1 and D_2. We can run our algorithm on each input combination $(P_i, \mathcal{T}_j, D_k)$, where $i, j, k \in \{1, 2\}$ and then combine the individual outputs into the corresponding output for $(P, \mathcal{T}, D)$.

In general, summary computation with shortcircuit trees can be parallelized by partitioning each input set into smaller subsets ("chunks"), then running our sequential algorithms on each chunk combination, and finally combining the individual outputs. However, the Map function of MapReduce is defined as a one-tuple-at-a-time processor for a single input set. To work around this limitation, we define a function GeneralizedMap (Algorithm 5), which processes any combination $(P', \mathcal{T}', D')$, where $P' \subseteq P$, $\mathcal{T}' \subseteq \mathcal{T}$, and $D' \subseteq D$. When implementing GeneralizedMap with a Map function, we choose the parameter we want the algorithm to scale in as the Map input and load the other two inputs into each mapper before executing the map function. For example, to scale in D, the corresponding Map function would declare D as its input (and hence the MapReduce runtime would assign a chunk of D to each mapper node); and it would load the entire sets P and $\mathcal{T}$ onto each mapper node. Through an appropriate Reducer (Algorithm 6), the output of the MapReduce computation will be the final summary outputs.

GeneralizedMap computes for each visualization point of a summary in P' the total contribution that D' and trees in $\mathcal{T}'$ make to the summary output at that visualization point. Each reduce function receives as key a $(p, \mathbf{v}_\mathcal{S})$ combination and as values a set of partial summary outputs computed over subsets of $\mathcal{T}$ and D. The reduce function performs a simple aggregation of this set to produce the final summary output for the $(p, \mathbf{v}_\mathcal{S})$ combination. The logic for this computation is encoded in the ComputeAVG function and depends on the ensemble type. For a bagged tree ensemble, ComputeAVG would first compute total data set size ($|D|$) and number of trees in the ensemble ($|\mathcal{T}|$) by adding the various $|D_i'|$ and $|\mathcal{T}_i'|$. The algorithm uses the data and ensemble chunk id's to avoid double-counting. (Hence these id's have to be included in the Map output.) Finally ComputeAVG simply adds all the sum_i values and then divides the total sum by $|D| \cdot |\mathcal{T}|$. For other ensembles the computation is similar, e.g., based on weighted sums for boosted trees and additive models like Groves.

VII. Experiments

We compare the performance of our shortcircuit-tree based algorithm against the only existing solution for the model-summary problem—the naive algorithm. This comparison was done on a single processor. Then we demonstrate the scalability of the parallel version of our algorithm on a cluster.

We experimented with different datasets. Due to space constraints, we present results for a single real dataset from bird ecology. These results are representative. In fact, the presented results are for a comparably *small* dataset for several reasons. (1) This demonstrates how expensive summary computation is in practice, even for small data. (2) The larger the data, the larger the models tend to be. Since our technique dramatically reduces model size in the prediction phase, its performance advantage over the naive algorithm increases with larger data. (3) For the parallel algorithm, scalability is not affected by larger input data.

We report results for a real bird ecology dataset obtained from the Avian Knowledge Network (www.avianknowledge.net), Project FeederWatch, that was joined with datasets containing geographical features of observation locations. It covers a geographical region in North America and contains about 90,000 observation records, described by 155 continuous attributes (e.g., time, location, habitat features, climate, census features, elevation). We use 60,000 records to train a model for predicting the probability of observing the Dark-eyed Junco. The model is a bagged tree model consisting of 10 trees and is trained using the IND package [16] using the information gain splitting criterion. Each tree in the ensemble had on average 10,300 non-leaf nodes. This was the best-performing tree-based model trained on the data.

For our experiments, we use the entire 60,000 training records as the D set in summary computations. The training data had missing values on some attributes, which we filled

TABLE I

COMPUTATION TIME (SEC): SINGLE SUMMARY, FREQUENT ATTRIBUTES

$\lvert V_S \rvert$	Naive	ShCkt
100	85.0	3.02 (= 2.96 + 0.06)
400	311.5	3.17 (= 2.97 + 0.20)
625	469.8	3.29 (= 2.96 + 0.33)

TABLE II

COMPUTATION TIME (SEC): SINGLE SUMMARY, INFREQUENT ATTRIBUTES

$\lvert V_S \rvert$	Naive	ShCkt
100	84.8	2.1 (= 2.1 + 0.001)
400	324.5	2.1 (= 2.1 + 0.001)
625	462.7	2.1 (= 2.1 + 0.002)

in using values randomly selected from the attribute's domain. We verified that all leaves of the tree contained data points to guard against degenerate cases. Not handling missing values in the short-circuited trees is not an inherent limitation of our approach, as discussed in Section VIII, just a limitation in our current implementation. The choice of the actual D set does not matter much, because our experiments are only aimed at evaluating the speedups we obtain in summary computations.

A. Single Machine Experiments

Our algorithms are implemented in Java and the experiments reported in this section were run on a Linux machine with a 2.66GHz processor and a JVM heap size of 3GB. All reported times are in seconds. Standard deviations in the reported times were negligible and hence are not reported.

Naive vs. short-circuiting: We begin our evaluation by comparing the benefits of the short-circuiting algorithm (Section V) over the naive algorithm (Section III-C). The first summary, which we refer to as the *frequent attributes summary*, is on a pair of attributes that are frequently used as split attributes in the ensemble (Table I), at a total of 23% of all nodes (12% and 11% for the first and second summary attribute, respectively). The second summary is on a pair of attributes used infrequently in the ensemble (Table II), together accounting for less than 1% of the splits. The tables report the time taken to compute a **single** summary for the naive algorithm (Naive) and the short-circuiting algorithm (ShCkt). For short-circuiting we break total cost down into time for generating the shortcircuit trees and time for querying these trees and generating the summary outputs. In both cases, our algorithm achieves significant speedup, between 30 times and more than 200 times. The speedup increases with the number of visualization points. As the cost breakdown shows, the cost of generating the shortcircuit trees remains constant, while the cost for querying the trees grows linearly with the size of V_S. As expected, for the infrequent attribute case, the trees are compressed more and hence the model evaluation time on the shortcircuit trees is almost zero (Table II). Notice that shortcircuit tree construction is slightly more expensive in the frequent attribute case. This is due to the fact that the shortcircuiting algorithm only needs to traverse a single subtree at nodes that split on non-summary attributes, while it has to traverse both subtrees for nodes splitting on summary attributes. In the frequent attribute case there are many more nodes that split on summary attributes. Small differences in the runtime of the naive algorithm for the same number of visualization points are caused by the fact that the query points are different and the trees are not balanced.

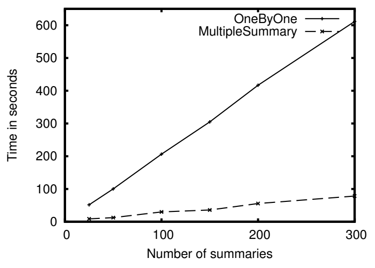


Fig. 6. Multiple Summaries

Multiple Summaries: The last experiment in this section measures the benefits of generating the shortcircuit trees for a set of summaries together (MultipleSummary) rather than one summary at a time (OneByOne). Summary workloads were generated by sampling randomly from the set of all possible one- and two-dimensional summaries over the attributes used by the trees in the ensemble as split attributes. Figure 6 shows that generating shortcircuit trees for multiple plots together is up to 10 times faster than generating them one summary at a time. Stated differently, on top of the 20-200 times speedup for single summary computation, our algorithm achieves another order of magnitude or more improvement over the naive algorithm for workloads with multiple summaries.

B. Distributing Summary Computation

In this section we evaluate how the MapReduce algorithms proposed in Section VI scale in the different input parameters of summary computations. The experiments were run on a cluster with 20 machines. Each machine in the cluster had a 2.66Ghz processor and 8GB RAM and the cluster was running Hadoop v0.18, the open source implementation of MapReduce configured with all the default settings (see `hadoop.apache.org`). All times in this section are job completion times as reported by the Hadoop framework and include all costs such as job setup and teardown times. Deviations in the measured times were small and hence not reported.

Recall that while GeneralizedMap can work on any subset $P' \times D' \times \mathcal{T}'$ of the $P \times D \times \mathcal{T}$ input parameter space, Map only allows a single input file. To scale in a certain parameter, we declare it as the Map input and let the MapReduce system partition the space along this parameter; we do not partition along the other parameters. For example, to scale in D, D is the input to Map and hence each mapper works on chunks $P \times D' \times \mathcal{T}$, where $D' \subseteq D$.

Figures 7, 8 and 9 show how the MapReduce algorithm scales in D, P, and $\mathcal{T}$ respectively. (The graph for scalability

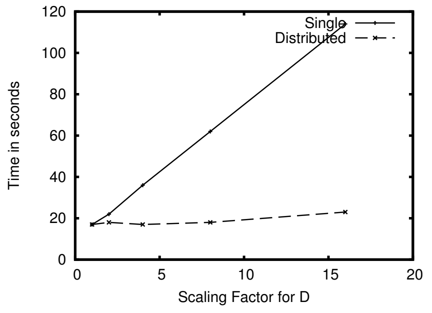


Fig. 7. Scaling in $|D|$

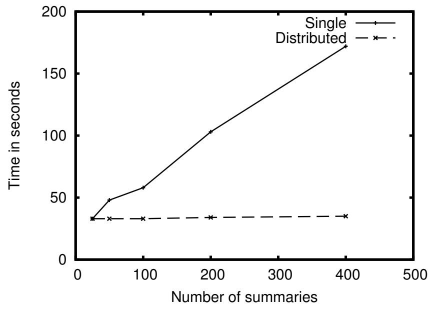


Fig. 8. Scaling in $|P|$

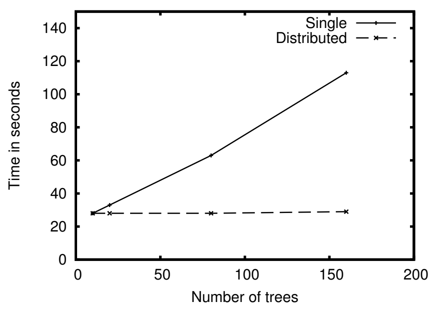


Fig. 9. Scaling in $|\mathcal{T}|$

in $|V_S|$ looks virtually the same, and hence is omitted.)

For Figure 7 we fixed the number of summaries at 1 (the frequent attributes summary) with 900 visualization points. The model is the same 10-tree ensemble as before. The number of reducers was set to 1. We then computed the summary for D of varying size. The larger datasets were generated by simply replicating the D set used in the previous section. This ensures that as the datasets are scaled up, access patterns in the tree remain the same and any increase in cost is only due to processing additional data points. The scaling factor is the size multiplier for D. Line Single shows job completion time when the entire computation is done on a single mapper, for the Distributed graph, we increased the number of mappers in proportion to the scaling factor of the dataset. As we can see, response time remains approximately constant with increasing $|D|$, showing that the framework scales well in D.

For Figure 8, we fixed the dataset D to the usual 60,000 points, used the same 10-tree ensemble, and fixed the number of visualization points at 900. The number of summaries is varied from 20 to 400 (scaling factor 1 to 20). Notice that when partitioning on P, each mapper works on a subset of P, but the entire D and $\mathcal{T}$. Hence we do not need a reduce phase and the number of reducers was set to 0. We use the same method as described above for generating summary workloads.

Figure 9 reports scalability in the size of the ensemble. We fixed the number of summaries to 1 (the frequent attributes summary) with 900 visualization points, and used the usual dataset of 60,000 points. Like the experiment for scaling in D, the number of reducers was set to 1.

VIII. Extensions

For the sake of clarity, we made a few simplifying assumptions in previous sections. Our algorithms generalize naturally and can be extended with additional functionality, as we briefly discuss in this section.

Confidence Intervals: For aggregate summaries, scientists are also interested in obtaining the standard deviation. This can be supported by not only maintaining sums of residual predictions for each child of a node, but also the sum of squares of these residual predictions.

Missing Values: Trees can handle missing values in a query point gracefully by sending partial weights down each sub-tree of a node that splits on an attribute whose value is missing; then computing the weighted average of the corresponding leaves. Our algorithms, which were described for the case that there are no missing values in D, can be extended to support this behavior. We extend Algorithm 4 by associating a weight with each active summary at a node. At the root all weights are 1. When line 7 in Algorithm 4 fails because $\mathtt{x}$ is missing the value for the split attribute, all children are recursively visited with $P_{\mathtt{nde}}$ as the active set. However, when visiting a child $\mathtt{chld}$, the weight for a summary $p \in P_{\mathtt{nde}} - P^s_{\mathtt{nde}}$ is modified to the current weight of p times the fraction of training cases that went into the subtree of $\mathtt{chld}$. Line 3 returns the prediction in the leaf multiplied with the weight of the summary. Also note that op$[p]$.shckts could now contain multiple node pointers and not just one.

Complex Trees: Our approach also generalizes to tree types with multivariate splits at non-leaf nodes and non-trivial functions as leaf predictions. For these trees, the prediction made by a leaf and the predicate evaluation at a non-leaf node may require the values of both the visualization point and some non-summary attributes. This means that the multi-point shortcircuit tree is not guaranteed any more to have only a subset of the nodes of the original tree. In the worst case it degenerates to the equivalent of all single-point shortcircuit trees, which still represents a significant improvement over the naive algorithm. (Time and space complexity of the algorithm would in the worst case be that of the single-point shortcircuit algorithm.) The multisummary optimization can also still be applied.

IX. Related Work

Several papers discuss partial dependence plots [5], [7] for summarization of complex models. Friedman [5] proposes a technique for computing *approximate* partial dependence plots in tree models. This method gives no approximation quality guarantees (a major limitation for use by scientists), it produces accurate summaries only when strong independence assumptions hold, it is limited to partial dependence summaries, and it does not support summary computation in "slices" and "dices" of the data space without generating a new model. Our focus is on efficiently computing the *exact* same summaries as the naive algorithm for all types of summaries and all possible data partitions, no matter what the attribute distribution.

Our work is motivated by OLAP [17], but OLAP techniques cannot be directly applied because our main bottleneck is the evaluation cost of a model. Prediction cubes [18] focus on efficiently (and often approximately) computing predictions of a model for a large region of the data space from many models covering smaller partitions of this region. This approach is problematic for sparse high-dimensional data and it is in some sense the dual of computing region-based summaries for a "large" model.

Other model summary types like dependence diagrams [19] were recently proposed, but data mining research usually concentrates on improving prediction quality [11], [12], [13] or on scalable algorithms for *training* tree models from large data sets [20]. In general, little work has been done to address the performance issues that arise when using complex data mining models for *making predictions*. Bucila et al. [21] propose model compression to reduce model size and computational cost for making predictions. Model prediction time has also been studied in the context of scientific simulations [22]. However, in both cases the original model is approximated, and elimination of redundant computation for summaries is not considered. Our approach is orthogonal in the sense that one could speed up summary computation for such approximate models further by eliminating redundant computation.

The database community has started to explore efficient data management for models [23], [4], but has not considered summary computation from complex models. Sen et al. [24] show how to exploit shared correlations to reduce the cost of inference on probabilistic graphical models. Their work is superficially related, having the same high-level theme of exploiting workload structure to improve query performance.

Multi-query optimization has been studied in many different contexts like relational databases [25] and stream processing [26]. While our algorithms have a similar theme of sharing computation, the structural properties that we exploit are very different.

X. Conclusions and Future Work

We introduced a new data management problem that arises during the analysis of observational data in many domains. We identified various types of structure in summary computation workloads and showed how to exploit it to speed up model-summary computations in tree-based models. Our algorithms produce the exact same results as the naive approach, but are several orders of magnitude faster. Tree-based models are widely used and our algorithms support all types of common model summaries, hence the algorithms in this paper are widely applicable in practice.

There are many directions for future work. First, since algorithms have to be model-specific, new techniques need to be developed for other complex data mining models. Second, scientists need automatic techniques for selecting visualization points that capture all interesting features of a summary. Third, summary computation cost can be further reduced through approximation, but it is only useful to scientists if such techniques provide confidence bounds.

Acknowledgment

This research was supported by the National Science Foundation (NSF) under award 0920869. Additional support came from the Leon Levy Foundation and the NSF (awards 0427914, 0542868, 0612031, 0734857, 0832782). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the sponsors.

We would like to thank the reviewers and the Avian Knowledge Network team, in particular Wesley Hochachka, Steve Kelling, Art Munson, and Giles Hooker for their contributions.

References

[1] S. Kelling, W. M. Hochachka, D. Fink, M. Riedewald, R. Caruana, G. Ballard, and G. Hooker, "Data intensive science: A new paradigm for biodiversity studies," *BioScience*, vol. 57, no. 7, pp. 613–620, 2009.

[2] D. Jensen, A. S. Fast, B. J. Taylor, and M. E. Maier, "Automatic identification of quasi-experimental designs for discovering causal knowledge," in *KDD*, 2008, pp. 372–380.

[3] W. M. Hochachka *et al.*, "Data-mining discovery of pattern and process in ecological systems," *Journal of Wildlife Management*, vol. 71(7), pp. 2427–2437, 2006.

[4] A. Thiagarajan and S. Madden, "Querying continuous functions in a database system," in *SIGMOD*, 2008, pp. 791–804.

[5] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

[6] G. Hooker, *Diagnostics and extrapolation in machine learning*. PhD thesis, Stanford University, 2004.

[7] O. Linton and J. P. Nielsen, "A kernel method of estimating structured nonparametric regression based on marginal integration," *Biometrika*, vol. 82(1), pp. 93–100, 1995.

[8] D. Sorokina, R. Caruana, and M. Riedewald, "Additive groves of regression trees," in *Proc. ECML*, 2007, pp. 323–334.

[9] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proc ICML*, 2006, pp. 161–168.

[10] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and regression trees*. McGraw-Hill, 2000.

[11] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, pp. 123–140, 1996.

[12] R. Schapire, *The boosting approach to machine learning: An overview*. MSRI Workshop on Nonlinear Estimation and Classification, 2001.

[13] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.

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

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] W. Buntine and R. Caruana, *Introduction to ind and recursive partitioning*. Technical Report FIA-91-28, NASA Ames Research Center, 1991.

[17] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[18] B. Chen, L. Chen, Y. Lin, and R. Ramakrishnan, "Prediction cubes," in *VLDB*, 2005, pp. 982–993.

[19] K. Karimi and H. J. Hamilton, "Using dependence diagrams to summarize decision rule sets," in *Advances in AI*, vol. 5032, 2008, pp. 163–172.

[20] J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh, "BOAT-optimistic decision tree construction," in *Proc. SIGMOD*, 1999, pp. 169–180.

[21] C. Bucila, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proc. SIGKDD*, 2006, pp. 535–541.

[22] B. Panda, M. Riedewald, J. Gehrke, and S. B. Pope, "High-speed function approximation." in *Proc. ICDM*, 2007, pp. 613–618.

[23] A. Deshpande and S. Madden, "Mauvedb: Supporting model-based user views in database systems," in *SIGMOD*, 2006, pp. 73–84.

[24] P. Sen, A. Deshpande, and L. Getoor, "Exploiting shared correlations in probabilistic databases," *PVLDB*, vol. 1, no. 1, pp. 809–820, 2008.

[25] T. K. Sellis, "Multiple-query optimization," *ACM TODS*, vol. 13(1), pp. 23–52, 1988.

[26] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White, "Towards expressive publish/subscribe systems." in *EDBT*, 2006, pp. 627–644.