High-Speed Function Approximation

Biswanath Panda, Mirek Riedewald, Johannes Gehrke
Dept. of Computer Science
Cornell University
{bpanda,mirek,johannes}@cs.cornell.edu

Stephen B. Pope
Dept. of Mechanical & Aerospace Eng.
Cornell University
pope@mae.cornell.edu

Abstract

Learning methods for predictive models have traditionally focused on prediction quality and model building time, while prediction time (the time taken to make a prediction) is often ignored. However there is an increasing need for models that are not only accurate, but also make fast predictions. Some of the most accurate models like ensemble models are often too slow to be used in practice. We believe that exploring the tradeoff between prediction time and model accuracy is an exciting new direction for data mining research.

In this paper, we make a first step toward exploring this tradeoff. We introduce a new learning problem where we minimize model prediction time subject to a constraint on model accuracy. Our solution is a generic framework that leverages existing data mining algorithms while taking prediction time into account. We show a first application of our framework to a combustion simulation, and our results show significant improvements over existing methods.

1. Introduction

Predictive models, both for classification and for regression problems, play a major role in machine learning and data mining. After a predictive model is learned from a given set of training cases, it can be used to make predictions for new inputs. Traditionally, learning algorithms for such models have focused on improving prediction quality, e.g., measured by accuracy, root mean squared error (RMSE), area under the ROC curve and other metrics [7]. Research in data mining also considered model building time, i.e., to improve the time it takes to learn predictive models for large or high-dimensional data sets. However, there is another aspect of a predictive model, which is usually ignored by learning algorithms—*prediction time* — the time taken by the model to process an input and make a prediction. Let us describe a concrete application where prediction time is important.

High-dimensional function approximation (HFA) for combustion simulations was recently introduced by [22]. Scientists study how the composition of gases in a combustion chamber changes over time due to chemical reactions. The composition of a gas particle is described by a high-dimensional vector. The simulation consists of a series of time steps. During each time step some particles in the chamber react, causing their compositions to change. This reaction is described by a complex high-dimensional function, which, given a particle's current composition vector and other simulation properties, produces a new composition vector. Combustion simulations usually require up to 10^8 to 10^{10} reaction function evaluations. For most experiments, a single evaluation of the reaction function costs tens of milliseconds of CPU time on a modern PC. This makes running large scale simulations computationally infeasible. Scientists address this problem by building computationally less expensive models that approximate the reaction function within a user defined error tolerance of ϵ [23]. Our work is motivated by these specialized solutions for building models with low prediction time.

Combustion represents one of many physical phenomena studied by scientists using simulation methods. In most cases the mathematical model describing the phenomenon is complex, making it necessary to build approximate models that improve simulation runtime. Recently Bucila et al. [6] observed that ensemble models, while being the most accurate in many scenarios, are often too slow to be used in practice. In addition to scientific simulations, predictive models with low prediction time are also important for online transactions, financial forecasting, fraud detection and numerous other applications where it is important to be both fast and accurate. Building models for applications where prediction time is crucial is the focus of this paper.

One approach to reducing prediction time would be to concentrate on a given data mining model and its construction algorithm and modify them to take prediction time into account. This modification would have to be made for each model/algorithm combination, an arduous task. We instead propose a meta-learning framework that leverages existing

data mining models and model building algorithms. The main idea is a local model approach, where we divide the domain of the learning problem into regions with associated data mining models. The search algorithms in our framework select appropriate regions and models across a large space of possible region/model configurations. Our work shows that this novel local model approach that uses different model types in different parts of the space can significantly reduce prediction time while maintaining high prediction accuracy. We make the following contributions.

- We introduce a new learning problem, *Low Prediction Time Learning*, with the goal to minimize model prediction time while maintaining a user-defined model accuracy. (Section 2)
- We propose a generic framework for Low Prediction Time Learning. Our framework is application-independent and it is not limited to any particular model type or learning algorithm. (Section 3)
- We show how our ideas lead to significant speed-up for real simulation workloads. (Sections 4 and 5)

Section 6 discusses related work and Section 7 concludes the paper.

2. Problem Formulation

We formally define the Low Prediction Time Learning problem and then describe a detailed example, which illustrates several aspects that make the problem challenging.

Assume we are given a distribution $\mathcal{D}$ on R^m and two functions $f : R^m \rightarrow R^n$ and M : $R^m \rightarrow R^n$. We say that M is an (ϵ, δ)-approximation of f with respect to $\mathcal{D}$ if

$$E_{\mathcal{D}}[||f(\mathbf{x}) - \mathrm{M}(\mathbf{x})|| \leq \epsilon] \geq 1 - \delta, \quad (1)$$

where $||$ is some metric. Let $c_{\mathrm{M}}(\mathbf{x})$ be the time taken by M to compute M($\mathbf{x}$).

We can now define the *Low Prediction Time Learning Problem* as follows. Given a set $\mathcal{I} = \{(\mathbf{x}_1, f(\mathbf{x}_1)), (\mathbf{x}_2, f(\mathbf{x}_2)), \ldots, (\mathbf{x}_N, f(\mathbf{x}_N))\}$ find a function M (the *model*) such that M is an (ϵ, δ) approximation of f while minimizing

$$\text{ModelCost} = E_{\mathcal{D}}[c_{\mathrm{M}}(\mathbf{x})]$$

We now describe a simple example to illustrate why Low Prediction Time Learning is an interesting problem. The example will also provide insights into the overall solution described in the next section. Suppose we want to approximate the one dimensional function f shown in Figure 1(A) within a specified (ϵ, δ) error constraint for the distribution $\mathcal{D}$ shown in the figure. Further assume that we have a set of model types denoted by $\mathcal{M}$ that can be used to approximate the function. Let this set consist of polynomials up to

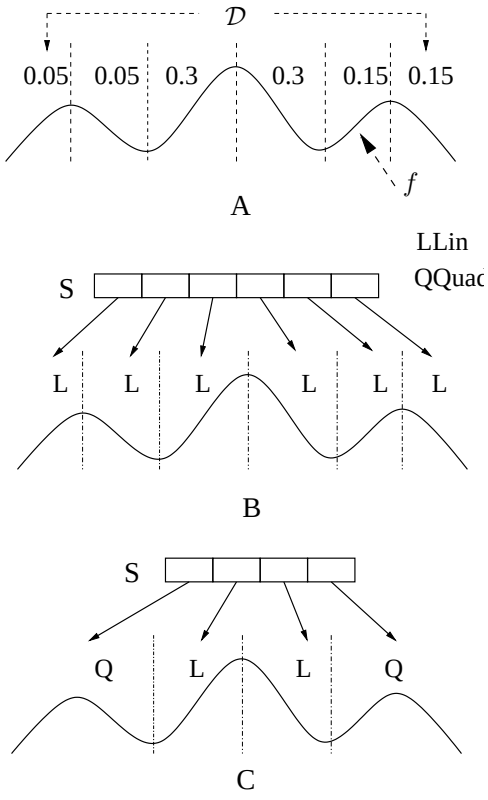

Figure 1. Example

degree 10, that is

$$\mathcal{M} = \{\sum_{i=0,n} a_i \cdot x^i | n = 0, 1, \ldots, 10\}$$

For simplicity, assume the cost of evaluating a polynomial of degree n is equal to the number of multiplication operations, i.e., it is $2n - 1$. (Note that one can compute x^i as $x^{i-1} \cdot x$, hence all powers of x up to the n-th can be obtained with $n - 1$ multiplications.)

Suppose the true function f is a polynomial of degree 10. Then it is clearly possible to approximate f with a polynomial of degree 10 with (ϵ, δ) error. If we approximate f using a polynomial of degree 10, then the model will take 19 time units per prediction.

Observation 1: Assume f can also be approximated within (ϵ, δ) by a 6th-degree polynomial. This reduces model cost to 11 time units per prediction.

Observation 2: Assume further that polynomials of degree less than 6 do not approximate the function well in all parts of the domain. However, lower degree polynomials may work well in some parts of the space. For example, in part (B) of Figure 1 the function domain has been divided into 6 parts and a polynomial of degree 1 is fit in each part. Assume that for all points in a particular partition the linear model in that partition approximates the function within

(ϵ, δ). Therefore, this set of linear models defines another model that overall satisfies the (ϵ, δ) constraint. However, now the prediction time is not just an evaluation of a polynomial, but actually involves two steps. Given a query point, we first have to find the partition that contains the point (*search time*) and then evaluate the polynomial in the partition (*approximation time*).

In order to find a partition containing the query point, we need a search structure S on the partitions. In this example we use a simple linear list S as shown in part B of the figure. For a given query point, the list is scanned until the corresponding partition is found. For simplicity we assume that the search cost is equal to the number of list elements accessed. Hence for the overall prediction time we obtain on expectation $0.05 \cdot 1 + 0.05 \cdot 2 + 0.3 \cdot 3 + 0.3 \cdot 4 + 0.15 \cdot 5 + 0.15 \cdot 6 = 3.9$ units for search and 1 unit for evaluating the corresponding degree-1 polynomial, for a total cost of 4.9 units per query.

Observation 3: Part (C) of Figure 1 shows another partitioning of the function. In this case the first and the last partitions have polynomials of degree 2, while the second and third partitions have polynomials of degree 1. Using an argument similar to Observation 2, assume that this model also satisfies the (ϵ, δ) constraint and again we use a list S to search for partitions. In this case the average approximation time per query is $0.1 \cdot 3 + 0.3 \cdot 1 + 0.3 \cdot 1 + 0.3 \cdot 3 = 1.8$ time units and the average search time per query similarly is $0.1 \cdot 1 + 0.3 \cdot 2 + 0.3 \cdot 3 + 0.3 \cdot 4 = 2.8$ time units, resulting in a total prediction time of 4.6 time units per query.

The example illustrates several interesting tradeoffs for Low Prediction Time Learning.

- Observation 1 showed that at a particular error tolerance there may exist several models of different complexity that can approximate f. As the error tolerance is increased, simpler models can be used, reducing prediction time. We call this the *Accuracy-Prediction Time Tradeoff*.
- Observation 2 showed that there exists a tradeoff between search time and approximation time. Fitting a polynomial of degree 6 resulted in a model with no search time but high approximation time. Partitioning the domain and using a linear model in each partition resulted in model with high search cost and low approximation cost. We call this the *Search-Approximation Time Tradeoff*.
- Observation 3 indicated that exploiting the Search-Approximation Time Tradeoff is challenging because there are many different ways to partition the function domain and build models for each part. In this simple example the difference in prediction times did not vary significantly between the two partitioning schemes, but for more complex functions it can be significant.

In the following sections we will develop a cost-model based optimization framework in order to find models that exploit both the tradeoffs described in this example.

3 Algorithmic Framework

Recall that in the example in the previous section, different partitionings of the input domain and using different model types in the partitions resulted in varying prediction times. In this section we formalize the approach and discuss how to explore the design space of possible regions and models.

3.1 Model Definition

A region-model M for a function $f : R^m \rightarrow R^n$ consists of a set of convex regions $\mathrm{R} = \{r_i | r_i \subseteq R^m\}$, stored in some search structure S, and a mapping Q of regions to standard data mining models such that $\forall r_i \in \mathrm{R} : \mathrm{Q}[r_i] = m_i$. Here m_i is an instantiation of a model type in $\mathcal{M}$, where $\mathcal{M}$ is a set of types of data mining models.

The search structure S supports a Lookup(S,**x**) operation that returns a region $r \in \mathrm{R}$ containing **x**. Given a query point **x** the prediction process consists of the following steps: (1) find $r = \mathrm{Lookup}(\mathrm{S}, \mathbf{x})$, (2) then select $m = \mathrm{Q}[r]$, and (3) compute prediction $m(\mathbf{x})$. We can now revisit the notion of an (ϵ, δ)-approximation of a function f with respect to a region-model. We say that a region-model M is a (ϵ, δ)-approximation of a function f if the following holds:

$$E_{\mathcal{D}}[||f(\mathbf{x}) - \mathrm{Q}[\mathrm{Lookup}(\mathrm{S}, \mathbf{x})](\mathbf{x})|| \leq \epsilon] \geq 1 - \delta.$$

Notice that there might be (R,Q) configurations where some query points are not covered by any of the regions in R, i.e., Lookup(S,**x**) returns no result. To handle this, we assume the existence of a ground truth model of function f, which would be evaluated for such query points. The ground truth model returns $f(\mathbf{x})$ for any $\mathbf{x} \in R^m$ at some (high) cost C. For scientific simulations, this ground truth model is usually a differential equation solver. For traditional machine learning prediction problems this could be a highly accurate, but expensive ensemble model. If such a ground truth model does not exist, we can still apply our approach by simply setting $C = \infty$.

As described earlier, the prediction time per query consists of two costs: search time and approximation time. Let $\mathrm{s_S}(\mathbf{x})$ be the time taken by Lookup to find a region r containing **x** using search structure S. Similarly, let $\mathrm{a}_m(\mathbf{x})$ be the time taken to compute an approximation using model $m = \mathrm{Q}[r]$. Then the expected total prediction time per query can be written as $\mathrm{ModelCost} = E_{\mathcal{D}}[\mathrm{s_S}(\mathbf{x}) + \mathrm{a}_{\mathrm{Q}[r]}(\mathbf{x})]$.

Important properties: We would like to point out some important observations about the model definition above. First, we do not impose any restrictions on what model

types can be included in set $\mathcal{M}$ and what search structure S to use. Any predictive model (e.g. neural nets, decision trees, SVMs) that can represent parts of the target function could be used. Similarly, the search structure could be a spatial index, a point index with post-processing to take region extent into account, a simple list, or any other structure that supports lookup functionality. Second, the models in $\mathcal{M}$ need not be modified to be included in our framework. This way we can leverage existing techniques, without having to modify each technique individually. Third, "global models", i.e., those where a single model is learned for the entire function domain, are a special case of our model definition. For a global model search time is zero. Finally, it has been observed that models similar to ours may exhibit variance because of discontinuities at region boundaries, that is addressed using a more general mixture model framework [15]. We discuss ways to address this in Section 7.

3.2. Algorithms

Let $\mathcal{I}$ denote a set of input points with known function values. We partition this set into a training set ($\mathcal{T}$) and a validation set ($\mathcal{V}$) for model building. Generalization error and model cost (ModelCost) will be measured on an independent test set not used for model building.

An exhaustive exploration of all possible combinations of region partitioning, models used for each region, and index for managing regions, is practically infeasible. To reduce the complexity, we divide the problem into smaller sub-problems. In particular, our algorithm has two major steps:

1. Generate a set of regions and find the best model for each region.
2. For each index structure under consideration, select the set of region-model pairs that minimizes expected prediction time for this index. Return the best solution.

These two steps that we call *Region-Model Candidate Set Selection* and *Region-Model Selection* are discussed in more detail below.

Region-Model Candidate Set Selection: Any subset of points in $\mathcal{T}$ could be connected as a candidate region, resulting in a number of regions exponential in the training set size. We therefore have to resort to heuristics for generating "the most promising" candidate regions. To reduce the search space, without being overly restrictive, we propose the following general approach. Assume we are given a set of relatively small regions, which we refer to as *base regions*. These base regions could be obtained from a regular grid partitioning of the space, from the leaves in a regression tree [5], or based on ISAT's regions of accuracy [23]. Notice that base regions do not need to be disjoint. We restrict region candidates to be either base regions or larger *derived* regions, which are the union of some base regions that are *near* each other. We will present a concrete algorithm in Section 4.

For each region under consideration, base region or derived, the next step is to find a local model for that region. This is described in Algorithm 1. Using the points from $\mathcal{T}$ and $\mathcal{V}$ that lie in a given region r (called $\mathcal{T}_r$ and $\mathcal{V}_r$), the algorithm finds the lowest prediction time (t_m) model instantiation (m) from $\mathcal{M}$ that can be learned in the region and produces ϵ-approximations for at least $1-\delta$ fraction of the points in $\mathcal{V}_r$.

Two observations make the implementation of Algorithm 1 efficient. First, $\mathcal{T}_r$ and $\mathcal{V}_r$ for a derived region can be approximated by merging the corresponding lists from base regions. Second, it is common for more complex models to have higher prediction time. Rather, than trying all models in a region we sort $\mathcal{M}$ in increasing order of model complexity and iterate the list till a model satisfying the error constraint is found.

Region-Model Selection: The region-model generation algorithm produces a set with elements of the form $(r_i, m_i, \mathrm{t}_{m_i})$. We call this set of region model pairs RM. Notice that each of the models in RM satisfies the (ϵ, δ) error constraint for its region. Region-model selection involves selecting a subset of RM and initializing a model M (as defined in Section 3.1) that has lowest prediction time. Therefore, selection finds a model that minimizes $\sum_{\mathbf{x}\in\mathcal{V}}(\mathrm{s_S}(\mathbf{x}) + \mathrm{a}_{\mathrm{Q}[r]}(\mathbf{x}))$. There are two important observations about this problem formulation.

- A selected subset of regions need not cover all points in $\mathcal{V}$. The ground truth model (Section 3.1) will be used to make predictions for such non-covered points. A ground truth model with approximation time of ∞ forces the selection algorithm to search for subsets of RM that completely cover the function domain.
- Algorithm 1 guarantees that every region-model pair in RM satisfies the (ϵ, δ) error constraint. However, if regions are allowed to overlap this does not guarantee that the (ϵ, δ) error constraint will hold for a model M consisting of a subset of RM. In our experience having all regions satisfy the error constraint leads to tighter error for M. This is not surprising, because M will only have worse error for some corner cases. We do not elaborate on this further due to space constraints. As the experiments show, in practice enforcing (ϵ, δ) for each region usually leads to better global error.

Several factors make the region-model selection problem difficult. First, lookup cost in a search structure depends on the properties of the regions it stores like their degree of overlap, extent, and orientation. If multiple regions in the search structure S contain a given query point, then approximation cost depends on the region-model pair that will be finally used in the prediction. These issues aside, we

Algorithm 1 : Model Generation

Require: Training set $\mathcal{T}$, Validation Set $\mathcal{V}$, Region r, Model Set $\mathcal{M}$, Error ϵ, Error Rate δ

1: $\mathcal{T}_r = \{(\mathbf{x}, f(\mathbf{x}))|\mathbf{x} \in r \wedge (\mathbf{x}, f(\mathbf{x})) \in \mathcal{T}\}$
2: $\mathcal{V}_r = \{(\mathbf{x}, f(\mathbf{x}))|\mathbf{x} \in r \wedge (\mathbf{x}, f(\mathbf{x})) \in \mathcal{V}\}$
3: **for all** model types $\in \mathcal{M}$ in ascending order of complexity **do**
4: **if** model instantiation m using $\mathcal{T}_r$ exists **then**
5: Y=$\{(\mathbf{x}, f(\mathbf{x}))|(\mathbf{x}, f(\mathbf{x})) \in \mathcal{V}_r \wedge ||m(\mathbf{x}) - f(\mathbf{x})|| \leq \epsilon\}$
6: **if** $\frac{|Y|}{|\mathcal{V}_r|} > 1 - \delta$ **then**
7: **return** (m,t$_m$)
8: **return** "No model found"

can show that even if we make very restrictive assumptions about the search time and approximation time of a query point, the region model selection problem is very hard.

Theorem 1. *For a non-trivial set of region-model pairs RM, selecting the subset of region-model pairs from RM, such that expected prediction time is minimized, is NP-hard.*

We skip the proof due to space constraints. Given the complexity of the selection problem, we use a greedy heuristic, shown in Algorithm 2. The algorithm starts out with an initial solution of base regions. This initial solution is biased toward high search cost and low approximation cost. In each step the algorithm replaces a set of regions in the current solution with a larger region from the set of candidate regions, such that the larger region covers all the removed regions. This is done greedily by selecting the region that brings about the largest reduction in prediction time. The algorithm stops when no more improvement is possible.

Notice that Algorithm 2 assumes the existence of a cost function ($\mathcal{C}$), which, given a set of region-model pairs and a validation set $\mathcal{V}$, returns the prediction time of the best model that can be created using the given region-model pairs. Finding such a cost function is challenging, because of reasons pointed out earlier. We will discuss this in more detail in the next section.

4. Instantiations

There are many ways to instantiate the above framework, differing in how base regions are generated and merged and the search structure used to store the regions. One can define a grid-based partitioning of the function domain [3], attempt to merge adjacent grid cells and use a search structure that performs a binary search along each dimension to find the cell the query point lies in. Another possible instantiation is a regression tree style partitioning of the function domain with a binary tree search structure. In this case

Algorithm 2 : Greedy Region Selection

Require: RM, Validation Set $\mathcal{V}$, Cost function $\mathcal{C}$

1: Sol ($\subseteq$ RM) = $\{(r_i, m_i, \mathrm{t}_{m_i})|r_i$ is a base region$\}$
2: Cost = $\mathcal{C}$(Sol)
3: **while** Cost improves **do**
4: TempSol=$\{\}$
5: **for all** $(r, m, \mathrm{t}_m) \in S \wedge (r, m, \mathrm{t}_m) \notin$ Sol **do**
6: Rem = $\{(r_i, m_i, \mathrm{t}_{m_i})|(r_i, m_i, \mathrm{t}_{m_i}) \in \text{Sol} \wedge r_i \subseteq r\}$
7: tSol$_r$ = Sol + (r, m, t_m) - Rem
8: tCost$_r$ = $\mathcal{C}$(tSol$_r$)
9: TempSol = TempSol $\cup$(tSol$_r$, tCost$_r$)
10: **if** $\exists$(tSol$_r$, tCost$_r$) $\in$ TempSol s.t. tCost$_r$ $<$ Cost **then**
11: (Sol,Cost)=(tSol$_r$, tCost$_r$)
12: S= Regions in Sol, Q= Region-Model map for Sol
13: **return** S,Q

the base regions correspond to the leaf nodes of a regression tree (T) like CART [5]. The region merge process could then attempt to merge a subtree of T into a single region with a more complex model. Intuitively the selection algorithm would prune away subtrees of T whenever it is cheaper to use the complex model in the merged region to make a prediction compared to traversing the subtree and using the simpler models in the leaves. For both the grid-based and the regression tree approach defining cost function $\mathcal{C}$ is fairly straightforward and we omit the details.

A third and more general instantiation is to treat each individual point in $\mathcal{I}$ as a base region and define a merge that creates regions enclosing the 1, 2,..., n nearest neighbors of a point. In this case the set of regions can have arbitrary shape, size, overlap; and the search structure (S) can be any high dimensional index. We discuss a variation of this idea for the combustion simulation where scientists build models with flexible region definitions.

4.1. Simulation Instantiation

The ISAT algorithm used by the domain scientists [22] approximates the combustion reaction function by a set of (possibly overlapping) high-dimensional ellipsoids with linear models inside these ellipsoids. These regions are obtained based on selective evaluations of the reaction function, which is the ground truth model for this application.

To ensure that the ellipsoids satisfy the model definition in Section 3.1, we use a slightly modified version of the algorithm [30]. The main modification is a stricter error control mechanism that periodically checks existing regions in the model and updates region boundaries to not include parts of the space where the model is producing poor approximations. Studies also indicated that hyper-rectangular regions work at least as well as ellipsoids, we will therefore

use hyper-rectangular base regions. In the remainder of the paper, this modified algorithm is referred to as the ISAT algorithm.

Domain scientists also observed that their long-running simulations ($> 10^9$ queries) almost always have the following two properties. First, the future query distribution of the simulation can be fairly accurately estimated after a few million queries. Second, simulation time is dominated by model prediction time, i.e., model construction and maintenance time are negligible. We describe the instantiation of our framework for such simulations.

Without loss of generality we model the simulation as a 2-phase process. During the first phase (a few million queries) the ISAT algorithm is run. This algorithm produces a set of base regions in the function domain with a similar model in each region. In order to create this set of region-model pairs, the ISAT algorithm has to evaluate the reaction function for some query points. These points will be used as the training and validation data for our technique ($\mathcal{I}$). At the end of the first phase we apply our framework using $\mathcal{I}$ as the input data set and build a new model optimized for prediction time. This model is used for the rest of the simulation. Long-running simulations need not have exactly two phases; in that case the above procedure can be repeated periodically. Note that the framework instantiation for the combustion simulation can also be applied to improve prediction time in a *traditional supervised learning model*, using the training data explicitly provided.

Our instantiation for the combustion problem starts with the set of regions created by the ISAT algorithm during phase one as the base regions. Larger regions are created by merging a base region with its nearest neighbors. Specifically, for each base region r, we add the following derived regions: r merged with its first nearest base region, r merged with its two nearest base regions, and so on until some upper limit n of neighbors. Duplicate derived regions are eliminated. Since the base regions are hyper-rectangles, we define a derived region as the smallest bounding hyper-rectangle of the merged base regions. Conceptually, we do not need to use ISAT's regions as base regions, and could use individual points in $\mathcal{I}$ as base regions instead. However, if cardinality of $\mathcal{I}$ is large this would make nearest neighbor search costly.

Having defined the region creation process, the next step is to find models for each region (Algorithm 1). We now turn our attention to the major challenge for the next step—defining cost function $\mathcal{C}$.

Cost Function ($\mathcal{C}$): For high dimensional indexes, it is difficult to accurately estimate the search cost of a query just based on the set of regions to be stored, without actually building the index. Unfortunately, building the index for each iteration of the greedy region selection algorithm (step 8 in Algorithm 2) is very expensive.

We discuss cheaper alternatives for selected index structures. Due to space constraints we omit implementation details. The main idea is to take advantage of two properties of the problem. (1) The selection process picks region-models from a fixed set and optimizes the solution for a fixed set of points ($\mathcal{V}$). Hence we can precompute information like the subset of $\mathcal{V}$ in each region. (2) At each step the algorithm leaves most of the solution unchanged and only replaces a small set of regions with a single larger region. We can leverage this property for incremental computation.

Random List stores regions in a simple list. The lookup operation scans the list from the beginning until a region containing the query point is found. While lists are not sophisticated index structures, linear scans are known to perform well for disk-based accesses in high dimensions [28] and also as in-memory data structures for combustion simulations [22].

Different orders of regions in the list will result in different prediction costs. Given a set of regions it is infeasible to try all possible orders to find the best one. The idea behind the random list approach is to compute and minimize the expected cost assuming all region orders are equally likely, and then to pick the best order for the set of regions with the lowest expected cost.

Given a selected set of region model pairs of size |S|, the cost function computes $\sum_{\mathbf{x}_i \in \mathcal{V}} (\frac{|\mathrm{S}|}{f_i+1} + Avg(\mathrm{t}_{m_1} \ldots \mathrm{t}_{m_{f_i}}))$. The intuition for the formula is as follows. For a set of regions, if a query point lies in multiple regions, then in any random order of the list it is very likely that a region containing the query point is found early. Therefore, the search time for a query point is approximated as $\frac{|\mathrm{S}|}{f_i+1}$ where f_i is the number of regions that query point $\mathbf{x}_i$ lies in. The approximation time for a query point is simply the average of the cost of the models in the regions that the query point lies in (each one is equally likely to be found first in the list). After the selection algorithm finds a set of regions with the lowest expected cost, we try a few different sort orders of these regions and pick one with the lowest cost.

MFU List: In practice it is often a good heuristic to store the most frequently queried regions in front of the list. This strategy is called Most Frequently Used (MFU). Notice that this need not be an optimal order, because the model in a frequently accessed region might be expensive and the query point might also be covered by a region with a cheaper model later in the list. We use the validation set $\mathcal{V}$ to estimate the fraction of future queries that will fall into a given region.

In a MFU list the order in which a set of regions will be stored is known and therefore search and approximation cost for all query points can be accurately computed. In this case an efficient implementation exists by first sorting all candidate regions in RM according to the number of points in $\mathcal{V}$ that they contain.

RTree: For hierarchical indexes like the RTree, it is known that finding accurate cost models for high-dimensional data is very difficult [18]. Fortunately, for our technique we do not need absolute costs, but rather an estimate of the net benefit of merging a set of regions into a single region. In this section we propose a fairly simple and robust heuristic that can be used for optimizing any index structure which prunes search space by building a hierarchical structure on the set of regions being indexed. We describe the heuristic for the RTree [14], a popular index for spatial data. One can develop more accurate cost models for different index structures but our aim is to show that even a simple heuristic works well for improving model prediction time. More sophisticated cost models can be easily plugged into our algorithm (Line 8 in Algorithm 2).

The RTree is a balanced tree structure. Nodes in the tree correspond to hyper-rectangles in the data space. If the tree indexes hyper-rectangles, a leaf node stores actual data objects (up to a specified maximum), while a non-leaf node stores the minimum bounding box of hyper-rectangles in its subtree. During a search, all subtrees whose bounding boxes contain the query point are examined, hence the search cost is determined by number of hyper-rectangles examined till a data object containing the point is found, often called the *false positive rate* of an index. A tree can have a non-zero false positive rate because in high dimensions it is difficult to partition objects well, causing the bounding boxes of non-leaf nodes to overlap. This results in multiple search paths in the tree for a given query point and some paths may not have a data object containing the query point (hence false positives). Our goal is to estimate the reduction in false positives for queries if a region merge is done in Lines 7 and 8 of Algorithm 2. This cost reduction has to be compared with the cost increase associated with a more complex model in the larger merged region.

We estimate the benefit of merging as follows. Assume the RTree on average has k false positives for a query. Since RTrees (and any hierarchical index) tend to cluster nearby objects, all false positives of a query tend to be in the neighborhood of the query. Hence, if we merge some neighboring regions, then nearby query points will see a reduction in their false positive rate because some of their false positives have been merged. We estimate this reduction in false positives by defining a neighborhood around the merged regions, such that it contains all queries that are affected by the merge.

The order in which these affected queries will access regions in the tree depends on the actual tree layout. Lacking further knowledge, we assume that all regions in the neighborhood are accessed in some random order. Hence we use the random list cost model (see above) to estimate the benefit of a region merge in the affected neighborhood. The main challenge is to select the correct neighborhood. We define the neighborhood by selecting a small number of nearest neighbors, parameterized by γ, of each region that participates in the merge. Details on γ and the performance of the heuristic are described in the experiments.

Name	Description
ISAT	ISAT algorithm
Opt	Proposed optimization algorithm
C	Constant model
L	Linear model
Q	Second order model
$\|S\|$	Index size grouped by model type
k	Average number of false positives
Obs δ	Observed δ on test set
Search Time	Total cost of index lookup
Approx Time	Total cost of model evaluation
Total Time	Total prediction time
StdDev	Standard deviation of total time

Table 1. Legend

5. Experiments

As a proof of concept, we implemented and tested our approach for the combustion simulation application. We use libraries and data from a Hydrogen+Air simulation provided by the authors of [22]. The dataset comprises 5 million simulation query points. Each query point is a 10 dimensional composition vector. The reaction function that describes the simulation in this case is a high dimensional function $f : R^{10} \rightarrow R^{11}$.

The overall setup is as follows. We run the ISAT algorithm on the first 3 million query points to generate the base regions and training/validation data set $\mathcal{I}$, which are used by our algorithm as discussed in Section 4.1. A random sample of $\approx 2 \times 10^5$ query points from the last 2 million queries is used as an independent test set. We compare total simulation time on the test set against the original ISAT model as it is currently used by the domain scientists.

All experiments used a $70-30$ split of $\mathcal{I}$ into training ($\mathcal{T}$) and validation ($\mathcal{V}$) set and $\delta = 0.1$. For each base region, 8 derived regions are created by merging the base region with its 1,2,...,8 nearest neighbor base regions. For a fair comparison we use exactly the same data that ISAT uses for model building. Notice that $\mathcal{I}$ usually is not exactly a uniform sample of the query points due to peculiarities of the ISAT algorithm. This puts our algorithm at a slight disadvantage, but overall we did not find significant differences between the distribution of $\mathcal{I}$ and the test set. All experiments were run on a Windows XP PC with a 2.79GHz processor and 8GB RAM.

ExptNo:(S, ϵ)	Method	$\mathcal{M}$	|S|	k	Obs δ	Search Time(ms)	Approx Time(ms)	Total Time(ms)	StdDev (ms)
1: $(\mathrm{RL}, 5 \times 10^{-3})$									
	ISAT	L	L:63	26	0.01	623	337	960	68
	Opt	L,Q	L:28,Q:9	6	0.005	114	434	548	-
	Only S	L,Q	L:26,Q:41	1	0.0002	84	1750	1834	-
2: $(\mathrm{RL}, 5 \times 10^{-5})$									
	ISAT	L	L:2263	977	0.05	20477	383	20860	2983
	Opt	L,Q	L:1430,Q:332	122	0.01	2071	1620	3691	-
3: $(\mathrm{MFU}, 3 \times 10^{-3})$									
	ISAT	C	C:2226	113	0.08	2367	93	2460	-
	Opt	C,L	C:1362,L:115	19	0.003	414	342	756	-
4: $(\mathrm{RTree}, 3 \times 10^{-3})$									
	ISAT	C	C:2226	212	0.11	15530	78	15608	819
	Opt	C,L	C:687,L:229	92	0.07	6751	266	7017	-
5: $(\mathrm{RTree}, 5 \times 10^{-5})$									
	ISAT	L	L:2263	166	0.06	12238	380	12618	1327
	Opt	L,Q	L:1986,Q:36	124	0.05	8927	385	9312	-

Table 2. Results Summary

γ	|S|	Avg Total Time(ms)	StdDev(ms)
ISAT	2226	15608	819
0.004	1210	10584	1303
0.008	916	8350	736
0.012	**802**	**8050**	**950**
0.02	653	6151	667
0.03	555	5362	773

Table 3. Neighborhood Effect

5.1. Results

We ran simulations using different values of ϵ, index structures and model types ($\mathcal{M}$). Table 2 summarizes the results; variables are explained in Table 1. All measurements are on the test set and times reported are in milliseconds, rounded to the nearest integer.

Experiment 1 is for $\epsilon = 5 \times 10^{-3}$ and the Random List (RL) index. ISAT built regions with linear models (L)[1] and our framework used both linear and quadratic (Q) models. ISAT created 63 regions. Since index size and search cost are small in this case, our method (Opt) does not merge many of the linear regions into quadratic ones (only 9). Nevertheless a significant reduction in prediction cost by $\approx 30\%$ is achieved. The increase in approximation cost (some query points are approximated using quadratic models) is offset by the decrease in search cost. Recall that our algorithm for a random list tries a few random orders and returns the best as the solution. For ISAT there is no optimization algorithm for selecting the best list order, therefore we report average cost across 30 different random sort orders and standard deviation.

To show that both approximation and search cost must be considered for prediction time optimization, we repeated Experiment 1 using a simpler optimization goal—only minimize search cost ("Only S"). In this case the selection algorithm aggressively merges regions to cover validation points with the smallest number of derived regions containing quadratic models. As the results show, the additional decrease in search cost is not significant enough to offset the higher approximation cost. A surprising observation in this experiment is that the number of regions created by "Only S" is greater than for ISAT, even though the selection algorithm usually *replaces* a set of regions with a larger region. This happens because it is possible to select a candidate region that does not completely contain any regions in the current solution, but significantly overlaps with a lot of them (i.e., Rem={} in Line 6 and 7 of Algorithm 1). Adding such a region increases list size but may still reduce expected search cost per query as some query points now are covered by multiple regions (recall that search cost=$\frac{|\mathrm{S}|}{f_i} + 1$).

Experiment 2 uses the same setup as Experiment 1 but with $\epsilon = 5 \times 10^{-5}$. As ϵ is stricter, it is not surprising that ISAT creates a larger number of regions and hence search cost dominates prediction time. Opt in this case more aggressively selects regions with quadratic models, causing the approximation time to increase significantly. An even larger decrease in search time results in $\approx 70\%$ improvement in total time.

When we repeated Experiments 1 and 2 for the MFU

[1] ISAT always uses the same model in every region; it must be specified when the simulation starts.

List, Opt did not merge any regions and simply continued to use the base regions created by ISAT. The reason is the skewed distribution of query points over base regions. The first few regions in the list account for the vast majority of accesses, resulting in very low search cost. Hence for the MFU List the benefit of merging regions would be too low to offset the higher approximation cost of a quadratic model in a merged region. Stated differently, if the search cost is low, then it is preferable to stay with the simplest models in each region.

To show more clearly that Opt makes the right decisions even for the MFU List, we performed **Experiment 3** using a MFU List and $\epsilon = 3 \times 10^{-3}$, but this time setting ISAT to produce base regions with *constant models* (C). Now Opt chooses between linear and constant models. Because constant models lead to smaller base regions (to guarantee the error), the list has now more elements and hence higher search cost. Again Opt automatically makes the right choice to merge regions into larger ones with linear models, significantly improving cost.

Experiments 4 and 5 report results for the RTree index. In Section 4.1 we introduced parameter γ to control the affected neighborhood size of a region merge. We use a simple heuristic to set γ. First, an RTree is built from the base regions. Points in $\mathcal{V}$ are queried using the tree and the average number of base regions probed per query (k_l) is recorded. Based on the assumption that if on average k_l leaves are scanned per query, this corresponds to a random list of size $2 \cdot k_l$ being examined, we set γ such that at most[2] $2 \cdot k_l$ regions are affected by a merge, when the current solution has only base regions. Once initialized, we do not change γ. Hence affected neighborhood size decreases with index size.

Using this heuristic and otherwise the same setup as Experiment 3, in **Experiment 4** Opt shows $\approx 50\%$ improvement over ISAT. Our RTree implementation [2] uses a one-by-one insertion scheme and different insertion orders can lead to slightly different RTrees. Therefore, for ISAT we report average measurements and standard deviation in total runtime across 10 different insertion orders. Opt uses its cost model to select the best among a few different RTree insertion orders.

Experiment 5 uses the same setup as Experiment 1, but with an RTree. The improvement in runtime is comparably small, suggesting either that linear models are good or poor choice of γ. Notice that even though approximation time remains almost unchanged, search cost actually decreases. This can be explained by regions that contain very few query points next to heavily accessed regions. Merging the lightly accessed regions does not change approximation cost. But it does help reduce search cost for the heavily accessed region, because less false positives are encountered.

[2]We say "at most" because we scale γ according to the size of the merge. Larger merges affect larger neighborhoods.

As we mentioned earlier, the goal of this paper is not to develop the most accurate cost models for high dimensional indexes. Rather, we wanted to provide a proof of concept that pursuing optimization of prediction time is worthwhile. More accurate cost models can easily be leveraged in our framework. However, we end the discussion here with a micro benchmark that shows that the proposed simple Rtree cost model is robust (i.e., not sensitive to γ). Table 3 shows index size, average runtime and standard deviation (across 10 different insertion orders) for Rtrees optimized using different values of γ. For instance, $\gamma = 0.012$ for an index of size $|\mathrm{S}|$ implies $0.012 \cdot |\mathrm{S}|$ nearest neighbors of each replaced region are assumed to be affected be a region merge. These results are for the setup of Experiment 4, with the line in bold face representing the default γ value used in that experiment. Results were similar for Experiment 5, hence are not reported here explicitly.

The first conclusion from the results is that index size decreases with increasing γ. This is expected since a larger neighborhood size implies that the tree is expected to have a larger false positive rate and hence our algorithm predicts more cost savings by merging regions and using complex models. While it is clear from the table that total time is not very sensitive to γ, in this case it tends to improve as γ increases. This is an artifact of the setup and happens because here RTree search cost far exceeds approximation cost. As a result, Opt uses linear models in most regions. As we increase γ, Opt merges more regions. But this only insignificantly increases approximation cost, because most regions are already linear for smaller γ. However, search cost may still decrease significantly.

Discussion. Our experiments show that different indexes and model types work well in different simulation settings. Our proposed method (Opt) correctly and automatically captures the tradeoffs in the problem and effectively adapts the model to the index and simulation parameters. Our method does not improve runtime at the cost of degrading prediction quality (see δ values in Table 2). In fact, in most cases Opt produces a δ value better than the original ISAT model. This is because our algorithm performs robust error control by checking each region-model pair before admitting it as a candidate for region selection. ISAT on the other hand only randomly checks regions for error. Finally, in order for the method to be useful in practice, it should not generate a significant computational overhead for the simulation. In all our experiments the cost of the optimization algorithm was negligible compared to the total cost of a long-running simulation as used by the domain scientists.

6. Related Work

Closest to our work is recent work on model compression [6] where an ensemble model is approximated with a neural network to improve prediction time. Robot motion planning algorithms developed techniques to optimize prediction time in local regression models [24]. However, none of the prior work formalizes the learning problem and examines the various tradeoffs we discuss.

There is lot of work on local models. Instance based learning [19] is a special class of local models where rather than explicitly defining regions, function values at unknown points are interpolated from neighboring training samples. A regression tree [5] creates regions in the function domain. Regression trees are often pruned for accuracy [5], and our framework when applied to a regression tree uses the same idea for improving prediction time. No work has focused on optimizing regression trees for prediction time. [8, 17] propose new split criteria for accuracy, [17, 26] use complex models in the leaves (again for accuracy), [8] assumes that shorter trees are easy to interpret and always creates the most complex model for a subtree, [15, 13] propose methods to reduce variance in regression tree models, and [11] optimizes tree construction costs. We optimize and build a more general class of models and the regression tree is only an instance of a model in the class.

Existing techniques in the combustion community use region-models that differ in the types of regions and models used [23, 3, 9, 22, 27]. However, no work in this community addresses search and approximation costs together.

Numerous methods have been proposed for finding cost models for high dimensional index structures with different focus from our work [1, 4, 16, 29, 12, 21, 10, 20, 25, 18].

7. Conclusions and Future Work

We introduced and formalized the low prediction time learning problem. We proposed a general framework that leverages existing data mining models to minimize model prediction time and used it to significantly speed up a scientific application. Understanding how existing data mining models can be optimized for prediction time is an interesting direction for future research.

Future directions include reduction of the model variance using the overlap among regions (recall our remark from 3.1) and periodic application of our framework for long running combustion simulations.

References

[1] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest neighbor searching. In *Symposium on Computational Geometry*, pages 336–344, 1995.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD*, pages 322–331, 1990.

[3] J. B. Bell, N. J. Brown, M. S. Day, M. Frenklach, J. F. Grcar, R. M. Propp, and S. R. Tonse. Scaling and efficiency of PRISM in adaptive simulations of turbulent premixed flames. In *28th International Combustion Symposium*, 2000.

[4] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *ACM PODS*, pages 78–86, 1997.

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

[6] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. In *ACM SIGKDD*, pages 535–541, 2006.

[7] R. Caruana and A. Niculescu-Mizil. Data mining in metric space: an empirical analysis of supervised learning performance criteria. In *ACM SIGKDD*, pages 69–78, 2004.

[8] P. Chaudhuri, M. C. Huang, W. Y. Loh, and R. Yao. Piecewise-polynomial regression trees. *Statistica Sinica*, pages 143–167, 1994.

[9] J. Y. Chen, W. Kollmann, and R. W. Dibble. Pdf modeling of turbulent nonpremixed methane jet flames. *Combustion Science and Technology*, pages 315–346, 1989.

[10] P. Ciaccia and M. Patella. Bulk loading the m-tree. In *Australasian Database Conference*, pages 15–26, 1998.

[11] A. Dobra and J. Gehrke. Secret: A scalable linear regression tree algorithm. In *ACM SIGKDD*, 2002.

[12] C. Faloutsos and I. Kamel. Beyond uniformity and independence: analysis of r-trees using the concept of fractal dimension. In *ACM PODS*, pages 4–13, 1994.

[13] J. H. Friedman. Multivariate adaptive regression splines. *Stanford University Tech Report-102*, 1988.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.

[15] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. Technical Report AIM-1440, 1993.

[16] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM*, pages 490–499, 1993.

[17] A. Karalic. Linear regression in regression tree leaves. In *ECAI*, 1992.

[18] C. A. Lang and A. K. Singh. Modeling high-dimensional index structures using sampling. In *ACM SIGMOD*, pages 389–400, 2001.

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

[20] A. Ning, J. Jin, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *ICDE*, pages 525–625, 2000.

[21] B.-U. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *ICDE*, pages 589–598, 2000.

[22] B. Panda, M. Riedewald, S. B. Pope, J. Gehrke, and L. P. Chew. Indexing for function approximation. In *VLDB*, 2006.

[23] S. B. Pope. Computationally efficient implementation of combustion chemistry using *in situ* adaptive tabulation. *Combustion Theory Modelling*, (1):41–63, 1997.

[24] S. Schaal, C. Atkeson, and S. Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *IEEE Int'l Conf. Robotics and Automation*, pages 288–293, 2000.

[25] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *ACM PODS*, pages 161–171, 1996.
[26] L. Torgo. Kernel regression trees. In *ECML*, 1997.
[27] I. Veljkovic, P. Plassmann, and D. C. Haworth. A scientific on-line database for efficient function approximation. In *ICCSA*, pages 643–653, 2003.
[28] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.
[29] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.
[30] Xxx. Xxx. In *Xxx*.