Using Errors to Create Piecewise Learnable Partitions

Oded Maron
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Sq., Room #755
Cambridge, MA 02139
oded @ ai.mit.edu

Keywords: decision trees, instance-based learning

Abstract

In this paper we describe an algorithm which exploits the error distribution generated by a learning algorithm in order to break up the domain which is being approximated into piecewise learnable partitions. Traditionally, the error distribution has been neglected in favor of a lump error measure such as RMS. By doing this, however, we lose a lot of important information. The error distribution tells us where the algorithm is doing badly, and if there exists a "ridge" of errors, also tells us how to partition the space so that one part of the space will not interfere with the learning of another. The algorithm builds a variable arity k-d tree whose leaves contain the partitions. Using this tree, new points can be predicted using the correct partition by traversing the tree. We instantiate this algorithm using memory based learners and cross-validation.

1 Motivation

Traditionally, the use of error in learning algorithms has been limited to improving the learner (through gradient descent where the algorithm tries to minimize the error) or as a judgment of how well the final algorithm does (by taking the root-mean-square of the error of a test set or cross validation). However, that is throwing away a lot of potentially useful information. If we think of learning as trying to approximate a function, it is often useful to see where in the domain of the function the learner performs badly. By looking for clusters and patterns of the errors, we can try to improve our original learner. For example, let's take the classic example of trying to learn the absolute value function $f(x) = |x|$. Most learning systems (most notably neural nets) generate higher errors when queried on points around $x = 0$. Figure 1 shows this distribution. If we notice this "hill" in the distribution of errors, then we can infer that the learning box behaves badly there. The bad behavior depends

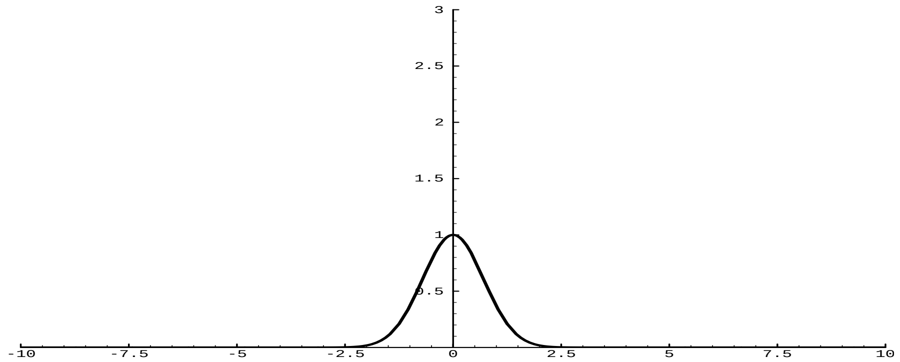

Figure 1: individual errors for absolute value

on the learning mechanism used — for neural nets, discontinuities tend to generate bad behavior; for memory based algorithms (such as variants of nearest-neighbor), sparseness generates bad behavior. Unless we can get more training points in the hilly area, we can't hope to learn that area any better. However, we can partition the space so that pieces of it (separated by hills) can be learned easily. In this case, we can learn the left and right arms of the absolute value function easily, and the hill in the error distribution tells us approximately where the break between the two arms occurs.

There are two underlying ideas that the rest of the paper is based on. One idea is that we are actually doing something which is much more general than just finding smooth or constant sections of a function. Not enough training points will produce non-smoothness-like effects, while with enough training points we can easily deal with discontinuities. By partitioning the input space into sections which are *learnable* (not necessarily smooth) we are able to cope with whatever the underlying learning mechanism is not able to cope with — thus we create *piece-wise learnable partitions*. In fact, the partitioning algorithm only looks at the errors produced by the learning algorithm. The actual predictions are not used.

The second idea is that learning systems are comprised of two equally important parts - the architecture and the training data. Once a particular architecture is set (e.g. 1-nearest-neighbor or a neural net with 50 hidden units using backprop), then the system is defined by the set of points used to train the system. This principle is fundamental to memory-based learning and is also used by the algorithm developed in this paper. By assigning a different learner to each partition, even if its architecture is the same, we have different learners. Given a prediction query, we find the partition that the query falls in, and then use the learner associated with that partition to answer the question.

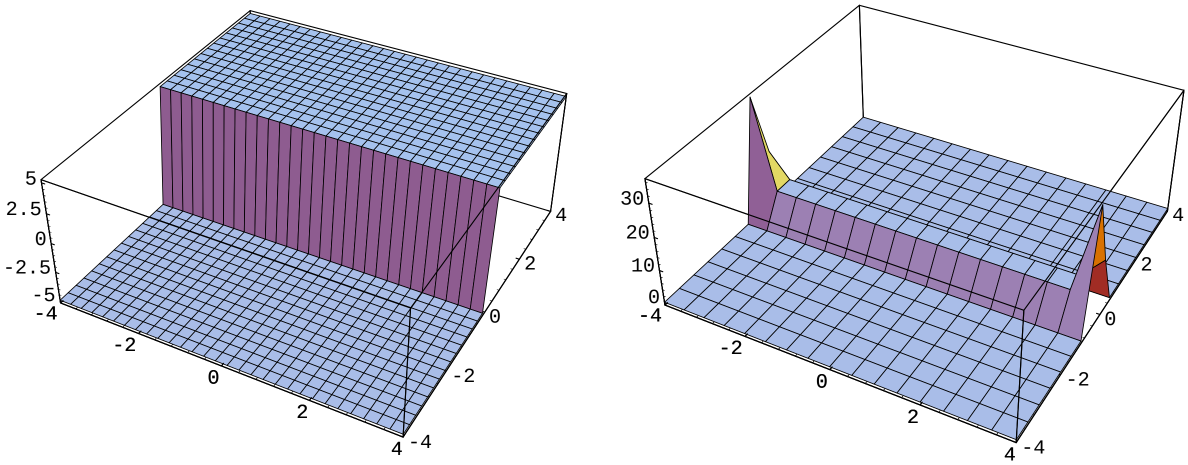

Figure 2: Function $a(x, y)$ and its error distribution

The paper is organized as follows: we first present the assumptions we made in order to make this algorithm computationally feasible. Then, we describe the algorithm and give an example of its execution. We proceed to discuss how we use the result of the algorithm (the partitioned space) to predict new points. Finally, we discuss related work.

2 Assumptions

In order to create a partitioning algorithm which is computationally feasible, we need to make a number of assumptions. I will try to explain how the assumptions affect the behavior of the algorithm and also why these assumptions are reasonable.

The main assumption which the Partitioning Algorithm (PA) makes is that the hills of errors which the learning algorithm generates occur in hyperplanes that are axis parallel. For example, let us try to learn the following two functions, plotted in Figures 2 and 3. They are both two-dimensional step functions. In this case, the error distribution are no longer 'hills' in two dimensions, but 'ridges' in three dimensions.

The error distribution graphs for a 4-nearest-neighbor algorithm on functions $a(x, y)$ and $b(x, y)$ are shown in Figures 2 and 3.

The PA would detect the errors in error graph a easily, but would have a difficult time deciding where to partition by looking at error graph b. This assumption is made for two reasons, the first one being computational tractability. By generalizing the problem to find any hyperplane of errors, we have a reduction to the problem of finding arbitrary edges in d-dimensional space, where d is the input dimension (or number of attributes) of the function we are trying to learn. While techniques from vision research exist for $d = 2, 3$, it appears that the problem is difficult for arbitrary dimensions. This assumption exists for most popular partitioning algorithms such as CART [Breiman *et al.*, 1984] or MARS [Friedman, 1988]. The approach described here was brought about by looking at functions which have some

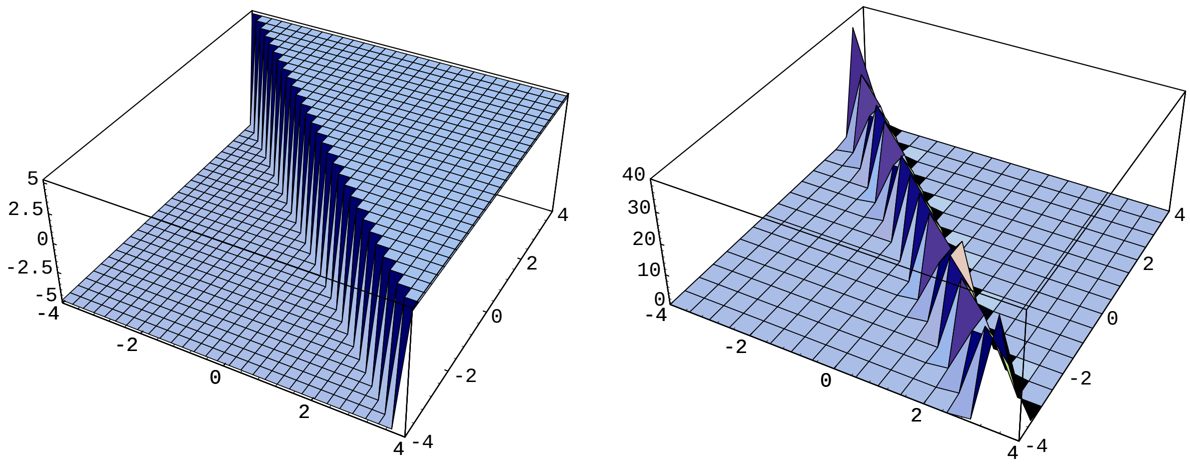

Figure 3: Function $b(x, y)$ and its error distribution

continuous attributes and some discrete attributes. In most of those functions, the output changes drastically when one of the discrete attributes changes value. In other words, there's a discontinuity along the axis where that discrete attribute flips from value x to value $x + 1$. That discontinuity is parallel to an axis and creates a hyper-plane which cuts through the entire input space and divides it into two parts.

The second assumption is concerned with the learning algorithm used to generate the errors. The PA treats the learning mechanism as a black box (which I will refer to as a "learning box"). The PA is not meant to have any knowledge of the behavior of the learning algorithm, except for the errors that it produces. I assume that by simply looking at the errors which certain inputs produce in *arbitrary* learning algorithms, we can find appropriate places to break the domain space. It seems unlikely that a proof of this claim is tractable unless some simplifying assumptions are made about the learning box (e.g. it uses polynomials of degree m to approximate the function).

Finally, the ideas here have been tested using the effective combination of memory-based learning boxes [Stanfill and Waltz, 1986] and leave-one-out cross validation [Wahba and Wold, 1975]. Memory-based methods are highly efficient with partitioning algorithms since once we have partitioned the training set, we do not need to retrain. The only work occurs during prediction, and training simply involves dumping all the points into memory - where they already are. Cross validation is a method for measuring the error of an approximator without dividing the original training set into a training and a test set. This is done by training on all the points but one, then finding the prediction error for that point. This is repeated for all points and averaged to produce the leave-one-out cross validation error. While this is an expensive measure of error with learning systems involving neural nets (since the whole net needs to be retrained for each leave-one-out), it is highly efficient with memory based systems. That is because the point can simply be "covered up" in memory, thereby producing a training set of all but that one point. Therefore, cross validation of a memory based system is just as cheap as using a test set. In addition, in tasks where there is not enough data for both training and testing, cross validation gives us a cheap way of estimating the error function over the entire training domain.

3 VAKID trees and the Partitioning Algorithm

In this section I will develop a more formal algorithm using the intuitions presented in earlier sections. The underlying ideas remain the same as in the absolute-value example — we want to look for hills in the error function and partition the space into regions which lie in between the hills. When the function we are trying to approximate is multi-dimensional, we are no longer looking for hills, but for ridges of mountains as in Figure 2. However, because of the assumption that these ridges are axis-parallel, we can look for the ridges one dimension at a time. This is exactly what the algorithms does: for each dimension, we scan along that dimension looking for hills, and partition along those that we find.

The partition itself is represented as a hyper-rectangle and also contains the points which fall in that partition. After we're done partitioning, we would like to predict a new point by using the points in the partition which the new point falls in. In order to find that partition efficiently, we construct a variable arity k-d tree (VAKID tree). Each node in the VAKID tree contains a partition. A node's children are subpartitions of that node. Therefore, the root node contains all of the training data and its hyper-rectangle represents the entire domain space. The leaf nodes contain the final partitions of the training data and the union of their hyper-rectangles results in the entire domain space. At the end of the algorithm, each of the leaf nodes contains a subsection of the domain space which is easily learnable. The tree is composed of $d+1$ levels, where d is the number of dimensions of the input space. The root is at level 0. At level i are all of the nodes which represent partitions along the i^{th} dimension.

The algorithm has d iterations. On the i^{th} iteration the algorithm attempts to partition every node at level $i-1$. If node q produces k partitioning points (there are k hills), then the algorithm creates $k+1$ children to node q and assigns each child their respective training subset and hyper-rectangle.

All that we need now is a procedure to find hills given some data and given a dimension i to look across. *Conceptually*, what we want to do is scan along the i^{th} dimension and detect hills which are parallel to our scan line. Once we find one of these hills, we consider all the points before it to be in one partition, and continue scanning the remaining points. The obvious question is: how do we recognize a hill of errors when we see one? In other words, what are the necessary dimensions (*height* and *width*) of a hill? The hill must be high enough so that noisy data (which creates "noisy" errors) will not cause over-partitioning. In addition, the hill must be wide enough so that a few outliers will not be cause over-segmentation of the training set. Finding values for the *height* and *width* parameters will be discussed later.

Practically, we perform the scan along the i^{th} dimension by sorting the points according to the value of the i^{th} attribute. This reduces the d-dimensional error graph to a one-dimensional plot just like the one in Figure 1. Finding hills of a certain *height* and *width* in a one-dimensional graph is now straightforward. The following example will make this trick more intuitive.

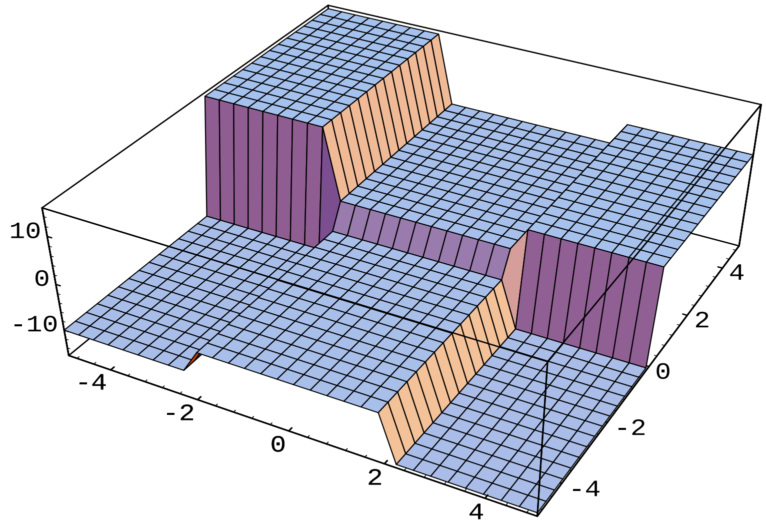

Figure 4: function $SixSteps$

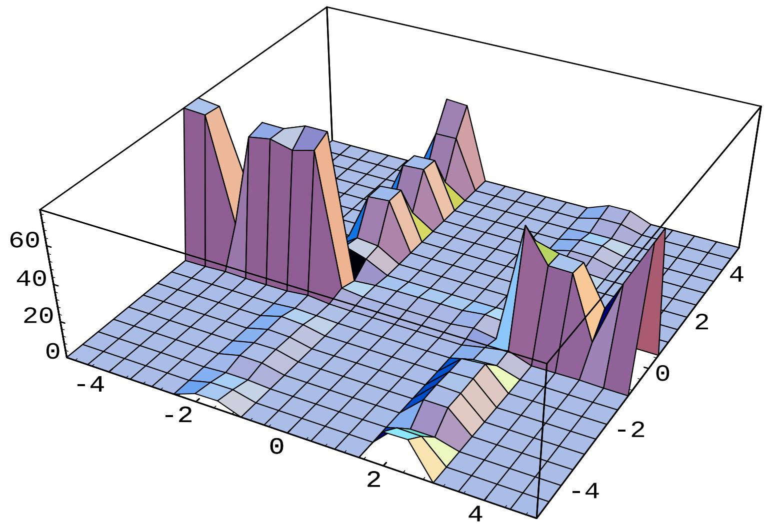

Figure 5: error function of $SixSteps$

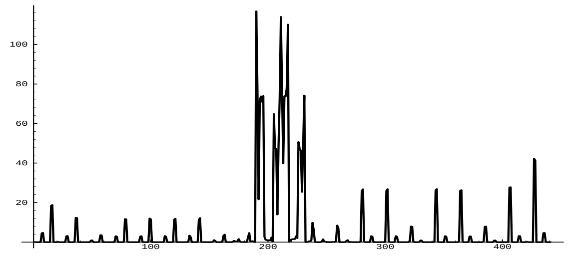

Figure 6: errors along the x dimension

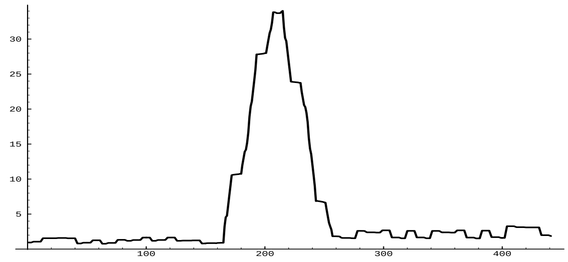

Figure 7: smoothed errors along the x dimension

3.1 Example

Following is an extended example of the execution of this algorithm. Let's say that the function which we are trying to learn is composed of six piece-wise constant sections, called $SixSteps(x, y)$, which is shown in Figure 4. The sections are constant for the sake of simplicity; they can be of any form which the learning box learns easily. In fact, if each region has a constant value, we are actually trying to perform classification rather than regression. A good learning algorithm for this function turns to be one which uses local weighted regression over about 10% of the points. In other words, in order to predict a new point after having m points in memory, we do a weighted average over $m/10$ of the nearest points to it. Using cross validation, we generate the error graph shown in Figure 5. The error graph consists of points of the form $\{x, y, Error_{x,y}\}$.

We arbitrarily choose to begin partitioning along the x dimension. Imagine taking horizontal strips of Figure 5 and putting them end to end — that is the equivalent of sorting according to the x values. This gives us the desired one-dimensional plot of errors shown in Figure 6. The central, wide spike is the central hill of errors which stretches along $y = 0$. The smaller, narrower spikes are the perpendicular hills. We smooth (described in the refinements below) this graph to get Figure 7. The partitioning is now clear — we split the set of points into two parts, those to the left of the center of the hill, and those to the right. The $VAKID$ tree shown in Figure 8 represents the state of the data structure at this point of the algorithm. We repeat the process for each of the two children, except that this time we sort along the y dimension. The two (one for the left child, one for the right child) smoothed graphs are shown in Figures 9 and 10. In each of these plots, there are two hills (creating three partitions). The final $VAKID$ tree is shown in Figure 11. Its six leaves correspond to the six piecewise learnable partitions of the space we're trying to learn.

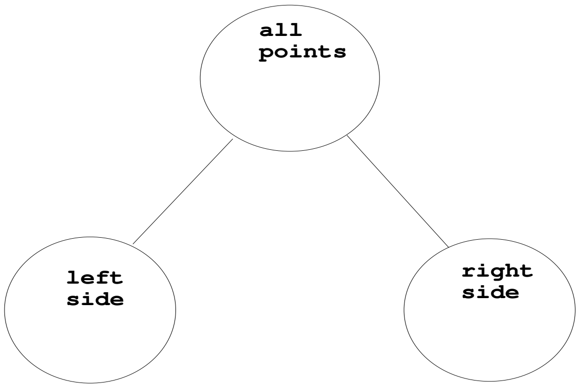

Figure 8: initial VAKID tree

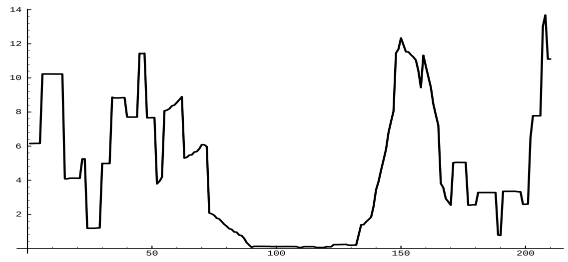

Figure 9: smoothed errors along the y dimension, left node

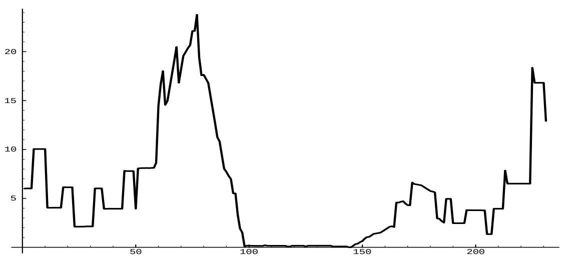

Figure 10: smoothed errors along the y dimension, right node

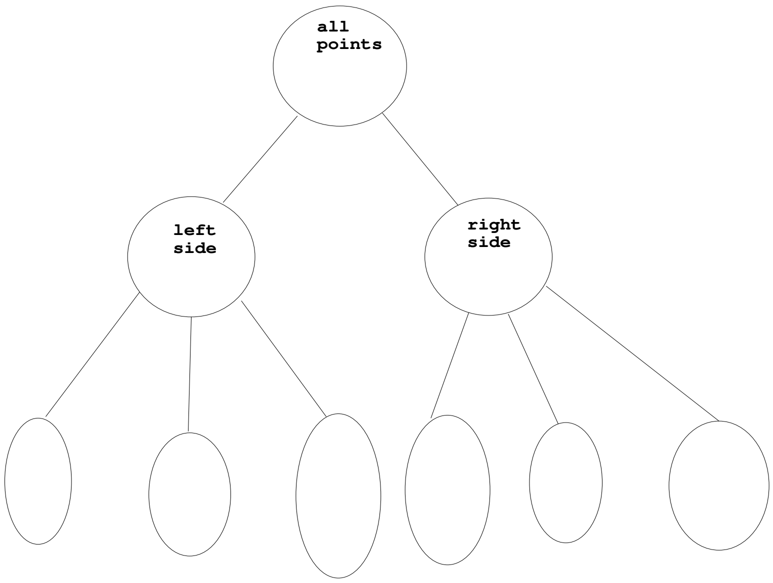

Figure 11: final VAKID tree

3.2 Refinements

There are two refinements to this algorithm which in practice greatly improve performance. The first one is a smoothing refinement. In practice, due to non-uniform point distribution, most hills are in the form of a series of spikes (as in Figure 6). In order for the algorithm to behave correctly (i.e. identify the spikes as a single hill), we smooth the errors so that each error is an average of its $width$ neighbors.

So far, we have maintained an abstraction barrier of treating the learning mechanism as a black box and not looking at anything but the errors that it produces. We'll break this barrier for the second refinement and look at the actual values predicted by the learning box. Splitting the hill at its midpoint is fine if the attribute on which we are splitting is continuous. However, if that attribute is discrete, we want to split so that all points on one side of the split have one value on that attribute and all points on the other side of the split have a different value on that attribute. The intuition behind this is twofold: first, it is probably the flip of that particular attribute from having value a to having value b which caused the hill of errors. The only reason for the hill not being symmetric about the flip is that there were more training points with one value of the discrete attribute than there were with the other. The other reason for the non-symmetric split is that we should create children which have non-overlapping hyper-rectangles. This is so that there will be no ambiguity when we use the VAKID tree for prediction.

4 Using the VAKID tree for prediction

By partitioning the training set, we have effectively partitioned the domain which we are trying to learn. In fact, we have partitioned it into sections which should be learnable independently of the other sections. However, there is no reason to think that the original learning box (the one who's errors we used to find the partitions) is the best learner for each one of the partitions. Traditionally, learning algorithms need to be tweaked and different constants can be used for different pieces of the domain. This paper does not deal with finding a good learner for each partition — we simply assume that each node in the tree also has a learner associated with it. If the learners are memory based, then there is no need for retraining; the memory for a learner at node q is simply the set of points in node q. If the learners are neural nets, then a net at node q will be trained only on points in node q.

Now that we have a trained learner for each part of the space, we can start predicting new points using the appropriate learner. Given a new point, we traverse the tree according to the values of the point's attributes until we hit a leaf or until none of the children's hyper-rectangles cover the new point. This can happen if the new query falls right in the cusp between two partitions. For example, in the absolute value example, the partitioning will probably not happen right at $x = 0$. One of the partitions might extend from $x = -0.3$ downward and the other one from $x = 0.01$ upward (depending on the training set). If we try to predict the value of the point at $x = 0$, we can't use either partition. Therefore, we predict from the node at which we stopped traversing, which in this case is the root node. Notice that the PA hasn't resolved the problem of predicting points at discontinuities; this can only be remedied with more training points. However, we have made sure that points on either side of a discontinuity do not effect each other. In general, once the tree traversal has stopped, we use the current partition's learner to answer the query.

If the query point falls completely outside the area of the original training set then the traversal stops at the root node and the entire set of points is used. This might indicate that this technique would be bad for time-series predictions. However, that's not completely true. The algorithm can be used to see if the same model (learning box) is applicable to various time periods. Many time series have different models at different time periods, and the PA can be used to determine when those time periods start and end.

5 Discussion

5.1 Cost of partitioning

Clearly, partitioning costs more than learning the entire space. Luckily, the expense does not grow exponentially. In terms of space, we keep the entire set of points at each level of the tree, and since the tree has d levels, we have a d-fold increase in space. As for computational cost, the main expense comes from sorting the points at each level of the tree, even if no partitioning ever occurs. Therefore, creating the VAKID tree costs $O(d \cdot n \log n)$ if the original training set has n points. If we bound the number of hills found along each dimension to be B, then each testing query carries an overhead of $O(d \cdot B)$.

The cost of finding a new learner for each partition depends on the particular instantiation of this algorithm and therefore will be not be discussed here. It can be as cheap as just using the original learning box at each node, or as expensive as doing a full search for the best fit for each partition. For some examples of memory based model selection see [Maron, 1994] or [Moore *et al.*, 1992].

5.2 Finding *Height* and *Length*

Unfortunately, it doesn't seem that we can rigorously determine an appropriate hill size for arbitrary learning algorithms with arbitrary training distributions. We need to find large and continuous deviations from the average error, but we'll need heuristics in order to determine just how large and how wide the hill need to be. One obvious heuristic for the *height* parameter is that it should be higher than the average error.

The *width* parameter describes the lowest length of the 'ridge'. If it is set too high, then no hill be wide enough and no partitioning will occur. If it is set too low, then the algorithm will find too many hills and partition too often. This leads to a fast deterioration in prediction accuracy since there aren't enough points in each partition in order to train the associated learner. If we assume that the axis-parallel hills cut through the entire training space, then hills whose ridge cuts across only half of the domain are not wide enough. Therefore, we use the heuristic that hills must be at least $\sqrt{n}$ wide, where n is the number of training points in the region currently being partitioned.

6 Related Work and Conclusion

The notion that high prediction errors occur along boundaries between models or classifications is not new. In fact, one technique of lowering the space requirements for memory-based learning algorithms is to just keep the instances with high error, and therefore just remember where the boundaries are [Aha, 1990]. The idea of creating a tree of learners is similiar to [Jordan and Jacobs, 1992] where a tree of experts is constructed. The partitioning idea is an integral part of any decision tree algorithms (such as CART or MARS). Like this algorithm, CART can only create partitions which are axis-parallel.

However, there are two main differences between the algorithm presented here and 'decision tree'-like learning systems. One difference is that the decision to break up a set of points is determined by the *errors* of whatever learning mechanism we used to originally try to learn the points (as opposed to looking at the attribute values and the prediction values). CART attempts to break the space at every possible place and picks the one with the least resulting cross-validation error. We immediately choose to break at the place which alligns with a hill of errors. Neither one of these approaches guarantees that the choice is an optimal cut, despite the fact that they are both intuitive approaches (see [Schaffer, 1993] and [Wolpert, 1993] for criticisms of the CART approach). The second difference is that this algorithm breaks the space into learnable partitions, which is much more general than trying to break it into constant or linear sections [Quinlan, 1992].

[Jordan and Jacobs, 1992] The algorithm presented in this paper is meant to take a learning algorithm with some training data and see where it performs badly. If the errors are distributed in axis-parallel ridges, then the algorithm can partition the domain which we are trying to learn into pieces which fall between those ridges. This creates piece-wise learnable partitions which can be learned and queried independently of each other. The algorithm is particularly efficient when used with memory-based learners, and we can get a cheap estimation of the error distribution by using cross validation.

References

[Aha, 1990] D. W. Aha. A Study of Instance-Based Algorithms for Supervised Learning Tasks: Mathematical, Empirical and Psychological Evaluations. PhD. Thesis; Technical Report No. 90-42, University of California, Irvine, November 1990.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[Friedman, 1988] Jerome H. Friedman. Multivariate Adaptive Regression Splines. Technical Report No. 102, Stanford University, Department of Statistics, November 1988.

[Jordan and Jacobs, 1992] Michael I. Jordan and Robert A. Jacobs. Hierarchies of adaptive experts. In J. Moody, S. Hanson, and R. Lippman, editors, *Neural Information Systems 4*. Morgan Kaufmann, 1992.

[Maron, 1994] Oded Maron. Hoeffding Races: Accelerating model selection search for classification and function approximation. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, April 1994.

[Moore *et al.*, 1992] A. W. Moore, D. J. Hill, and M. P. Johnson. An empirical investigation of brute force to choose features, smoothers and function approximators. In S. Hanson, S. Judd, and T. Petsche, editors, *Computational Learning Theory and Natural Learning Systems, Volume 3*. MIT Press, 1992.

[Quinlan, 1992] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1992.

[Schaffer, 1993] Cullen Schaffer. Overfitting avoidance as bias. *Machine Learning*, 10:153–178, 1993.

[Stanfill and Waltz, 1986] C. Stanfill and D. Waltz. Towards memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, December 1986.

[Wahba and Wold, 1975] G. Wahba and S. Wold. A completely automatic french curve: Fitting spline functions by cross-validation. *Communications in Statistics*, 4(1), 1975.

[Wolpert, 1993] David H. Wolpert. On overfitting avoidance as bias. Technical Report No. 92-03-5001, The Santa Fe Institute, 1993.