

Algorithms for Manipulation of Level Sets of Nonparametric Density Estimates ¹

Jussi Klemelä

Department of Statistics, University of Mannheim, L 7 3-5,
Verfügungsbäude, 68131 Mannheim, Germany

Summary

We present algorithms for finding the level set tree of a multivariate density estimate. That is, we find the separated components of level sets of the estimate for a series of levels, gather information on the separated components, such as volume and barycenter, and present the information together with the tree structure of the separated components. The algorithm proceeds by first building a binary tree which partitions the support of the density estimate, followed by bottom-up travels of this tree during which we join those parts of the level sets which touch each other. As a byproduct we present an algorithm for evaluating a kernel estimate on a large multidimensional grid. Since we find the barycenters of the separated components of the level sets also for high levels, our method finds the locations of local extremes of the estimate.

Keywords: CART, Fast computation, Kernel estimation, Level set tree, Multivariate density estimation, Visualization

¹Writing of this article was financed by Deutsche Forschungsgemeinschaft under project MA1026/8-1.

1 Introduction

We are interested in making inference on the shape of a multivariate density function by studying level sets

$$\Lambda_\lambda = \{x \in \mathbf{R}^d : \hat{f}(x) \geq \lambda\}$$

for a number of different levels $\lambda \in \mathbf{R}$, where $\hat{f} : \mathbf{R}^d \rightarrow \mathbf{R}$ is a density estimate based on a sample $X^1, \dots, X^n \in \mathbf{R}^d$. Shape characteristics of a density function include the number and location of modes, relative largeness of the modes, skewness, kurtosis, and the tail behavior of the function. Finding information on the shape characteristics of a multivariate function is a complex task, and the mere ability to evaluate the function does not imply that we are in a position to find the shape of the function. The level set based approach which we consider is a promising approach for the problem of visualization of multivariate functions.

Level set tree plots as an exploratory tool in multivariate density estimation were introduced in ?. Level set tree plots include *volume plot* and *barycenter plot*. Volume plot visualizes the number and relative largeness of the modes of the density, and gives information on the kurtosis. In Section 3 we show examples of volume plots (see Figure 4). Barycenter plot draws the “skeleton” of the function, visualizing locations of the barycenters of the level sets, in particular the locations of the modes, and giving information on the skewness.

Level sets has been applied in density estimation and mode detection in 3 and 4 dimensional cases for example by Scott (1992) and Härdle and Scott (1992), who present a sliding technique for visualizing 4 dimensional functions. They visualize 3D density contours as the fourth variable is changed over its range. Our aim is to apply level sets in arbitrary dimension with the help of level set tree plots.

Calculating the level set tree plots requires performing the following tasks: for some grid of levels,

1. partition each level set to pairwise separated components so that each component is connected, and form the *level set tree*, and
2. calculate volume and barycenter for each separated component of each level set.

(The concepts of “separated” sets and a “connected” set are defined in page 7.) *Level set tree* of a piecewise constant function is a tree structure formed by taking as root nodes the pairwise separated and connected regions of the support of the function. The child nodes of a given node are those pairwise separated and connected regions of the level set corresponding to the one step higher level than the level of this parent node, which are subsets of the

parent node. Level set trees for continuous functions are constructed by first discretizing the function.

Our approach to the calculation of the level set tree is the following.

1. We represent the function (density estimate) with the help of an *evaluation tree*. Evaluation tree is a binary search tree which defines a partition of the support of the function. The function is approximated with a function which is constant over the sets of the partition.
2. We find the separated and connected components of level sets with bottom-up travels of the evaluation tree. Those terminal nodes of the evaluation tree which are close to each other in the tree structure correspond to sets of the partition which are spatially close. During the travel of the tree we join always those sets which are touching (are not separated).

We consider three types of estimates in this article: (1) kernel estimates, (2) CART histograms, and (3) aggregated estimates. CART (classification and regression tree) histograms were defined in Breiman, Friedman, Olshen and Stone (1984). Aggregated estimates are enhancements of simple estimates with bagging (bootstrap aggregation) and boosting. These methods increase granularity of simple estimates, for example CART-histograms. Average shifted histograms (ASH), as introduced by Scott (1985), fall also in the category (3) of aggregated estimators. We will not study new density estimation methods, but want to present algorithms for the manipulation of level sets of some common classes of estimators.

Even when we are not interested in the calculation of the level set tree, the method of evaluation trees provides a method for evaluating kernel estimates on a large multidimensional grid. The method of evaluation trees may also be combined with binning, since we may store the information on the bins in evaluation trees.

Although we specialize to the case of density estimates, the algorithms of this article apply to the manipulation of level sets of quite general multivariate functions. In particular, manipulation of level sets of estimates of regression functions is of interest.

Implementations of the algorithms of this article may be found from R-package “denpro” which can be downloaded from <http://www.denstruct.net>.

In Section 2.1 we define the evaluation tree, Section 2.2 gives an algorithm for finding the level set tree, Section 2.2.1 gives an algorithm for decomposing a single level set to separated and connected components, Section 2.2.2 gives some detailed pseudo codes for the algorithm of Section 2.2.1. Section 3 gives details for the case of calculating level set trees for kernel estimates, CART histograms, and aggregated estimates.

2 Finding the level set tree with the help of an evaluation tree

We define first the evaluation tree in Section 2.1. The algorithm for finding the level set tree which we present in Section 2.2 presupposes that the function is represented with an evaluation tree.

2.1 Evaluation tree of a function

We create a binary search tree to store a setwise constant function (or a setwise constant approximation of a function). We call this tree the *evaluation tree*.

Definition 1 An evaluation tree is any tree satisfying the following properties.

1. The tree has a single root node, and every node has 0, 1, or 2 children.
2. Non-leaf nodes are annotated with a splitting direction $k \in \{1, \dots, d\}$, splitting point $s \in \mathbf{R}$, a pointer to the left child, if it exists, and a pointer to the right child, if it exists.
3. Every node of the tree is annotated with a rectangle $R \subset \mathbf{R}^d$ whose sides are parallel to the coordinate axis. Given a non-leaf node which is annotated with rectangle $R \subset \mathbf{R}^d$, splitting direction $k \in \{1, \dots, d\}$, and splitting point $s \in \mathbf{R}$, then denote $R_1 = \{x \in R : x_k \leq s\}$ and $R_2 = \{x \in R : x_k > s\}$. We assume that splitting point s is such that these sets have positive volume. If the node has left child, then this child is annotated with R_1 and if the node has right child, then this child is annotated with R_2 .
4. Leaf nodes are annotated with real values. These values represent the values of the function.

The function associated with the evaluation tree. An evaluation tree represents a rectangularwise constant function. Evaluation tree T represents the function

$$\bar{f}(x) = \sum_{A \in \mathcal{A}} f_A I_A(x) \quad (1)$$

where \mathcal{A} is the collection of the annotations of the leaf nodes of T with rectangles:

$$\mathcal{A} = \{A \subset \mathbf{R}^d : A \text{ is a rectangle annotated with some leaf node of } T\} \quad (2)$$

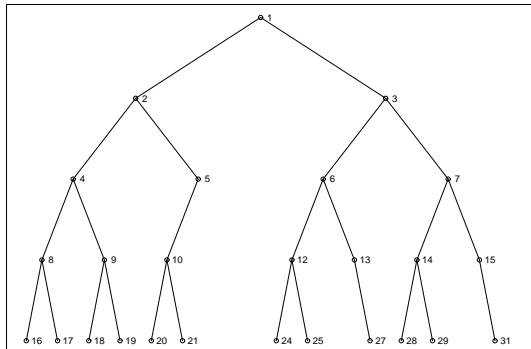


Figure 1: A partition generating tree. Figure 2 shows the annotations of the nodes with rectangles.

and $f_A \in \mathbf{R}$ are the annotations of the leaf nodes of T with real values. We denote $I_A(x) = 1$ when $x \in A$ and $I_A(x) = 0$ otherwise.

Partition generating tree. The conditions 1-3 of Definition 1 define a partition generating tree. The corresponding partition is given in (2). By adding condition 4 to Definition 1 we make this partition generating tree an evaluation tree.

Example. Figure 1 shows a partition generating tree. Figure 2 shows the rectangles annotated with the nodes of this tree and the process of growing this tree. We start with the rectangle $[0, 4] \times [0, 4]$. We split first parallel to y -axis making dyadic splits, and then parallel to x -axis.

Evaluation trees in density estimation. In density estimation piecewise constant functions may appear at least in two ways: (1) we construct first a continuous density estimate, like a kernel density estimate, and then approximate this continuous function with a piecewise constant function, or (2) we construct directly a piecewise constant estimate, like in the case of CART-histograms and aggregated estimates.

By the definition of the evaluation tree, the root node of this tree is annotated with a rectangle R_0 and the evaluation tree generates a partition of R_0 . When a density estimate vanishes in some regions of R_0 , we do not have to include rectangles corresponding to those regions in the evaluation tree. For example,

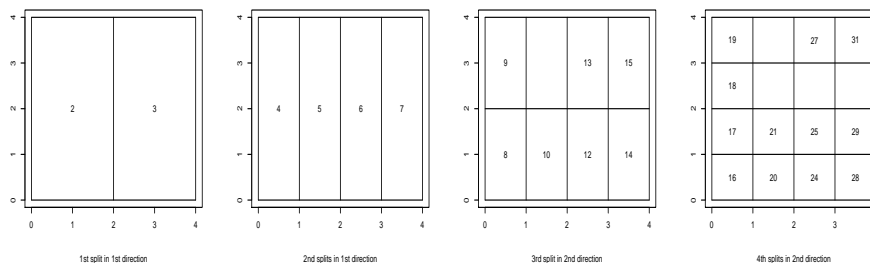


Figure 2: The rectangles annotated with the nodes of the tree shown in Figure 1, and the process of growing this tree.

the evaluation tree of Figure 1 does not have leaf nodes corresponding to all regions of $[0, 4] \times [0, 4]$.

Advantages of the evaluation trees. There are two advantages we get from representing piecewise constant functions with binary trees.

1. We are able to find fast the value $\bar{f}(x)$ at any $x \in \mathbf{R}^d$, using the binary search algorithm. Thus we are also able to refine fast the values of the stored function. Refining may mean changing the value of the estimate at some region. In the case of the function approximation refining may mean splitting further some regions where the approximation was constant, thus giving a more accurate approximation of the function.
2. Siblings in the evaluation tree of the function correspond to spatially close sets. Thus we may apply a dynamic programming algorithm to find separated components of the level sets. We make a bottom-up travel of this tree, during which we join the unseparated parts of level sets, thus solving the problem of finding separated components by solving a "dual" problem of joining the unseparated components.

2.2 Finding the level set tree

We describe an algorithm for decomposing a single level set to separated and connected components in Section 2.2.1. Section 2.2.2 gives a detailed pseudo code for the algorithm. The algorithm for forming the level set tree which we describe in Section 2.2.3 is straightforward once we have an algorithm for decomposing a single level set.

2.2.1 Decomposing a single level set to mutually separated and connected components

The basic ingredient of our algorithm for creating the level set tree is to decompose a single level set to the maximally separated components. In fact, typically we want to decompose only a part of the whole level set. That is, we want to decompose a part of the level set corresponding to a branch of the level set tree. For simplicity we present the algorithm for the case when we decompose the whole level set.

Definition of separated sets and a connected set. We say that sets $B, C \subset \mathbf{R}^d$ are *separated* if $\inf\{\|x - y\| : x \in B, y \in C\} > 0$, where $\|\cdot\|$ is the Euclidean distance. Then we say that set $A \subset \mathbf{R}^d$ is *connected* if for every nonempty B, C such that $A = B \cup C$, B and C are not separated.

The problem statement. We want to make a decomposition of a level set so that the members of the decomposition are mutually separated and each is connected. Level sets Λ_λ of \bar{f} , defined in (1), have the form

$$\Lambda_\lambda = \cup_{A \in \mathcal{A}_\lambda} A \quad (3)$$

where

$$\mathcal{A}_\lambda = \{A \in \mathcal{A} : f_A \geq \lambda\} \quad (4)$$

and \mathcal{A} is defined in (2). Assume that function \bar{f} is the associated function of some evaluation tree. Then it follows that the sets A in representation (1) are rectangles. Rectangles are connected sets. Since sets $A \in \mathcal{A}_\lambda$ are connected, our task is to find the partition $\{\mathcal{A}_{\lambda,1}, \dots, \mathcal{A}_{\lambda,M}\}$ of \mathcal{A}_λ , so that the following conditions are met.

1. $\mathcal{A}_\lambda = \cup_{i=1}^M \mathcal{A}_{\lambda,i}$.
2. For $i, j = 1, \dots, M, i \neq j$, $\cup_{A \in \mathcal{A}_{\lambda,i}} A$ and $\cup_{A \in \mathcal{A}_{\lambda,j}} A$ are separated.
3. For $i = 1, \dots, M$, $\cup_{A \in \mathcal{A}_{\lambda,i}} A$ is connected.

Condition 1 says that we make a partition of set $\Lambda_\lambda = \cup_{A \in \mathcal{A}_\lambda} A$ to sets $\cup_{A \in \mathcal{A}_{\lambda,i}} A, i = 1, \dots, M$. Condition 2 says that the members of the partition are pairwise separated. Condition 3 says that the members of the partition are connected. Condition 3 implies that the partition is maximal in the sense that trying to split some member of the partition will lead to a violation of condition 2.

A naive algorithm. A straightforward algorithm would compare all sets A in the representation (3), with each other, to find which sets A touch each other. However, this algorithm needs $O((\#\mathcal{A}_\lambda)^2)$ steps, where $\#\mathcal{A}_\lambda$ is the cardinality of set \mathcal{A}_λ . When this cardinality is very large, this is not a feasible algorithm.

Algorithm. We give a pseudo code for the algorithm **DynaDecompose** which decomposes a level set to maximally separated components. This algorithm takes as an input the evaluation tree of the function. The leaf nodes of this evaluation tree correspond to the lowest level set of the function defined in (1). Thus we need a special labeling for the leaf nodes of the evaluation tree which indicates which sets belong to collection \mathcal{A}_λ defined in (4), which is the collection of sets whose union is the level set Λ_λ . We travel the evaluation tree starting from the leaf nodes of the tree.

1. **Input** of the algorithm is an evaluation tree of function defined in (1) and a list \mathcal{L}_λ which labels a subset of the terminal nodes of the evaluation tree. (List \mathcal{L}_λ corresponds to set \mathcal{A}_λ .)
2. **Output** of the algorithm is a partition of \mathcal{L}_λ , that is, a partition of \mathcal{A}_λ .

ALGORITHM **DynaDecompose**

1. **travel** the evaluation tree, starting from the terminal nodes labeled by the list \mathcal{L}_λ ; traveling has to proceed so that the children are encountered before the parent;
 - (a) assume that we have encountered node m ;
 - (b) **if** node m is a leaf node annotated with set A , **then** we annotate node m with the partition $\{\{A\}\}$;
 - (c) **else** (m is not a leaf) we check whether there are connections with the sets in the partitions which are annotated with left and right child, and annotate node m with the partition joining the 2 partitions;
2. **end travel**
3. **return** the partition which is annotated with the root node

Section 2.2.2 gives a more precise pseudo code for the algorithm. Especially step 1(c) is rather complex, requiring to make a pairwise comparison of all “current components”. In the general case, when we decompose a subset of level set Λ_λ , then we give as input a list $\mathcal{L}_{\lambda,0} \subset \mathcal{L}_\lambda$.

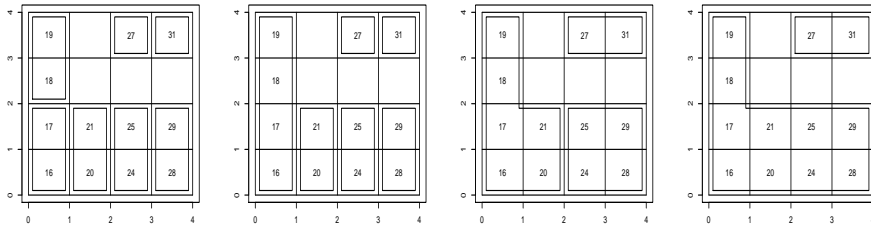


Figure 3: The process of finding the two separated regions of a level set of an estimate. Figure 1 shows the corresponding evaluation tree.

Example. Figure 3 illustrates the evolution of the algorithm. We assume that the evaluation tree of the function is given in Figure 1. We assume that the function is positive on all the rectangles corresponding to the leafs of the evaluation tree and we want to decompose the lowest level set to the separated components. The first window shows how we have formed 1×2 rectangles. Then we join subsets inside the 1×4 rectangles to get window 2. Then we join subsets inside the 2×4 rectangles to get window 3. Then we join subsets inside the 4×4 rectangle to get the 4th window, which shows that the support of the function consists of two separated and connected components.

2.2.2 Detailed pseudo code for the algorithm DynaDecompose

We give a more precise pseudo code for the algorithm **DynaDecompose** than the pseudo code given in Section 2.2.1.

The algorithm proceeds by joining those sets in the partitions annotated with the two children which are touching each other. (Partitions are represented as lists of lists of terminal nodes of the evaluation tree.) To find which sets are touching each other we have to travel through the boundaries of the sets. We have to consider 4 cases, corresponding whether left and right children exist, and whether the boundaries of the left and right children are empty or not.

1. **Input** and **Output** of the algorithm are given in page 8.
2. **Internal data structures** of the algorithm are the following. We associate to every node m of the evaluation tree 3 lists of lists of terminal nodes of the evaluation tree: a list of lists $DC(m)$ of nodes in the separated components, a list of lists $LB(m)$ of nodes in the left boundaries

of the separated components, and a list of lists $RB(m)$ of nodes in the right boundaries of the separated components.

ALGORITHM DynaDecompose

1. **travel** the evaluation tree, starting from the terminal nodes labeled by the list \mathcal{L}_λ ; traveling has to proceed so that the children are encountered before the parent;
2. assume we have encountered node m ;
3. **if** node m is a leaf node **then** we annotate node m with the partition consisting of the single rectangle which is annotated with this leaf node;
4. **else** (m is not a leaf node)
 - (a) **if** left child of m does not exist, **then**
 - i. the separated parts of m are inherited from the right child rm (which exists): $DC(m) = DC(rm)$;
 - ii. the left boundary of m is empty: $LB(m)$ is empty list;
 - iii. the right boundary of m is the right boundary of the right child: $RB(m) = RB(rm)$;
 - (b) **else, if** the right child does not exist, **then**
 - i. the separated parts of m are inherited from the left child lm (which exists): $DC(m) = DC(lm)$;
 - ii. the left boundary of m is the left boundary of the left child: $LB(m) = LB(lm)$;
 - iii. the right boundary of m is empty: $RB(m)$ is empty list;
 - (c) **else, if** the left boundary of the right child is empty **or** the right boundary of the left child is empty (so that there exists no connections between the sets of the left child and the sets of the right child), **then**
 - i. the separated parts of m are inherited from the children, no parts are joined: $DC(m)$ is the concatenation of $DC(lm)$ and $DC(rm)$;
 - ii. the left boundary of m is the left boundary of the left child (possibly empty): $LB(m) = LB(lm)$;
 - iii. the right boundary of m is the right boundary of the right child (possibly empty): $RB(m) = RB(rm)$;
 - (d) **else**, (both children exist and both boundaries are non-empty);

- i. the separated parts are the result of joining the separated components of the left child and the right child: $DC(m)$ is formed by pairwise comparison of boundaries of components in $DC(lm)$ and $DC(rm)$ to find which components touch each other, and then concatenating the components which touch each other;
 - ii. the left boundaries of m are the left boundaries of the left child (possibly some are joined): take $LB(m) = LB(lm)$ and then concatenate those lists for which the corresponding lists in $DC(lm)$ were concatenated in step 4(d)i to get $DC(m)$;
 - iii. the right boundaries of m are the right boundaries of the right child (possibly some are joined): take $RB(m) = RB(rm)$ and then concatenate those lists for which the corresponding lists in $DC(rm)$ were concatenated in step 4(d)i to get $DC(m)$;
- (e) **end if**
5. **end if**
6. **end travel**
7. **return** $DC(m)$ where m is the root node of the evaluation tree

The step 4(d)i is the most complex part of the algorithm. At this step we make pairwise comparisons of sets to find which sets touch each other. Note that in step 4(d)i we do not need always run through the complete boundaries, since we may stop immediately if we find a connection. Steps 4(a)-4(c) of the algorithm are only bookkeeping.

For completeness, we give the algorithm **PairwiseComparison** which is used in step 4(d)i to find which components of the left and the right child touch each other. Note that this is the “naive algorithm” mentioned in page 8.

1. **Input** is a collection \mathcal{A} of sets. In our application \mathcal{A} is the collection of the intersections of the disconnected components, of the left and the right child, with respective boundaries. Furthermore, these intersections are unions of rectangles.
2. **Output** is a partition \mathbb{P} of \mathcal{A} . This partition is initialized to be empty. The partition consists of separated components and each is connected.
3. **An internal data structure** is stack S of sets, which is initialized to be empty.

ALGORITHM PairwiseComparison

1. **first loop:** go through the collection of sets \mathcal{A} , consider $A \in \mathcal{A}$;
2. **if** set A is not already a member of some set in \mathbb{P} ;
 - (a) start creating a new component \mathcal{C} , by initializing component \mathcal{C} to be empty;
 - (b) put A to the stack S (stack S will contain sets whose union is a connected component whose one member is A);
 - (c) **while** stack S is not empty;
 - i. take from stack S set B ;
 - ii. include B to the current component \mathcal{C} ;
 - iii. **second loop:** go through sets in \mathcal{A} , consider set C ;
 - A. **if** C touches set B and is not already in set \mathcal{C} , **then** put C to the stack S ;
 - iv. **goto second loop**
 - (d) **goto while**
 - (e) current component \mathcal{C} will be added to \mathbb{P}
3. **end if**
4. **goto first loop**
5. **return** \mathbb{P}

2.2.3 The algorithm for finding the level set tree

A level set tree is a tree structure formed by taking as root nodes the maximally separated regions of the support of the function. The child nodes of a given node consist of the maximally separated regions of the level set corresponding to the one step higher level than the level of this parent node.

The algorithm for forming the level set tree is now straightforward. We may build the level set tree by starting from the root nodes, traveling towards upper levels, and always decomposing parts of the level sets with the algorithm **DynaDecompose**.

3 Examples

The general algorithm of Section 2.2 take as an input an evaluation tree, defined in Section 2.1. We give examples of creating the evaluation tree. We discuss the cases of kernel estimates, CART, and aggregated estimates.

3.1 Kernel estimates

Kernel estimator, based on sample $X^1, \dots, X^n \in \mathbf{R}^d$, is defined as

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^n K((x - X^i)/h), \quad x \in \mathbf{R}^d, \quad (5)$$

where $h > 0$ is the smoothing parameter and $K : \mathbf{R}^d \rightarrow \mathbf{R}$ is the kernel function.

Discretization of the kernel estimate. We evaluate the kernel estimate on a grid which lies on a rectangle which contains the support of the estimate. Let G be the set of grid points where the estimate is positive. We consider the function

$$\bar{f}(x) = \sum_{g \in G} \hat{f}(g) I_{R(g)}(x), \quad x \in \mathbf{R}^d$$

where $R(g)$ is the rectangle whose center is g and whose sidelengths are equal to the steps of the grid, and \hat{f} is defined in (5).

Splitting strategies. An evaluation tree for kernel estimates may be constructed in several ways. Typically we start with the smallest rectangle containing the support of the kernel estimate. An evaluation tree may be formed by splitting one direction at a time, and after reaching the finest resolution of the grid in a given direction we move to the splitting of the next direction. Figure 2 shows the process of splitting in this way. Figure 1 shows the corresponding evaluation tree. An alternative way of creating an evaluation tree of a kernel estimate would be to alternate the direction of splitting. It is also important to consider non-regular partitions. For example, we may make approximation more accurate at some regions by growing the evaluation tree larger at those regions. Then the evaluation tree is not balanced like in Figure 2.

An algorithm for creating the evaluation tree. We evaluate the kernel estimate at the grid points by going through the observations, finding which gridpoints belong to the support of the kernel centered at the current observation, and updating the value of the estimate at those grid points. Since we create a binary search tree from the grid points where the estimate is positive, we are able to find fast, whether a given grid point already exists in the tree, and if it exists, update the value of the estimate at this grid point. We present the algorithm for the case where the grid points are regularly spaced in each direction, but the number of grid points may be different in different directions. We do not fix otherwise the splitting strategy.

1. **Input** of the algorithm are the observations $X^1, \dots, X^n \in \mathbf{R}^d$, kernel function $K : \mathbf{R}^d \rightarrow \mathbf{R}$, smoothing parameter $h > 0$, and vector $\delta = (\delta_1, \dots, \delta_d)$ of the steplengths of the grid. Algorithm supposes that kernel K has a compact support.
2. **Output** of the algorithm is an evaluation tree.

ALGORITHM **ETofKDE** (evaluation tree of a kernel density estimate)

1. Find the smallest rectangle containing the support of the kernel estimate (5), whose sides are parallel to the coordinate axis. The root node of the evaluation tree will be annotated with this rectangle. We will evaluate the kernel estimate on the regular grid lying on this rectangle, with stepsizes given by vector δ ;
2. **for** $i=1$ to n (go through observations X^1, \dots, X^n);
 - (a) denote with $y_1, \dots, y_m \in \mathbf{R}^d$ the grid points which lie on the support of function $x \mapsto K((x - X^i)/h)$, $x \in \mathbf{R}^d$;
 - (b) **for** $j=1$ to m (go through gridpoints y_1, \dots, y_m);
 - i. **if** the evaluation tree of the kernel estimate already contains gridpoint y_j , **then** add the value $(nh^d)^{-1}K((y_j - X^i)/h)$ to the current value at this gridpoint;
 - ii. **else** create the node for gridpoint y_j and store to this node the value $(nh^d)^{-1}K((y_j - X^i)/h)$
 - (c) **end for**
3. **end for**

In step 2(b)i we apply the binary search algorithm to find out whether the evaluation tree already contains the gridpoint y_j .

Product kernels and the number of grid points. Kernels of product form are computationally attractive. Let

$$K(x) = \prod_{i=1}^d L(x_i), \quad x = (x_1, \dots, x_d), \quad (6)$$

where $L : [-1, 1] \rightarrow [0, \infty)$. One may for example choose L to be a truncation of a Gaussian density function, or the Bartlett-Epanechnikov polynomial $L(t) = (3/4)(1 - t^2)_+$ where $(x)_+ = \max\{x, 0\}$. Denote with M the number of gridpoints where the kernel estimate is positive. When the distance between grid points in direction i is δ_i , $i = 1, \dots, d$, and when the kernel has the product form (6), then

$$M \leq n \prod_{i=1}^d (2h\delta_i^{-1}). \quad (7)$$

Note that typically $h \asymp n^{-1/(4+d)}$.

Number of nodes of the evaluation tree. We may give an upper bound for the number of nodes of the evaluation tree of the kernel estimate. Evaluation tree has depth

$$D = 1 + \sum_{i=1}^d \log_2 N_i, \quad (8)$$

where N_i is the number of grid points in direction i . For the storing of the evaluation tree we need vectors whose length is at most $D \cdot M$, where M is the number of gridpoints where the estimate is positive, bounded in (7).

Extensions of the algorithm.

1. **Grid of smoothing parameter values.** Typically we want to evaluate the kernel estimate for a grid of smoothing parameter values. The above algorithm may be modified for this case. The evaluation tree corresponding to the estimate with the largest smoothing parameter value contains as subtrees the evaluation trees of the other estimates. Thus we annotate the leaf nodes of this largest evaluation tree with vectors whose length is equal to the number of different smoothing parameters, and the elements of the vector give the values of the estimates for different smoothing parameter values. Some of the entries of the vector will be 0.
2. **Further growing.** We may want at a later stage continue growing the evaluation tree to give a better approximation of the kernel estimate. To facilitate this we need to add pointers to the observations in leaf nodes. Thus we store slightly more information than in the case of binning, where only the frequencies need to be stored. When we store pointers to the observations, we do not have to go again through all observations in order to continue the growing of the tree. Note that the evaluation tree may be grown only locally in a neighborhood of a given point.

Other methods. At least the following proposals have been made for the fast computation of kernel estimates.

1. **Binning.** Binning in the one dimensional case was discussed in Fan and Marron (1994). Calculation of multivariate kernel estimates with binning was considered in Wand (1994) and the accuracy of binned kernel estimators was studied in Hall and Wand (1996). Accuracy and complexity of binning was studied by Holmström (2000).
2. **An updating method.** Seifert, Brockmann, Engel and Gasser (1994) consider an updating method for polynomial kernel functions. They

expand a kernel estimate $\hat{f}(x)$ in sums of powers of X^i , and update these sums when moving to nearby values of x . To our knowledge this method has been implemented only in the one dimensional case.

We do not have to consider binning as an alternative to the method of evaluation trees, since the evaluation tree is also a useful data structure to be used to store the result of binning. In this case every leaf node is annotated with a bin and a weight of this bin.

Evaluation at predetermined points vs. evaluation at arbitrary points. Algorithm **ETofKDE** evaluates the kernel estimate at a predetermined grid. We have 2 possibilities for evaluating the kernel estimate at arbitrary points: (1) we may apply binning or (2) we may evaluate kernel estimate at the knots of a grid and then use interpolation at the other points. Besides piecewise constant interpolation we may apply quadratic or cubic interpolation.

Direct level set estimation. Since the manipulation of level sets of kernel estimates is computationally expensive, one has proposed kernel type methods for the direct estimation of level sets. These estimates are unions of balls centered at some subset of observations, see for example Devroye and Wise (1980), Korostelev and Tsybakov (1993), Walther (1997), Baïllo, Cuevas and Justel (2000), Baïllo, Cuesta-Albertos and Cuevas (2001). However, even when one may store these estimates of level sets efficiently, manipulating of the estimates is still complex. Also, these estimates involve additional smoothing parameters (in particular, the radius of the balls centered at the observations) and statistical theory for choosing these smoothing parameters is not yet available.

Example. Let $f_l : \mathbf{R}^2 \rightarrow \mathbf{R}$ be the equal mixture of 3 standard Gaussian densities, located on the vertices of a triangle with sidelengths D . Let

$$f(x) = f_l(x_1, x_2) \prod_{i=3}^d f_i(x_i), \quad x \in \mathbf{R}^d, \quad (9)$$

where f_i , $i = 3, \dots, d$, are univariate Gaussian densities with zero expectation and with variance $1 + D^2/6$ (which is also the marginal variance of coordinates 1 and 2). This example is a generalization of the projection pursuit example introduced in Friedman, Stuetzle and Schroeder (1984) and studied by Scott and Wand (1991). Density f has a 2-dimensional structure (3 modes), and other dimensions are “noise dimensions”.

Figure 4 a) shows an example of (the central region of) a volume plot of a kernel estimate. We have $d = 4$, $D = 4$, and sample size is 2000. Smoothing parameter is $h = 1.4$ and Bartlett-Epanechnikov product kernel is used.

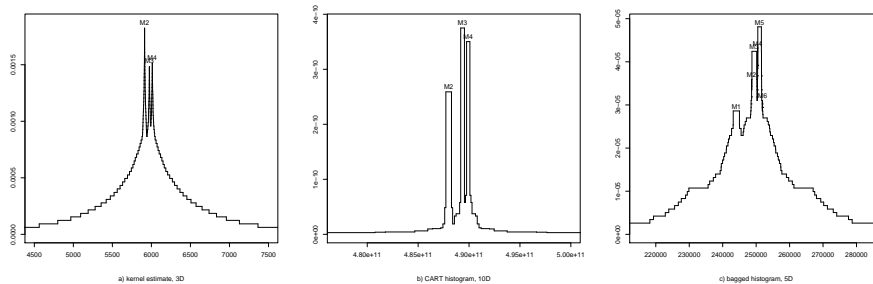


Figure 4: Three density estimates from 4D, 10D, and 5D projection pursuit data.

Estimate is discretized to 16^4 gridpoints and 60 levels. The calculation took about 40 seconds on a PC.

3.2 CART-histograms

CART (Classification and regression tree) is a famous procedure for creating adaptive histograms. It was introduced by Breiman et al. (1984). For a description of CART-type methods in density estimation and for further references, see Holmström, Hoti and Klemelä (2005).

The definition of CART is given in terms of an evaluation tree. In CART type methods the estimate is formed by a two step procedure. First a large evaluation tree is grown by myopic (greedy) splitting of the sample space minimizing an empirical error criterion. Secondly this evaluation tree is pruned by minimizing an error-complexity criterion.

Example. Figure 4 b) shows an example of (the central region of) a volume plot of a CART-histogram. We took a sample from the density defined in (9) with $d = 10$, $D = 6$, and sample size 1000. The estimate has 33 cells. The calculation took few seconds on a PC.

3.3 Aggregated estimates

ASH. Average shifted histograms were introduced in Scott (1985). Average shifted histograms are averages of regular histograms. As the number of averaged histograms increases, ASH approximates the kernel estimate whose kernel function is the product of triangular kernels $L(t) = (1 - |t|)_+$.

Bagging. Bootstrap aggregation was introduced by Breiman (1996*a*) and Breiman (1996*b*). In bagging we generate bootstrap samples from the original sample, produce unstable estimates (adaptive histograms) based on each bootstrap sample, and define the final estimate as the arithmetic mean of the estimates in the sequence. Bagging is a method to decrease variability of an unstable estimator. Bagging has been applied most successfully in classification, but see Holmström et al. (2005) for an application in density estimation.

Boosting. Boosting was introduced by Schapire (1990), Freund (1995), Freund and Schapire (1996). In the original boosting a weighted empirical risk is minimized, and the weights of the observations are adjusted in a step-wise manner, so that the weights of the observations at which the current estimate is inaccurate are increased. The final estimate is a weighted average of the estimates in the sequence. Boosting is a method to decrease the bias of the estimates.

An algorithm for creating an evaluation tree for aggregated estimates. We need an algorithm for creating an evaluation tree for a function which is a convex combination of rectangularwise constant functions. The problem may be reduced to the problem of creating an evaluation tree for a function which is a weighted average of two rectangularwise constant functions, when we have evaluation trees for these two functions. The general case will then be handled with the iteration of this procedure. The straightforward method of adding two evaluation trees is to go through the leaf nodes of the first tree, and to make a partition for each rectangle annotated with a leaf node by making the intersection between this rectangle and the partition generated by the second evaluation tree. Thus, the algorithm amounts to making the overlaying of two partitions. Figure 5 illustrates overlaying of two partitions.

Example. Figure 4 c) shows an example of (the central region of) a volume plot of a bagged histogram. We took a sample from the density defined in (9) with $d = 5$, $D = 5$, and sample size 1500. The estimate is the average of 5 CART histograms, which were pruned to have 15 cells. We applied $n/2$ -out-of- n without replacement bootstrap. The calculation took about 10 seconds on a PC.

Let us summarize Figure 4. All the estimates detected the 3 modes. The kernel estimator is statistically and computationally inefficient for high dimensional data, but works for moderate dimensional data. CART histograms are computationally efficient for high dimensional data, and we may increase statistical efficiency of these estimates with aggregation, but with a cost of

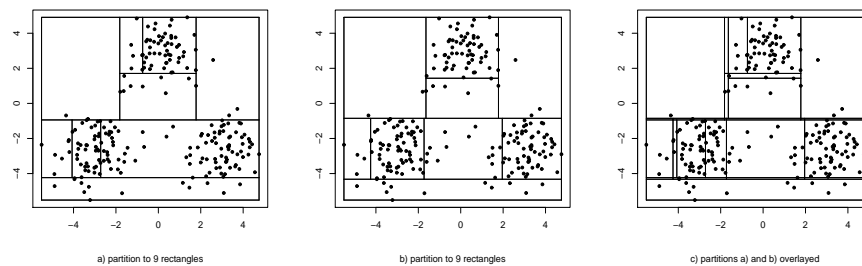


Figure 5: Two partitions and their overlaying. The data is generated from the density defined in (9) with $d = 2$, $D = 6$, and sample size 225.

computational efficiency.

References

- Baïllo, A., Cuesta-Albertos, J. A. and Cuevas, A. (2001), ‘Convergence rates in nonparametric estimation of level sets’, *Statist. Probab. Lett.* **53**, 27–35.
- Baïllo, A., Cuevas, A. and Justel, A. (2000), ‘Set estimation and nonparametric detection’, *Canadian J. Statist.* **28**, 765–782.
- Breiman, L. (1996a), ‘Bagging predictors’, *Machine Learning* **24**, 123–140.
- Breiman, L. (1996b), ‘Heuristics of instability and stabilization in model selection’, *Ann. Statist.* **24**, 2350–2383.
- Breiman, L., Friedman, J., Olshen, R. and Stone, C. J. (1984), *Classification and Regression Trees*, Chapman & Hall, New York.
- Devroye, L. and Wise, G. L. (1980), ‘Detection of abnormal behavior via nonparametric estimation of the support’, *SIAM J. Appl. Math.* **38**, 480–488.
- Fan, J. and Marron, J. S. (1994), ‘Fast implementations of nonparametric curve estimators’, *J. Comput. Graph. Statist.* **3**, 35–56.
- Freund, Y. (1995), ‘Boosting a weak learning algorithm by majority’, *Information and Computation* **121**, 256–285.
- Freund, Y. and Schapire, R. (1996), Experiments with a new boosting algorithm, in ‘Machine Learning: Proceedings of the Thirteenth International Conference’, Morgan Kaufman, San Fransisco, pp. 148–156.

- Friedman, J. H., Stuetzle, W. and Schroeder, A. (1984), ‘Projection pursuit density estimation’, *Amer. Statist. Assoc.* **79**, 599–608.
- Hall, P. and Wand, M. P. (1996), ‘On the accuracy of binned kernel density estimators’, *J. Multivariate Anal.* **56**, 165–184.
- Härdle, W. and Scott, D. (1992), ‘Smoothing by weighted averaging of rounded points’, *Comput. Statist.* **7**, 97–128.
- Holmström, L. (2000), ‘The error and the computational complexity of a multivariate binned kernel density estimator’, *J. Multivariate Anal.* **72**, 264–309.
- Holmström, L., Hoti, F. and Klemelä, J. (2005), ‘Flexible multivariate histograms and level set tree plots’. Manuscript.
- Korostelev, A. P. and Tsybakov, A. B. (1993), ‘Estimation of the density support and its functionals’, *Probl. Inf. Transm.* **29**, 1–15.
- Schapire, R. (1990), ‘The strength of weak learnability’, *Machine Learning* **5**, 197–227.
- Scott, D. W. (1985), ‘Average shifted histograms: effective nonparametric density estimators in several dimensions’, *Ann. Statist.* **13**, 1024–1040.
- Scott, D. W. (1992), *Multivariate Density Estimation*, Wiley.
- Scott, D. W. and Wand, M. P. (1991), ‘Feasibility of multivariate density estimates’, *Biometrika* **78**, 197–205.
- Seifert, B., Brockmann, M., Engel, J. and Gasser, T. (1994), ‘Fast algorithms for nonparametric curve estimation’, *J. Comput. Graph. Statist.* **3**, 192–213.
- Walther, G. (1997), ‘Granulometric smoothing’, *Ann. Statist.* **25**, 2273–2299.
- Wand, M. P. (1994), ‘Fast computation of multivariate kernel estimators’, *J. Comput. Graph. Statist.* **3**, 433–445.