Scalable Kernel Density Classification via Threshold-Based Pruning

Edward Gan, Peter Bailis
Stanford InfoLab

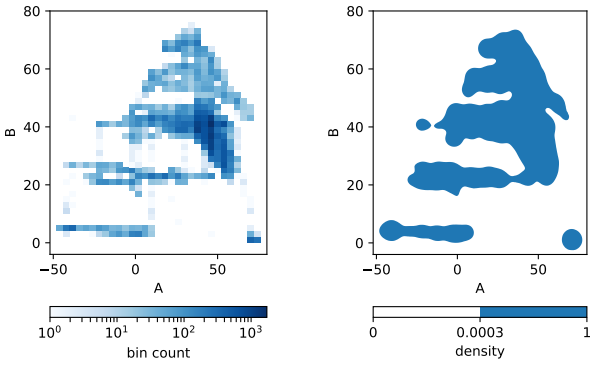
ABSTRACT

Density estimation forms a critical component of many analytics tasks including outlier detection, visualization, and statistical testing. These tasks often seek to classify data into high and low-density regions of a probability distribution. Kernel Density Estimation (KDE) is a powerful technique for computing these densities, offering excellent statistical accuracy but quadratic total runtime. In this paper, we introduce a simple technique for improving the performance of using a KDE to classify points by their density (density classification). Our technique, thresholded kernel density classification (tKDC), applies threshold-based pruning to spatial index traversal to achieve asymptotic speedups over naïve KDE, while maintaining accuracy guarantees. Instead of exactly computing each point's exact density for use in classification, tKDC iteratively computes density bounds and short-circuits density computation as soon as bounds are either higher or lower than the target classification threshold. On a wide range of dataset sizes and dimensions, tKDC demonstrates empirical speedups of up to 1000x over alternatives.

1. INTRODUCTION

As data volumes grow too large for manual inspection, constructing accurate models of the underlying data distribution is increasingly important. In particular, estimates for the probability distribution of a dataset form a key component of analytics tasks including spatial visualization [16, 17, 29], statistical testing [15, 33], physical modeling [5, 23], and density-based outlier detection [4, 19]. In each of these use cases, density estimation serves as a common primitive in classifying data into low and high-density regions of the distribution [9, 10, 54]. We refer to this task as *density classification*.

As an example of density classification, consider the distribution of two measurements from a space shuttle sensor dataset [34], illustrated in Figure 1a. The underlying probability distribution for these readings—even in two dimensions—is complex: there are several regions of high density, with no single cluster center, and a considerable amount of fine-grained structure. A high-fidelity model of the probability density distribution would enable several analyses. Identifying points lying in low-density fringes of the distribution can help identify rare operating modes of the shuttle. Computing the contour lines that separate the clusters can illustrate constraints on shuttle operation. Bounding the probability density of a given point lets us assign p-values to a given observation and perform statistical tests. Each of these tasks requires density classification, i.e. building a model of the distribution and using it to compare a density estimate against a threshold. Figure 1b depicts how density classification identifies points that lie above a density threshold.



(a) Histogram of measurements, cells colored by count.

(b) Classifying points with probability density $p > 0.0003$.

Figure 1: Measurements A and B (from columns 4 and 6 of the shuttle dataset) come from a complex two-dimensional distribution. Density classification identifies points with high probability density in the distribution.

Developing accurate and efficient models for these complex distributions is difficult. Popular parametric models such as Gaussian and Gaussian Mixture Models [6] make strong assumptions about the underlying data distribution. When these assumptions do not hold—as in the shuttle dataset—these methods deliver inaccurate densities. Moreover, even when their assumptions hold, popular parametric methods can require extensive parameter tuning. In contrast, *non-parametric* methods such as Kernel Density Estimation (KDE) [56], k-nearest neighbors (kNN) [43], and One-Class SVM (OCSVM) [48] can model complex distributions with few assumptions but are in turn much more computationally expensive.

In particular, KDE dates to the 1950s [46] and is the subject of considerable study in statistics, offering the benefit of asymptotically approximating any smooth probability distribution [50]. Moreover, KDE provides normalized and differentiable probability densities [52] that are useful in domains including astronomy [23] and high-energy physics [15]. These properties make KDE ideal for the density classification use cases outlined above. However, when implemented naïvely, the total runtime cost of density estima-

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064035

tion is quadratic in dataset size; calculating density estimates for a two-dimensional dataset of only 500 thousand points takes over two hours on a 2.9 GHz Intel Core i5 processor.

In this paper, we show that, when used in density classification, much of the computational overhead in computing kernel density estimates is unnecessary. We improve the performance of KDE-based density classification both asymptotically and empirically by up to three orders of magnitude by pruning density estimation calculations according to the target classification threshold. That is, instead of expending computational resources computing a precise density to be used in classification, we instead iteratively refine *bounds* on the density by traversing a spatial index. We short-circuit the density computation as soon as these bounds are above or below the target threshold. This way, we can quickly distinguish points in dense regions from points in sparse regions, only paying for more precise density estimates on query points close to the threshold. This avoids the overwhelming majority of kernel evaluations required for density estimation while still guaranteeing classification accuracy.

To apply this idea, we develop *Thresholded Kernel Density Classification* (tKDC), an efficient technique for performing kernel density classification. tKDC leverages two major observations:

First, tKDC incorporates Gray and Moore's prior insight that spatial kd-tree indices can be used to group points into regions, each of which can be iteratively refined to deliver increasingly accurate estimates [26]. This existing optimization yields an approximate estimate within ε of the true density. tKDC takes this observation a step further: instead of computing the true density within ε, we can stop as long as our bound places a point above or below the classification threshold. That is, tKDC pushes the density classification predicate into the process of approximate density calcluation. tKDC maintains upper and lower bounds on the estimated density and stops index traversal (i.e., kernel computations) when the bounded density is guaranteed to be either higher or lower than the classification threshold. This additional pruning rule yields orders-of-magnitude savings in the number of computations required to make an accurate classification. For d-dimensional data ($d > 1$), this pruning rule asymptotically reduces the complexity of computing the density of a single point from $O(n)$ to $O(n^{\frac{d-1}{d}})$.

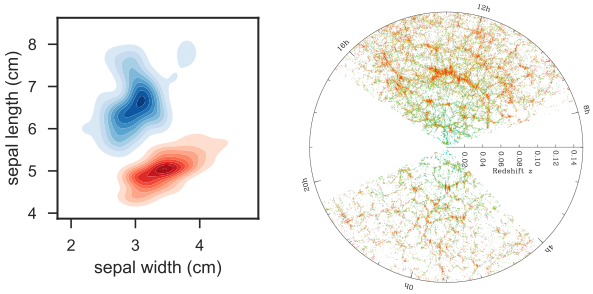
Second, densities can vary dramatically across datasets. Depending on the dataset, raw probability density values can differ by orders of magnitude. Specifying an a-priori density threshold is difficult. This leads to a chicken-and-egg problem for classification: tKDC must estimate densities in order to pick a good classification threshold, but estimating densities without a threshold as a guide is expensive. To address this, we develop a sampling-based algorithm for accurately estimating a quantile target threshold (i.e., one useful for classifying a given proportion of the data).

We evaluate the accuracy, runtime performance, and scalability of a tKDC prototype on a range of real-world datasets. In each case, tKDC achieves speedups up to 1000x compared with state-of-the-art alternative KDE approximation algorithms while providing bounds on its accuracy.

We make the following contributions in this paper:

- tKDC, a technique for KDE density classification that uses thresholds to prune kernel evaluations.
- A fast, sample-based technique for bootstrapping density quantiles, which tKDC uses for classification.
- An empirical evaluation of tKDC's accuracy and runtime, illustrating order-of-magnitude speedups over alternatives.

The remainder of this paper proceeds as follows: in Section 2, we provide additional background on kernel density estimation and the density classification task. Section 3 describes tKDC and presents a runtime analysis, Section 4 evaluates tKDC on a range of real-world datasets, Section 5 describes related work, and Section 6 concludes.



(a) Probability Density Contours from iris sepal measurements [24]. Region contours provide intuition for biological distinctions.

(b) Distribution of Galaxy mass across space [1, 7], probabilities densities signify physical mass densities.

Figure 2: Two applications of density classification

2. BACKGROUND AND TASK STATEMENT

In this section, we provide additional background on motivating use-cases, formally introduce Kernel Density Estimation, and define our target task: density classification.

2.1 Target Use Cases

When modeling a probability distribution, probability density values are essential in enabling a number of further analyses. In fact, in these cases, we may not need to compute the density values themselves. Classifying whether points have densities above or below a threshold (density classification) independently enables a number of tasks including:

1. density-based outlier classification,
2. spatial contour visualization, and
3. density-based statistical testing.

We motivate these with example scenarios below:

Outlier Classification. Given a data distribution, we can identify any points lying in low probability-density regions as outliers. For instance, a production engineer examining the shuttle measurement data (Figure 1a) can search for times when the shuttle entered unusual operating modes. The measurements lying in the low density filaments between larger clusters are natural outlier candidates, and ground truth data confirms that most of these low density observations in fact occurred during unusual operating states. Probability densities have been used for similar outlier classification tasks in computer vision, fraud detection, and traffic accident analysis [4, 19, 48, 49]. Unsupervised density-based outlier classification would be especially useful in explanation engines such as MacroBase [3]. Section 5 provides more details on using KDE for outlier detection compared with other methods.

Visualizing and Modeling Region Boundaries. The probability distribution of a dataset also allows us to understand the contour lines (i.e., level sets) that divide data points into distinct clusters and regions. Figure 2a illustrates the distribution of sepal measurements from a dataset of Iris flower measurements [24]: there are two dominant modes and a sparse region separating the two. For a biologist, understanding the shape of the contour lines defining these modes can yield valuable intuition. Scientific insights have been made possible by visualizing probability density contours to study volcanic lava flow [5] and understand the movement patterns of migratory whales [42]. In fact, as seen in visualization functions available in

popular Python [57] and R [20] packages, as well as visualization research efforts [41], one of the most common motivations for modeling data probability distributions is to visualize the boundaries of these high and low density regions. Automated procedures can also take advantage of knowing the region boundaries to perform clustering [16, 17] and run simulations [53].

Probability Densities for Statistics and Physics. Finally, a distribution estimate allows us to study other statistical and physical quantities that are depend on probability densities. Figure 2b depicts a cross section of the Sloan Digital Sky Survey: a multi-band, multi-dimensional survey of galaxy emissions [1]. Low probability density regions in this dataset have been used to successfully identify trends in *physical mass* distribution [23], allowing scientists to study for instance how galaxies formed in regions of space with low-mass density differ in spectrum. In statistics, bounds on the probability density also translate directly into bounds on hazard rate [51] or log likehood ratios [50] which are used in high energy physics classifiers [15], and there are direct statistical techniques for translating bounded density regions into classification confidence intervals [33].

Identifying points in low and high density regions is key to enabling the all of the above use cases, motivating the study of fast density classification algorithms.

2.2 Density Classification

Given a dataset X with probability density estimate $f(x)$ and a set of query points X_q, the density classification task consists of identifying whether $f(x_q)$ is above or below a threshold t for each point $x_q \in X_q$. This is equivalent to the statistical level-set identification problem, except that in our setting we seek to classify points according to the density estimate f itself, rather than the unobserved true distribution.

By adjusting t, density classification can serve a variety of uses. For relatively small t, classifying points with $f(x_q) < t$ allows us to identify outliers, or points that lie in the least likely regions of the distribution. For moderate t, we can estimate the shape of contour lines. We can also adjust t to provide bounds on probability densities for downstream statistical or physics-based tasks.

Figure 1b illustrates the results of estimating a probability distribution based on the shuttle data (Figure 1a) and then performing density classification on possible query points in that region of space. Points with density above the threshold are colored and those below are left uncolored. The shape and body of the distribution are clear from the classification contour, and these results would be valuable for many of the use cases discussed earlier.

This strict definition of density classification is convenient but difficult to implement efficiently. Thus, as in other fast KDE implementations [60], in this work we focus on solving an *approximate* framing of the density classification problem. By approximate we mean that classification errors are allowed for densities very close to (i.e., within $\pm\varepsilon t$ of) the threshold density t. This allows the algorithm to avoid the expensive computations required to make precise borderline decisions while bounding the severity of possible misclassifications. Note that ε does not define an absolute additive precision. Since our goal is to make classifications for different t with possibly widely varying magnitudes, precision is defined relative to t. An absolute *additive* precision of $\varepsilon_{abs} = 0.01$ would be unacceptably coarse for small thresholds $t < \varepsilon$.

This leads us to our final problem statement (Problem 1) for approximate density classification. Our algorithm, tKDC, solves this classification problem without explicitly computing $f(x_q)$, and is described in Section 3.

Problem 1 (Density Classification)**.** Given a dataset X with KDE $f(x)$ and threshold t to classify query points $x_q \in X_q$ as:

$$\begin{cases} \text{HIGH} & \text{when } f(x_q) > t \cdot (1+\varepsilon) \\ \text{LOW} & \text{when } f(x_q) < t \cdot (1-\varepsilon) \end{cases}$$

with undefined behavior otherwise.

2.3 Density Thresholds

Density classification as defined in Problem 1 is parameterized by a density threshold t. In practice, raw probability densities are relatively unwieldy: depending on the dataset size, dimensionality, and distribution, the range of densities in a distribution varies substantially, and it is difficult to a priori set thresholds for new datasets. Instead, it is useful to be able to specify a threshold in terms of a probability $p \in [0,1]$. That is, domain experts often have an idea of what fraction of the data they would like to classify as low density and set the threshold accordingly. Thus from this point forwards we will work with *quantile thresholds* $t^{(p)}$ [10].

In theory we would like to define the quantile threshold $t^{(p)}$ to be the point at which $f(x) < t^{(p)}$ with probability p. In other words, we would ideally let $t^{(p)} = \sup\{t : \Pr[f(x) < t] \leq p\}$ as in [10]. However, since we lack access to the true underlying distribution this $t^{(p)}$ is difficult to estimate and we instead define $t^{(p)}$ in terms of quantiles of the observed density estimates $f(x)$ for $x \in X$. The authors in [10] show that for kernel density estimators this quantile converges to the ideal $t^{(p)}$ hinted at above. Thus, in this work we will define $t^{(p)}$ in terms of the sample quantiles.

Let the quantile function $q_p(S)$ be defined on sets of real numbers S such that $q_p(S)$ is the (np) order statistic of S, i.e. the np-th smallest element of S. Then, let $t^{(p)}$ be defined to be the p quantile of the densities $\{f(x) - f_0 : x \in X\}$:

$$t^{(p)} := q_p\left(\{f(x) - f_0 : x \in X\}\right) \tag{1}$$

There is a bias here in using the same data points to train and then evaluate a density, so to compensate we subtract out the contribution a point in the dataset X makes to itself. The exact value of f_0 depends on the estimator used.

The threshold t in density classification can be arbitrarily specicfied, but since $t^{(p)}$ is defined in terms of the densities $f(x)$, it must be computed from the data. Thus, we present an algorithm for probabilistically estimating $t^{(p)}$ in Section 3.5. The quantile threshold estimation algorithm relies on sampling and thus has an adjustable failure probability δ, but our density classification algorithm is otherwise deterministic.

2.4 Kernel Density Estimation

Having defined density classification and the thresholds $t^{(p)}$, so introduce the kernel density estimate f which provides the densities we use in density classification. Kernel Density Estimation (KDE) provides a means of estimating a normalized probability density function $f(x)$ from a set of sample training data points X.

KDE can approximate most well-behaved arbitrary distributions with continuous second derivative [50]. Given n data points in d dimensions, the Mean Squared Error MSE shrinks at a rate $MSE \sim O(n^{-\frac{4}{4+d}})$. This is a powerful property: given enough data, KDE will identify an accurate distribution. In contrast, parametric methods are limited by their assumptions: for example, a mixture model offive Gaussians will be unable to accurately capture distributions that contain more than five distinct regions of high density. Other density estimation techniques such as histograms require asymptotically more data to achieve the same error [50], while methods like k-

	Role	Type	Default	Description
X	Input	$\{x_i \in \mathbb{R}^d\}$		Training Dataset
X_q	Input	$\{x_q \in \mathbb{R}^d\}$		Query points
$t^{(p)}$	Output	$\mathbb{R}$		Classification Threshold
$c(x_q)$	Output	$\{\text{LO}, \text{HI}\}$		Classification
b	Param	$\mathbb{R} > 0$	1	Bandwidth factor
p	Param	Probability	0.01	Classification rate
δ	Config	Probability	0.01	Failure probability
ε	Config	$\mathbb{R} > 0$	0.01	Multiplicative error

Table 1: Density Classification Task. Given X, calculate $t^{(p)}$ with failure probability δ. Then for $x_q \in X_q$ classify $c(x_q)$ according to the threshold with precision ε. The main parameters are the threshold probability p and bandwidth factor b.

nearest-neighbors classification do not provide smooth, normalized probability distributions [52].

KDE constructs an estimate of the probability density by summing contributions from small *kernel* distributions centered at each point. That is, each point in X contributes a small amount of local density to the overall distribution, and the probability density estimate at a given *query point* x_q is the sum of these contributed probabilities.

The *kernel function* K_H controls how the density contribution of each point in X falls off over distance: each data point contributes more density to nearby locations. Kernel functions are parameterized by a bandwidth matrix $H \in \mathbb{R}^{d \times d}$ that specifies how quickly the kernel falls off along different directions. The Gaussian kernel family given in Equation 2 leads to very smooth density estimates and we will use them by default in this paper. The bandwidth H here corresponds to the covariance of the Gaussian:

$$K_H(x) = \frac{1}{(2\pi)^{d/2}|H|^{1/2}} e^{-\frac{1}{2}x^T H^{-1} x} \tag{2}$$

Given a set of n training points $X = \{x : x \in \mathbb{R}^d\}$ and Kernel function K_H, the Kernel Density Estimate is then the probability density function $f(x_q) : \mathbb{R}^d \to \mathbb{R}$:

$$f(x_q) = \frac{1}{n} \sum_{x_i \in X} K_H(x_q - x_i) \tag{3}$$

For a training sample N of size n, KDE effectively acts a Gaussian Mixture Model with n Gaussians. The main parameter in KDE is the kernel bandwidth H. *Bandwidth selection* determines the amount of smoothing performed by KDE and there are many existing techniques for choosing a bandwidth parameter [31, 44]. The techniques in this work do not depend on specific kernel and bandwidth choices, so for simplicity we adapt standard product kernels with diagonal bandwidth $H = \text{diag}(h_1^2, \ldots, h_d^2)$ and Scott's rule for bandwidth selection (Equation 4) [50].

$$h_i = b \cdot n^{-\frac{1}{d+4}} \sigma_i \tag{4}$$

These are near-optimal choices for approximating multivariate normal distributions and serve as useful starting points for other data distributions. In Equation 4, b is a user-defined scale factor to allow for fine-tuning the bandwidth chosen by Scott's rule, and σ_i is the standard deviation of the i-th components of X, $\sigma_i = std(\{x^{(i)} : x \in X\})$.

3. tKDC OVERVIEW

Overview. In this section we present our algorithm for solving the approximate density classification problem defined in Section 2.2. Table 1 outlines the input, output, and parameters for the density classification task which our algorithm will address.

Our algorithm, Thresholded Kernel Density Classification (tKDC), constructs a spatial index over the dataset X and computes upper and lower bounds on the kernel density $f(x_q)$ in order to make a classification. tKDC takes advantage of a classic query optimization technique: predicate pushdown, in order achieve significant speedups over naïve density estimation.

Bounds via Spatial Indices. A naïve computation of $f(x_q)$ is prohibitively expensive: it involves accumulating the kernel contributions from every point in X. Computing upper and lower bounds instead of exact densities is much more efficient and still provides quantifiable accuracy guarantees. tKDC computes bounds on each density $f(x_q)$ by making use of a spatial index over the dataset X. This index gives us a way to group points into contiguous regions of $\mathbb{R}^d$ and lets us compute the minimum and maximum density contribution from each region. In fact, tKDC works with upper and lower bounds f_u, f_l for $f(x_q)$ instead of computing $f(x_q)$ exactly.

Predicate Pushdown. Predicate pushdown works well when applied to these bounds. Rather than computing expensive but precise bounds for $f(x_q)$ only to later perform a cheap comparison with $t^{(p)}$, we can push the threshold checks into the density computation. If we find that $f_u < t^{(p)}$ for instance, $f(x_q)$ must be less than $t^{(p)}$ and further computation is unnecessary for classification. We call these predicates pruning rules. Our key insight is that, since tKDC attempts to classify points rather than estimate exact densities, points far away from the threshold $t^{(p)}$ require only a coarse bound, and resources can be invested into estimating densities near the threshold more precisely.

Threshold Estimation. The major difficulty with using these pruning rules to speed up density classification is that they require knowing $t^{(p)}$. $t^{(p)}$ is also difficult to calculate exactly since we define it in terms of the densities of points in X. Thus, we instead calculate probabilistic upper and lower bounds $t_u^{(p)}, t_l^{(p)}$ on $t^{(p)}$. With probability $1 - \delta$, the true $t^{(p)}$ will lie within these bounds, and we can then use these bounds to estimate $t^{(p)}$ to within multiplicative error ε and perform approximate density classification.

Pseudocode. Algorithm 1 presents the pseudocode for tKDC with references to subroutines we will discuss later. First tKDC calculates probabilistic initial bounds on $t^{(p)}$ (BoundThreshold) and constructs a spatial index T on X (MakeIndex). This constitutes the training phase. Then, tKDC calculates bounds f_l, f_u on the densities of each point in X (BoundDensity). These point density bounds allow us to get a more precise estimate $\tilde{t}^{(p)}$ for $t^{(p)}$ by calculating the p-quantile q_p of D_x. Finally, for each query point $x_q \in X_q$, to classify it (Classify), we can calculate bounds on its density and compare it with the threshold estimate $\tilde{t}^{(p)}$.

In the following sections, we start by assuming that initial coarse bounds $t_u^{(p)}, t_l^{(p)}$ are provided by an oracle and discuss how bounds on $f(x_q)$ are computed. Subsequently, we explain how tKDC bootstraps initial coarse bounds on $t^{(p)}$, discuss additional optimizations, and analyze the runtime performance of the algorithm.

3.1 Bounds via Spatial Indices

k-d trees [47] provide a useful spatial index for computing upper and lower bounds on the kernel density. Most kernels (including the Gaussian) fall off rapidly with increasing distance, so grouping neighboring points into regions allows us to calculate upper and lower bounds on the exact density $f(x_q)$ without explicitly evaluating each kernel. Thus, we incorporate existing techniques for using k-d trees to evaluate kernel densities [26].

k-d Trees. A k-d tree is a binary tree index over points $X \subset \mathbb{R}^d$. Figure 3 illustrates the first two levels of a k-d tree over 2-

Algorithm 1 tKDC: Approximate Density Classification

$t_u^{(p)}, t_l^{(p)} \leftarrow$ BOUNDTHRESHOLD(X)
$T \leftarrow$ MakeIndex(X) ▷ Construct Spatial Index
$D_x \leftarrow [\cdot]$ ▷ Density estimates for $x_i \in X$
for $x_i \in X$ **do**
  $f_l, f_u \leftarrow$ BOUNDDENSITY($T, t_u^{(p)}, t_l^{(p)}, x_i$)
  append($D_x, (f_l + f_u)/2 - \frac{1}{N} K_H(0)$)
$\tilde{t}^{(p)} \leftarrow q_p(D_x)$ ▷ Approximate threshold

function CLASSIFY(x_q)
  $f_l, f_u \leftarrow$ BOUNDDENSITY($T, \tilde{t}^{(p)}, \tilde{t}^{(p)}, x_q$)
  if $(f_l + f_u)/2 > \tilde{t}^{(p)}$ **then**
    return HIGH
  else
    return LOW

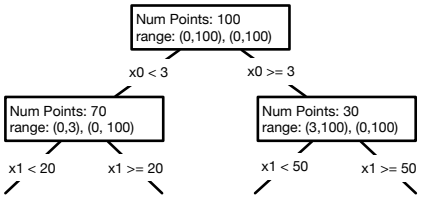


Figure 3: 2-dimensional k-d tree. Each node splits points along a specific dimension, and keeps track of both a bounding box range for the region it defines and the number of points contained within.

dimensional data points. Starting from the root node, each node defines a region of space and splits its region along one coordinate among its children. Thus, in figure 3 each point $x^{(j)} \in X$ would be assigned to one of the child nodes depending on whether $x_0^{(j)} < 3$. Each non-leaf node in the tree has two child nodes while each leaf node keeps track of the sample values contained inside. There are many standard techniques for choosing the axis along which to split, for tKDC we default to cycling through the dimensions in sequence, one for each level of the tree, so that in the worst case each axis will be considered regularly. In addition, we adapt some of the features of multi-resolution k-d trees [18]: each node in our tree keeps track of the number of points in its region as well as its bounding box.

Distance Bounds. The *bounding box* of a node is a conservative estimate of the region of space occupied by the points belonging to the node. In tKDC, this region is represented by a sequence of minimum and maximum coordinate values x_i^{min}, x_i^{max} for the points under a node and for each coordinate axis i. Given x_q, since the k-d tree tracks the number of points in a region as well as its bounding box, we can compute upper and lower bounds on the density contribution of an entire region of points [26]. For a region containing a subset of points X_r, the total kernel density contribution $f^{(r)}(x_q)$ is given as:

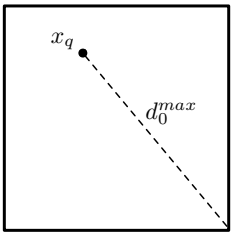
$$f^{(r)}(x_q) = \sum_{x_i \in X_r} \frac{1}{n} K_H(x_q - x_i) \tag{5}$$

$f^{(r)}(x_q)$: is bounded by the smallest and largest distance vectors d_{max}, d_{min} from x_q to the bounding box of X_r.

$$\frac{|X_r|}{n} K_H(d_{max}) \leq f^{(r)}(x_q) \leq \frac{|X_r|}{n} K_H(d_{min}) \tag{6}$$

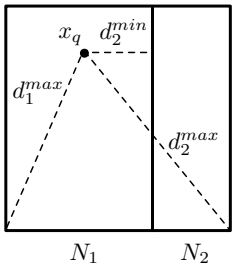
3.2 Iterative Refinement

As seen in the previous section, each node in the k-d tree has a bounding box which constrains the density contribution from points in its region. If the bounds are too coarse however, we need a way to improve them. This can be done by replacing the bound obtained from one node of the k-d tree with the bounds obtained from its children: the same underlying data points are still being counted, but now each point is constrained to a smaller region and we can obtain a better bound. Figure 4 illustrates how the bounds can be improved.



$$f_l = \frac{N}{N} K(d_0^{max})$$
$$f_h = \frac{N}{N} K(d_0^{min} = 0)$$

(a) Iteration 1: the root bounding box gives us a very coarse bound on the total kernel density. The two extremes are all points coincident with x_q or all points located at the furthest corner.



$$f_l = \frac{N_1}{N} K(d_1^{max}) + \frac{N_2}{N} K(d_2^{max})$$
$$f_h = \frac{N_1}{N} K(0) + \frac{N_2}{N} K(d_2^{min})$$

(b) Iteration 2: dividing the root node into its two child nodes gives us finer grained bounding boxes and a tighter final kernel density bound.

Figure 4: Iterative k-d tree refinement: the total density contribution from X is represented as a sum from disjoint subsets of X, each belonging to a node of the k-d tree. As nodes are replaced with their two children we get more and more precise estimates.

Starting with the root node, we can obtain a loose bound on the total density $f(x_q)$: the minimum possible density contribution would occur if all of the points were located at the furthest corner, with kernel value $K(d_{max})$, and similarly the maximum possible density contribution would occur if all points were exactly x_q, with kernel value $K(0)$. If we replace the root node with its two children, we are left with two distinct subregions with N_1, N_2 points in each. This leads to bounds using the new minimum and maximum distance vectors from x_q to points in the respective subregions: in particular no point in the second region can contribute more than $K(d_{2,min})$. This process is continued until the bounds are good enough (fulfilling our pruning rules) or we have exhausted the k-d tree and evaluated each leaf node's contribution exactly.

To summarize, for a collection of k-d tree nodes that partition X into disjoint subsets $\{X_i\}$ with bounding boxes $\{B_i\}$, we can bound the kernel density estimate $f(x_q)$ with:

$$f_l = \sum_i \frac{|X_i|}{n} K_H(d_{max}(x_q, B_i)) \quad f_u = \sum_i \frac{|X_i|}{n} K_H(d_{min}(x_q, B_i)) \tag{7}$$

Iteratively replacing nodes with their children provides incrementally refined bounds.

3.3 Pruning Rules

Tolerance. Iteratively refining the bounds provided by a set of k-d tree nodes gives us a sequence of more precise bounds:

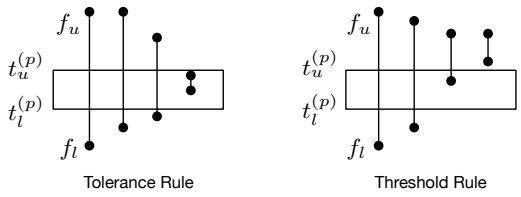


Figure 5: Pruning Rules: As the upper and lower bounds are refined, we can stop computation once the interval f_l, f_u for $f(x_q)$ is confirmed to lie on one side of the threshold, or the interval is narrower than $\varepsilon t^{(p)}$

$\{(f_l^{(i)}(x_q), f_u^{(i)}(x_q))\}$. The key to efficient computation in tKDC is knowing when these bounds are good enough by checking them against *pruning rule* predicates. One criteria, introduced in [26], is to stop when the upper and lower bounds are within a fraction ε of each other. This can result in savings when all nearby regions have been precisely resolved and only very distant regions remain. Thus, we use Equation 8 as one of our pruning rules, and refer to it as the *Tolerance Rule*:

$$f_u^{(i)}(x_q) - f_l^{(i)}(x_q) < \varepsilon t_l^{(p)} \tag{8}$$

Threshold. Since we wish to classify rather than estimate densities, we can go even further. Well before f_u and f_l are precise, we will often know enough to make a classification: if f_l is above the threshold or f_u is below, then no further computation is necessary for classification. This realization is key to the performance gains tKDC delivers. We encode this idea in Equation 9 and refer to these predicates as the *Threshold Rules*:

$$f_l^{(i)} > t_u^{(p)}(1+\varepsilon) \text{ or } f_u^{(i)} < t_l^{(p)}(1-\varepsilon) \tag{9}$$

The threshold rules are responsible for the vast majority of the speedups made possible by tKDC. Furthermore, both the tolerance and threshold rules allow us to confidently make classifications with respect to $t^{(p)} \pm \varepsilon t^{(p)}$.

Figure 5 illustrates how the tolerance and threshold rules allow tKDC to stop when it has enough information to make a classification. tKDC refines upper and lower bounds on the density until it can apply one of the pruning rules, stopping when the bounds are either clear of $t^{(p)}$ or within $\varepsilon t^{(p)}$ of each other.

3.4 Bounding the Density

tKDC combines the k-d tree density bounds and threshold and tolerance pruning rules by using a priority queue to control the order in which we traverse the k-d tree. We would like to prioritize nodes with the highest potential for improving the total density bound, so the queue prioritizes nodes with the largest discrepancy $n_r(K_H(d_{min}) - K_H(d_{max}))$ where n_r is the number of points contained in the node region and d_{min}, d_{max} are the smallest and largest distances from x_q to the node bounding box. In practice, for small $t^{(p)}$ thresholds this prioritizes hitting the threshold rule as quickly as possible.

Algorithm 2 presents our procedure for bounding the kernel density $f(x_q)$. w_{min}, w_{max} here are functions which compute the minimum and maximum weight contribution for a node-region *node* given its bounding box and the number of points inside, as in Equation 6 in Section 3.1. For now we assume that an oracle has provided upper and lower bounds $t_u^{(p)}, t_l^{(p)}$ on the threshold, the algorithm to estimate $t^{(p)}$ will be explained in Section 3.5.

The bounds f_l, f_u provided by the BoundDensity procedure are useful for two purposes as seen in Algorithm 1: they are used to perform classification of query points x_q and they are also used to calculate bounds on $t^{(p)}$. Intuitively, this is possible because the f_l, f_u bounds are exact in classifying whether a density is outside of $t_l^{(p)}, t_u^{(p)}$, and are precise to within $\varepsilon t^{(p)}$ otherwise.

Algorithm 2 Approximate Density Estimation

```
function BoundDensity(T, t_l^(p), t_u^(p), x)
    pq ← [T]                                   ▷ Node Priority Queue
    f_u ← w_min(x, T)                          ▷ Weight Bounds
    f_l ← w_max(x, T)
    while pq not empty do
        if f_l > t_u^(p) then                  ▷ Threshold Rule
            break
        if f_u < t_l^(p) then
            break
        if f_u − f_l < ε · t_l^(p) then        ▷ Tolerance Rule
            break
        curnode ← poll(pq)
        f_l ← f_l − w_min(curnode)
        f_u ← f_u − w_max(curnode)
        if curnode is leaf then
            f_cur ← Σ_{x_i ∈ curnode} (1/n) K_h(x − x_i)
            f_l ← f_l + f_cur
            f_u ← f_u + f_cur
        else
            for child in children(curnode) do
                f_l ← f_l + w_min(child)
                f_u ← f_u + w_max(child)
                pq ← add(pq, child)
    return f_l, f_u
```

Using f_l, f_u for classification is fairly straightforward. As in Algorithm 1, if $\frac{1}{2}(f_l(x_q) + f_u(x_q)) > t_u^{(p)}$ or $\frac{1}{2}(f_l(x_q) + f_u(x_q)) < t_l^{(p)}$ then we can classify $f(x_q)$ exactly. Otherwise, Algorithm 2 must have run until it hit the tolerance rule, so $f_u - f_l < \varepsilon t^{(p)}$ and $\frac{1}{2}(f_l(x_q) + f_u(x_q))$ will be within $\varepsilon t^{(p)}$ of the true density $f(x_q)$. This is within the error tolerance allowed in our definition of the approximate density classification problem.

In order to perform approximate density classification we also need to bound $t^{(p)}$ to within $\pm\varepsilon t^{(p)}$ as defined in Problem 1. One way to do this is to calculate f_l, f_u for all $x \in X$ using the BoundDensity procedure. If $f(x_q)$ is close to the threshold we will keep improving the bounds until we hit the tolerance rule and $f_u - f_l < \varepsilon t^{(p)}$. Thus, calculating quantiles on $\left\{\frac{1}{2}(f_l(x) + f_u(x)) : x \in X\right\}$ allows us to estimate $t^{(p)}$ to within $\varepsilon t^{(p)}$ as well.

Thus, the BoundDensity procedure allows us to obtain bounds on both $f(x_q)$ and $t^{(p)}$ accurate to $\varepsilon t^{(p)}$ and good enough for approximate density classification. However, in order to run efficiently the BoundDensity procedure relies on having coarse initial bounds on $t^{(p)}$.

3.5 Threshold Estimation

One way to estimate initial coarse bounds on $t^{(p)}$ is to calculate the densities of a smaller random sample of points. The order statistics and thus quantiles of the smaller sample can provide bounds on the quantiles of the larger dataset. Given a set of n real numbers D, let D_s be a random sample of s of these numbers. Let $d^{(i)}$ be the i-th order statistic (the i-th smallest number) of D and $d_s^{(i)}$ be the i-th order static of D_s. Then, the binomial theorem gives us

Equation 10 [25].

$$\Pr\left(d_s^{(l)} \leq d^{(np)} \leq d_s^{(u)}\right) = \sum_{i=l}^{u} \binom{s}{i} p^i (1-p)^{s-i} \tag{10}$$

For large n the binomial bound is well approximated by a normal distribution, so we can simplify the above equation:

$$\Pr\left(d_s^{\left(sp-z_{1-\delta}\sqrt{sp(1-p)}\right)} \leq d^{(np)} \leq d_s^{\left(sp+z_{1-\delta}\sqrt{sp(1-p)}\right)}\right) \geq 1-\delta \tag{11}$$

where the constant z_p is the p-th quantile of the normal distribution. For an acceptable failure rate δ, this allows us to construct $1-\delta$ confidence intervals for $d^{(p)}$ by calculating densities on a random subsample X_s of s random query points rather than all of the points in X. Thus, the specified failure probability δ dictates how large of a sample we must collect, thus influencing training time. For instance, for $s = 20000$, $\delta = 0.01$, $p = 0.01$, if we calculate 20000 densities and sort them into $d^{(i)}$, then $z_{0.99} = 2.576$ so we have: $\mathbb{P}\left(d^{(164)} \leq t^{(0.01)} \leq d^{(236)}\right) \geq 0.99$ and the 164th and 236th densities provide a confidence interval for $t^{(p)}$

However we are now left with a chicken and egg problem: in order to efficiently estimate bounds on densities using Algorithm 2 we need upper and lower bounds on $t^{(p)}$, but to obtain bounds on $t^{(p)}$ we need to estimate densities for points in a subset X_s of X. Calculating even a single exact density on a KDE trained on X is expensive for large datasets. Instead, tKDC bootstraps itself by iteratively training kernel density estimates on larger and larger subsets of the data X, using quantile estimates on smaller subsets of the training data to obtain bounds used in later iterations. Rather than constructing the full KDE by adding up contributions from each point in X, we can construct mini-KDEs trained on subsamples of X. In other words, for a training subset $X_r \subseteq X$ we can compute kernel densities f_{X_r} using data just from this subset.

$$f_{X_r}(x_q) = \frac{1}{N_r} \sum_{x \in X_r} K_H\left(x_q - x\right)$$

We do not assume that f_{X_r} will provide an accurate estimate of f trained on the entire dataset, but the estimates provided by evaluating f_{X_r} serve as starting points in our bootstrapping procedure.

Algorithm 3 outlines the procedure for estimating upper and lower bounds for $t^{(p)}$. We can start by evaluating KDE densities with small X_r and use these to calculate initial coarse bounds for $t^{(p)}$. Each set of coarse bounds is used as a starting point for obtaining more accurate bounds in the next iteration with a larger X_r.

For example, if we have bounds $t_l^{(p)}, t_u^{(p)}$ calculated from a KDE trained on X_r, then we can use we can use these bounds when calculating densities for a KDE trained on X_{4r} a subsample 4 times the size of X_r. The BoundDensity routine returns density bounds that have precision $\varepsilon t_l^{(p)}$ for densities within the threshold bounds, so as long as enough of the new densities remain within the $t_l^{(p)}, t_u^{(p)}$ bound we can use them to compute a new threshold bound. There are no guarantees that the old bounds will continue to apply as we increase X_r (in fact the bounds for small r can be off by orders of magnitudes when translated to larger r), but we can check after evaluating densities if the bounds were too high or low and repeat the computation with more generous bounds by multiplicatively scaling them back. In particular, if the order statistics required to calculate the bounds in Equation (11) lie outside of the old threshold bound then we do not have enough precision and must repeat our calculation with more conservative bounds.

At the end of the threshold bounding routine (Algorithm 3), we will have calculated density bounds for s query points using a KDE trained on the complete dataset X. This gives us enough accuracy to determine $d^{(l)}, d^{(u)}$ the $1-\delta$ confidence bounds for $t^{(p)}$ to within $\varepsilon \cdot t^{(p)}$. The initial sample sizes r_0, s_0 do not affect the correctness of the algorithm, and $r_0 = 200, s_0 = 20000$ were found to provide reasonably fast performance on our datasets. Similarly the multiplicative factors $h_{backoff}, h_{buffer}$ which control how quickly we adjust bad threshold bounds and how much extra buffer we allow threshold bounds when moving to larger training samples, and the training sample growth rate h_{growth}, do not affect correctness. $h_{backoff} = 4, h_{buffer} = 1.5, h_{growth} = 4$ provide good performance in practice.

Algorithm 3 Bootstrapped Threshold Bound

function BOUNDTHRESHOLD(X)
- $t_l^{(p)} \leftarrow 0$ ▷ Threshold bounds
- $t_u^{(p)} \leftarrow \infty$
- $r \leftarrow r_0$ ▷ Num training points
- $s \leftarrow s_0$ ▷ Num query points
- **while** $r \leq N$ **do**
  - $X_r \leftarrow sample(X, r)$
  - $X_s \leftarrow sample(X_r, s)$
  - Build kdtree on X_r
  - Recalculate bandwidth
  - $\{f_l^{(i)}, f_u^{(i)}\} \leftarrow$ BOUNDDENSITY$(t_l^{(p)}, t_u^{(p)}, X_s)$
  - $\{d^{(i)}\} \leftarrow sorted\left(\left(f_l^{(i)} + f_u^{(i)}\right)/2 - \frac{1}{|X_r|}K_H(0)\right)$
    ▷ Density estimates, correcting for self-contribution
  - $l \leftarrow sp - z_{(1-\delta)}\sqrt{sp(1-p)}$
  - $u \leftarrow sp + z_{(1-\delta)}\sqrt{sp(1-p)}$
  - **if** $d^{(u)} > t_u^{(p)}$ **then** ▷ Invalid bound
    - $t_u^{(p)} \leftarrow t_u^{(p)} \cdot h_{backoff}$
  - **else if** $d^{(l)} < t_l^{(p)}$ **then** ▷ Invalid bound
    - $t_l^{(p)} \leftarrow t_l^{(p)} / h_{backoff}$
  - **else** ▷ Valid Bound
    - $t_u^{(p)} \leftarrow d^{(u)} \cdot h_{buffer}$
    - $t_l^{(p)} \leftarrow d^{(l)} / h_{buffer}$
    - $r \leftarrow \max(r \cdot h_{growth}, N)$
- **return** $\left(d^{(l)}, d^{(u)}\right)$

3.6 Classification Accuracy

With all of the major components of tKDC introduced, we can revisit Algorithm 1 to discuss the accuracy of its classifications. The BoundDensity rountine is deterministic and calculates exact (up to floating point precision) bounds on a density $f(x_q)$. From the two pruning rules, we know that either $f_l > t_u^{(p)}$ or $f_u < t_l^{(p)}$ and we can precisely classify a point x or else $f(x)$ is near the threshold and $f_u - f_l < \varepsilon t_l^{(p)}$.

Thus, assuming that $t_u^{(p)}, t_l^{(p)}$ are valid bounds for $t^{(p)}$, then the p-quantile of the densities D_x, $q_p(D_x)$, is an estimate $\tilde{t}^{(p)}$ for $t^{(p)}$ that is accurate to within $\varepsilon t_l^{(p)} < \varepsilon t^{(p)}$. Ignoring constant factors of $\varepsilon t^{(p)}$, this means that the Classify routine correctly classifies all points with densities more than $\varepsilon t^{(p)}$ away from $t^{(p)}$, and solves the density classification problem (Problem 1) for $t^{(p)}$.

With probability $1-\delta$, the initial probabilistic bounds $t_l^{(p)}, t_u^{(p)}$ are valid on $t^{(p)}$ and we furthermore have correctly classified densities with respect to $t^{(p)}$ as defined in Equation 1. However, there is a probability δ chance the bounds on $t^{(p)}$ are invalid, in which case

we have solved the density classification problem for an inaccurate threshold $\tilde{t}^{(p)}$. We can detect when this has occurred by counting what fraction of the points in X had densities which were higher than t_u or lower than t_l, and then repeat the threshold estimation procedure to try and obtain a valid bound.

3.7 Optimizations

Two other algorithmic optimizations proved useful in implementing tKDC efficiently: a grid for caching known dense regions and a custom k-d tree splitting rule.

Grid. Once a lower bound $t_l^{(p)}$ is known for the density threshold, tKDC tries to prune out obvious inlier points before even beginning a tree traversal. This can be done using a d-dimensional hypergrid with grid dimensions equal to the bandwidth of the data. Before evaluating any densities, a single pass through the datset allows us to count how many points lie within each grid cell. Then, future queries $f(x_q)$ can first be checked against the count $G(x_q)$ of points sharing a grid cell with x_q. If $G(X_q)/N \cdot K_H(d_{diag})$ where d_{diag} is the length of the diagonal, then x_q can be immediately classified above the threshold. The size of the grid can be tuned though we have found that setting the grid dimensions equal to the bandwidth works well for low dimensions. The grid provides noticeable performance improvements for small p thresholds and low dimensions but is not as useful for large p. Due to its poor scaling with dimensionality, we disable the grid for dimensions $d > 4$.

Equi-width Trees. k-d trees are usually constructed so that they are balanced: splitting each set of points along the median of an axis. However, this is not as efficient for tKDC and we have found that splitting each node at $\frac{1}{2}(x_i^{(10)} + x_i^{(90)})$ performs better, where $x_i^{(p)}$ is the p-th percentile of the data points along the ith coordinate. Since the Gaussian kernel falls off exponentially with distance it is more important to quickly identify tightly constrained regions than it is to identify regions with a roughly equal number of points inside. Splitting the index along a midpoint rather than median is also used in the formal runtime analysis in Appendix A.

3.8 Runtime Analysis

In this section, we analyze tKDC runtime as the size n of the training set X grows, where $X \in \mathbb{R}^{n \times d}$ is a d-dimensional dataset drawn from a distribution D. Since each classification is performed independently, we analyze the runtime cost of classifying a single query point $x \in \mathbb{R}^d$. We omit the cost of index construction (total $O(n \log n)$ time) and estimating the threshold (number of queries dependent on ε and δ) in this analysis.

Theorem 1. For a query point x drawn from D, tKDC runs in expected $O(n^{\frac{d-1}{d}})$ time when $d > 1$ and $O(\log(n))$ when $d = 1$.

Theorem 1 gives a runtime bound on the tKDC classification procedure. In contrast, the naïve strategy takes $O(n)$ time to compute the density of a given point. Moreover, any approximation that evaluates kernels on neighbors within a fixed distance of the query point (such as rkde) will also incur $O(n)$ running time, since the number of such points will be proportional to n. tKDC is asymptotically faster than these algorithms with substantial gains for small d. We provide more details in Appendix A and present a proof sketch here.

Recall that tKDC traverses a k-d tree index built over X, maintaining increasingly precise bounds on the query x's true density. We can analyze the behavior of this traversal in two cases: first, when the bounds provided by the index (f_l, f_u in Algorithm 2) are sufficiently precise to classify x, and, second, when the index bounds are insufficient and tKDC must examine some individual points within the leaf nodes of the k-d tree. These correspond to points whose densities are correspondingly far from (and easily distinguishable) or near the density threshold.

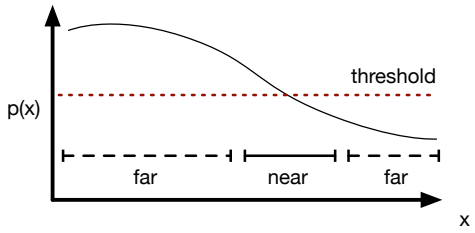


Figure 6: Near and Far queries: Far queries can be evaluated using only index lookups. Near queries are more expensive. The near region shrinks for larger n.

Definition 1. A *far* query point is one which tKDC can classify using only the bounds derived from the k-d tree index, while a *near* query point is one which tKDC must evaluate one or more exact kernel densities to classify.

For a given training dataset X, the possible far and near query points fall into regions of space $R_X^{far}, R_X^{near} \subseteq \mathbb{R}^d$. Figure 6 depicts these regions for a one-dimensional dataset. The near and far regions depend on the size n of the training data. In fact, larger training sets yield larger far-regions R_X^{far}. This is because adding more points to the training set (and thus index) improves the index precision and allows tKDC to classify more possible query points using just the index. Conversely, larger training sets X have smaller near-regions.

Lemma 1. The probability a query point x falling inside R_X^{near} is proportional to $O\left(n^{-\frac{1}{d}}\right)$

Lemma 1 states the above observation more precisely. Again, a proof is deferred to Appendix A. Given this, we can derive a bound on the expected runtime of a query.

Consider the average case for two training sets, one X_n of size n and one X_{2n} of size $2n$ from the same distribution D, with respective near regions $R_{X_n}^{near}, R_{X_{2n}}^{near}$. We can derive a recurrence relating the runtime cost for these two training sets. On average, any query points that were far for X_n are also far for X_{2n}. That is, an index traversal on the larger index over X_{2n} will suffice to classify any points that were far under X_n. On the other hand, the cost of evaluating the kernel for near points is $O(n)$ as, in the worst case, tKDC must evaluate the contribution from every point in the training set.

Even though near points are expensive to evaluate, the near region shrinks for larger n. By Lemma 1 an $O(n^{-\frac{1}{d}})$ proportion of query points will be near (requiring $O(n)$ computation) and the remaining far points have the same runtime cost under X_{2n} as they did under X_n. If we let F_n denote the expected runtime cost for X_n, and let F_n^{far}, F_n^{near} be the costs of evaluating far and near points respectively for training set X_n, then we can derive the recurrence:

$$\begin{aligned} F_{2n} &\leq F_n^{far} + \Pr\left(x \in R_{X_n}^{near}\right) \cdot F_{2n}^{near} \\ &\leq F_n + O\left(n^{-\frac{1}{d}}\right) \cdot O(2n) \\ &\leq F_n + O\left(n^{\frac{d-1}{d}}\right) \end{aligned}$$

By the master theorem [14], the runtime is then $O(n^{\frac{d-1}{d}})$ for $d > 1$ and $O(\log(n))$ for $d = 1$.

Name	Lang	Description
tKDC	Java	Density classification w/ pruning
simple	Java	Naïve algorithm, iterates through every point
sklearn	Cython	K-d tree approximation algorithm [26]
ks	C	Binning approximation algorithm [56]
rkde	Java	Contribution from only nearby points [47]
nocut	Java	tKDC with the threshold rule and grid disabled

Table 2: Algorithms used in evaluation

4. EVALUATION

In this section, we empirically evaluate tKDC's performance, accuracy, and scalability via a combination of synthetic and real world datasets. We focus on the following questions:

1. Does tKDC improve throughput? (§ 4.2)
2. Is tKDC accurate in classifying densities? (§ 4.3)
3. Does tKDC scale with dataset size and dimension? (§ 4.4)
4. How does each optimization in tKDC contribute? (§ 4.5)

Our results show that tKDC achieves up to 1000x speedups over other accurate approaches on our datasets and has excellent classification accuracy throughout. Notably, the cost of a single query scales sublinearly with dataset size as expected from the runtime analysis in Section 3.8, and tKDC remains faster than competing approaches across different dimensions and threshold values. Each optimization in tKDC plays an important role and the threshold pruning rule is especially valuable for efficient classification.

4.1 Setup

Environment. We implement tKDC in Java,[1] processing single-threaded memory-resident data. tKDC uses the Apache Commons FastMath library for expensive floating point operations such as exponentiation. We run experiments on a server with four Intel Xeon E5-4657L 2.40GHz CPUs containing 12 cores per CPU and 1TB of RAM. We measure throughput using wall-clock runtime including both training and query time. To isolate algorithmic runtime from data loading, we omit the time needed to load data from disk.

Unless otherwise stated, we measure both the time taken to train tKDC on a dataset by constructing a k-d tree and then estimating $t^{(p)}$ as well as the time taken to score queries from the same dataset. Thus, we measure throughput by amortizing the training time across the time taken to classify each point in a dataset. This is the effective throughput for performing tasks such as outlier detection using tKDC. When tKDC is used for other use cases with additional query points not in the training dataset, the training cost remains fixed and the performance should be even better.

Alternative Algorithms. We are unaware of alternative algorithms that specifically solve the density classification task for KDE. Thus we focus on comparing tKDC with two leading kernel density estimation implementations and three of our own baselines. These are summarized in Table 2. Scikit-learn [40] (sklearn) contains an implementation of KDE in cython (a wrapper for Python C-extensions) also based on k-d trees and the approximation techniques in [26], while the Kernel Smoothing "ks" R package [20] is written in C and implements an approximate KDE algorithm based on binning techniques in [55]. Scikit-learn KDE was run with default settings and $\varepsilon = 0.1$ relative error, and ks was run with default settings and binning enabled. Since ks and sklearn have their core routines written in C or C-like (cython) code, standard language benchmarks suggest that a Java implementation will be about a factor of two slower. Thus, any performance advantages in our Java implementations will be a conservative measure of the algorithmic speedups in tKDC. Furthermore, the "nocut" baseline we implemented reproduces the optimizations in sklearn and [26] and is usually around 2x slower than the scikit-learn implementation.

[1] Source code available at https://github.com/stanford-futuredata/tkdc

Name	d	n	Description
gauss	2	100M	Multivariate Gaussian with zero mean and unit covariance
tmy3	8	1.82M	Hourly energy load profiles for US reference buildings [39]
home	10	929k	Home Gas Sensor measurements from the UCI repository [28, 34]
hep	27	10.5M	High Energy Particle collision signatures from the UCI repository [34]
sift	128	11.2M	SIFT computer vision image features extracted from Caltech-256 [34]
mnist	784	70k	28x28 images of handwritten digits [32], reduced to smaller dimensions via PCA
shuttle	9	43.5k	Space shuttle flight sensors from the UCI repository [34]

Table 3: Datasets used in evaluation

We were unable to find many other implementations of KDE which supported $n > 2$ dimensions. For example, Spark ml-lib and Weka only support one-dimensional KDE. The ks library also only supports up to 4 dimensions with binning. Thus, as a baseline, we also benchmark against three of our own baselines implemented in Java. First, we implemented a naïve KDE (denoted "simple") where each kernel density is evaluated and summed explicitly. We also run tests against a version of tKDC with the threshold rule and grid disabled, but the tolerance rule still enabled with $\varepsilon = 0.01$. This baseline (called "nocut") emulates the functionality of the scikit-learn algorithm.

Finally, we implement an algorithm that performs a range query around the query point using same k-d tree as tKDC [47] to find all points within a certain radius of the query point, and then add up the kernel contributions from only those nearby points. We call this algorithm "rkde" for radial KDE, with radius set by default to the smallest possible radius with guaranteed error $\varepsilon = 0.01t$ based on the points excluded. The radius is thus set conservatively for most of our experiments, and we show in Figure 13 in Appendix B that even for very small distances r the same trends hold. We run all algorithms with the same bandwidth selection rule described in Section 2.

Datasets. Our experimental analysis makes use of seven datasets with varying size n and dimensionality d. We list the datasets in Table 3. Unless other stated we run queries over complete datasets, but ignore columns with more than 50% missing values in the tmy3 dataset.

4.2 End-to-End Throughput

In Figure 7, we compare the classification throughput including training time of tKDC with other algorithms on our datasets with at least 50k points. Here, we reduce mnist to 64 and 256 dimensions via PCA, and sift 64 dimensions by taking the first 64 features. tKDC is at least 1000x faster than all implementations besides ks on low dimensional datasets ($d < 10$). ks is even faster in two dimensions but its binning efficiency falls off exponentially with dimension. In fact, the library only supports $d \leq 4$, so we were unable to benchmark it on higher dimensional datasets. Furthermore, ks does not provide accuracy guarantees, as seen in Section 4.3. In contrast, the other baselines can provide moderate speedups over the

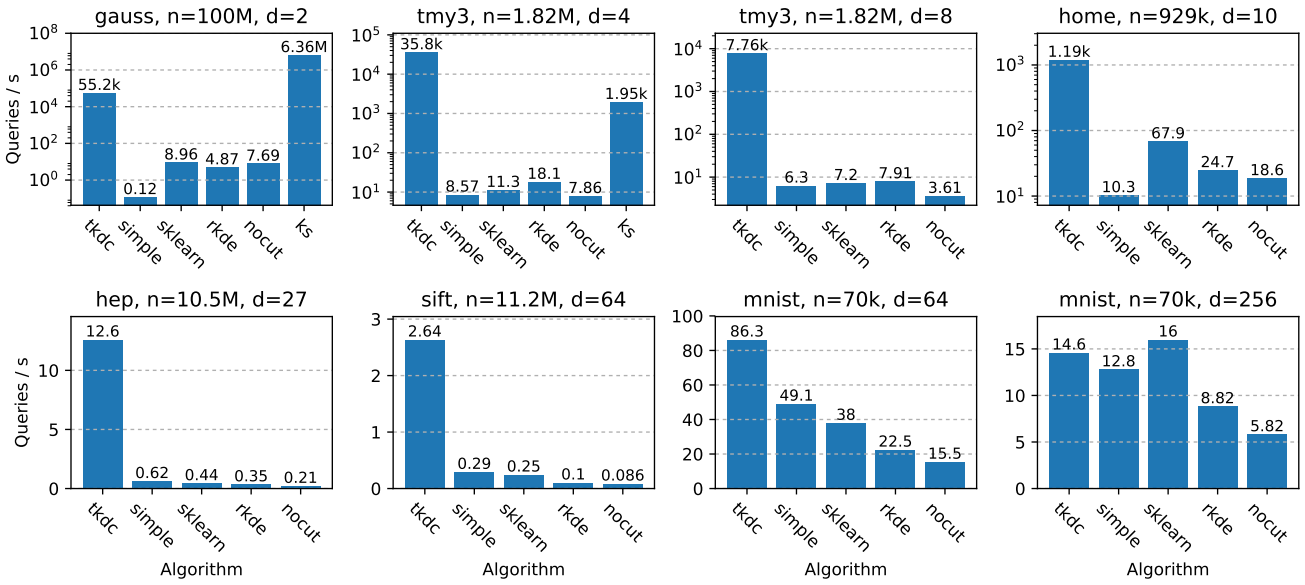


Figure 7: Throughput Comparison: tKDC exhibits significant speedups across a range of datasets and is only outperformed by ks in 2 dimensions. ks is effective in low dimensions but does not scale. tKDC does not perform as well on the 256-dimensional mnist dataset. ks omitted when the dimensionality ($d > 4$) is higher than the library supports.

naïve algorithm, especially in 2-dimensions, but also do not scale on the hep, mnist, and sift datasets.

However, tKDC does not perform as well on the 256-dimensional mnist dataset, and we believe this is because the dataset size is not large enough to allow tKDC to effectively prune query points in such high dimensions. Figure 14 in Appendix B illustrates the behavior for other mnist dimensions; for our target dataset sizes (up to 10M), we have observed that tKDC does not provide meaningful speedups on most datasets with more than 100 dimensions.

4.3 Classification Accuracy

One of the primary benefits of using kernel density estimates is that, at scale, they are guaranteed to converge to the true probability distribution. tKDC allows for some error $\varepsilon t^{(p)}$ in its classifications, so in this section we examine how well tKDC preserves the behavior of calculating an exact kernel density estimate and then classifying points based on their true kernel density. As ground truth, we compute exact kernel densities using scikit-learn on 50k rows of the tmy3 and home datasets, and all 43500 rows of the shuttle dataset. With $p = 0.01$, we classify points based on whether the ground truth density was below $t^{(p)}$. Similarly, we evaluate tKDC, ks, and sklearn by estimating densities for each point in the dataset, estimating $t^{(p)}$ using these densities, and classifying the points accordingly. Since $p = 0.01$, the classification problem identifies points under the threshold. Figure 8 presents the F-1 classification score for each of the algorithms. As expected from using an $\varepsilon = 0.01$ error parameter, tKDC has nearly perfect accuracy, only making incorrect classification for points within εt of the threshold. ks accuracy degrades considerably in 4-dimensions due to its coarse bin size.

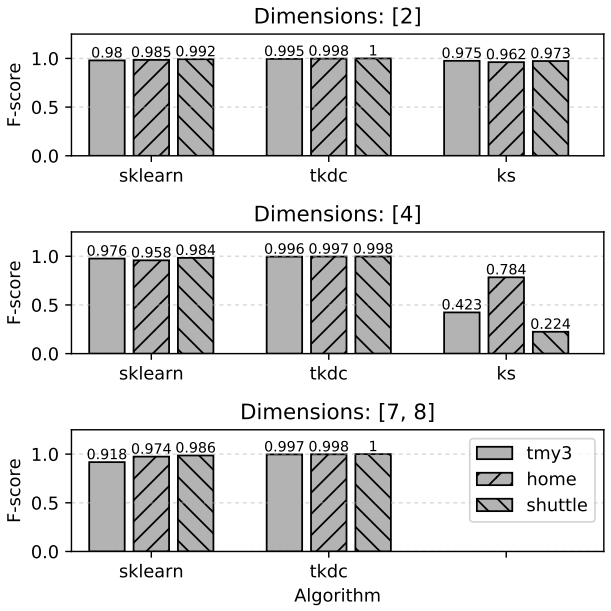


Figure 8: Classification Accuracy Evaluation. tKDC consistently provides high classification accuracy with guaranteed $\varepsilon = 0.01$. Due to its use of bins, ks has consistently worse accuracy that degrades sharply with dimension.

4.4 Scalability

A naïve KDE can produce precise density estimates and has relatively few performance sensitive parameters. However, its major weakness is that its single query runtime increases linearly ($O(n)$) with dataset size: queries that are instantaneous on 10k data points become unwieldy at 100M. Thus in this section we show how tKDC scales well over dataset size, data dimensionality, and configuration settings such as p.

Figure 9 describes throughput (excluding training time) for classifying query points on datasets of different sizes, in this case all subsets of the 2-d gauss dataset. We did not include ks here since its query throughput is independent of the training set size. tKDC achieves asymptotically better throughput as n increases as sug-

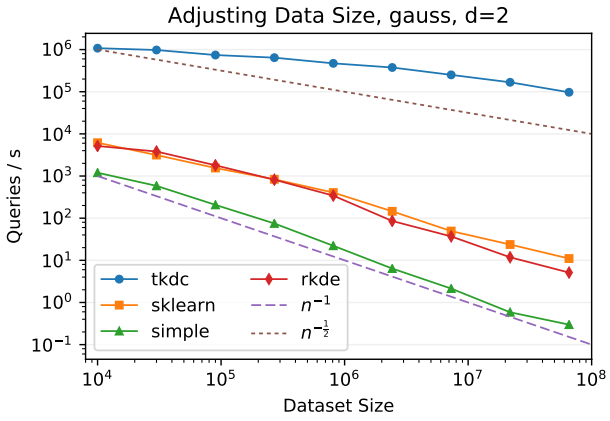


Figure 9: Scalability over dataset size. tKDC maintains its high throughput as n increases, while other algorithms degrade at a much higher rate. Expected runtimes of $O(n^{-0.5})$ and $O(n)$ from Section 3.8 are shown for clarity.

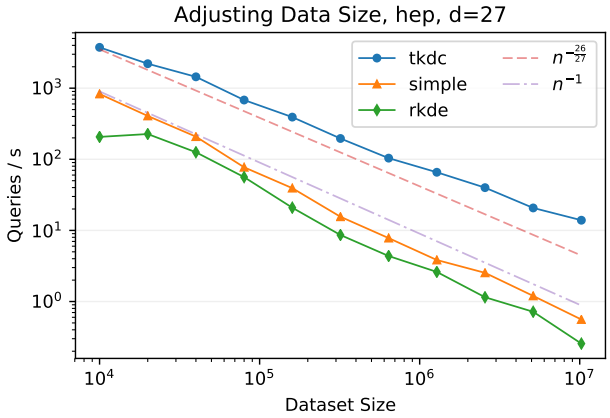


Figure 10: Scalability over dataset size on a higher-dimensional dataset. tKDC remains asymptotically faster than $O(n)$ algorithms, though the difference in 27 dimensions is less pronounced than in $d = 2$.

gested by the $O(n^{\frac{d-1}{d}})$ runtime bound derived in Section 3.8. In fact, the measured throughput exceeds the $O(n^{-\frac{1}{2}})$ bound for $d = 2$. The other algorithms appear to have $O(n^{-1})$ throughput scaling. Figure 10 repeats this experiment on the higher dimensional (27) hep dataset. Since tKDC scales as $O(n^{\frac{d-1}{d}})$ for $d = 27$ the asymptotic speedup is not as dramatic, but tKDC still performs better than our conservative runtime bound would expect and its advantage over naïve algorithms improves as n increases.

Figure 11 describes how tKDC scales with dimensionality for different subsets of the hep dataset. The runtime of the naïve algorithm is nearly independent of dimensionality, but all other approaches benchmarked have worse performance in higher dimensions. tKDC retains at least an order of magnitude of speedup across different dimensions over other algorithms. Figure 14 in Appendix B illustrates the results on the mnist dataset up to 768 dimensions. tKDC is competitive for these dimensions, but does not provide significant speedups past $d > 100$.

In addition to dataset properties, tKDC performance also varies with the quantile threshold parameter p which defines $t^{(p)}$. Figure 15

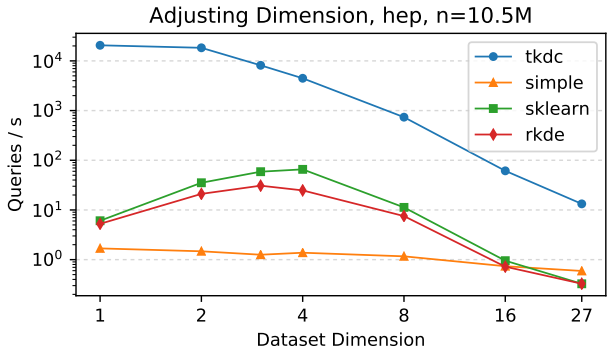


Figure 11: Scalability over data dimensionality. tKDC performance degrades with dimensionality on small datasets, but remains at least an order of magnitude faster than alternative approaches.

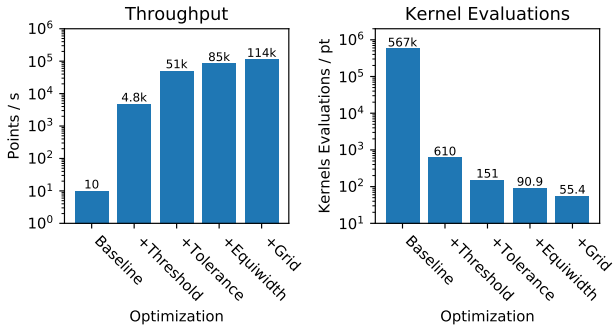


Figure 12: Cumulative Factor Analysis on 500k rows of a 4-d tmy3 dataset. Adding optimizations one a time shows that each optimization brings us closer to our final throughput, and reduces the amount of kernel evaluations necessary for classification.

in Appendix B shows how the performance varies with p: tKDC throughput is highest for very low and very high thresholds with few neighboring points.

4.5 Performance Factor Analysis

A variety of optimizations contribute to the speedups provided by tKDC. To understand these, we can consider the following components of tKDC individually: the tolerance pruning rule (Section 3.3), the threshold pruning rules (Section 3.3), trimmed midpoint tree construction (Section 3.7), and the grid cache (Section 3.7). We will denote these components tolerance, threshold, equiwidth, and grid, respectively.

Figure 12 illustrates the cumulative impact of introducing these optimizations sequentially to a baseline algorithm which traverses the k-d tree and accumulates all individual kernel densities. We measure both throughput and the number of kernel evaluations per point, but exclude training time in this figure. The initial baseline has worse throughput than a simple loop over all datapoints since it incurs the overhead of tree traversal. However, with all optimizations enabled, tKDC can make classifications using on average 55 kernel evaluations for each query, out of 500k possible training data points. The threshold pruning rule is responsible for the bulk of the order-of-magnitude speedups, and each optimization contributes an incremental improvement to the runtime.

A lesion analysis is given in Figure 16 in Appendix B which illustrates the effect of removing each optimization individually

from the complete tKDC implementation: this further shows that no optimization is redundant.

5. RELATED WORK

Classification. Classification is a core topic in fields including statistics, machine learning, and data mining. In particular, the literature contains a wealth of methods for anomaly detection and outlier detection [11, 12], including k-nearest neighbors [43], local outlier factor [8], and DBSCAN [22]. In this paper, we examine classification via kernel density estimation, an *unsupervised* (i.e., label-free) *statistical* method for anomaly detection that can be used to identify data that occur in particular probability regions of a stochastic model; KDE in particular is a non-parametric statistical model in that the model structure is not defined in advance but is instead determined from given data [12].

We focus on KDE for two reasons: first, its non-parametric behavior, and, second, its statistically interpretable outputs, which are commonly used in domain science.

First, since the 1980s, KDE has been the de facto method in statistics to infer a continuous distribution from a set of discrete points [51, 60]; Since KDE is non-parametric, it is able to recover a model without making assumptions about the data. In contrast, parametric unsupervised models such as Gaussian mixture models require the user to manually configure the number of components, a potentially brittle process that can lead to incorrect results [36].

Second, KDE is statistically interpretable: it outputs actual probability densities that are useful in scientific domains including statistical physics and numerical analysis [15]. With probability densities, one can not only make classifications but reason about the likelihood of the classified points. As described in Section 2.1, KDE-based density classification has applications in the visualization of spatial datasets [16, 17, 29], ecology [38, 42], and earth science [5]. Probability density level sets have been used to construct statistical confidence intervals [33] and also perform various forms of outlier detection [4, 19, 49]. In contrast, the outputs of detection and classification methods that are not statistically interpretable (e.g. dbscan, local outlier factor) cannot directly be used in these analyses. Reflecting the popularity of density classification, software packages such as Seaborn [57] and ks [20] implement functionality specifically for visualizing kernel density contours.

Given the utility of this combination of non-parametric behavior (i.e., knob free) and statistical interpretability, we seek to improve KDE's computational overhead, thus improving the performance and scalability of the use cases represented by the above applications and existing packages. For other use cases that do not demand statistical interpretability or have labeled data available, parametric and/or supervised outlier detection techniques may be preferable.

Density Estimation. As a core statistical primitive, density classification is the subject of considerable mathematical analysis [9, 54]. In particular, [10] studied the effectiveness of using kernel density estimates to identify level-sets and quantile level-sets, although this line of work did not improve on the computational complexity of computing these quantities. The task of density classification is also closely related to the support estimation problem in machine learning [48], which can be solved using one-class Support Vector Machines (SVMs). However, one-class SVMs require $O(n^3)$ training time naïvely and $O(n^{2.5})$ using accelerated methods [48]. Thus, training a one-class SVMs is even slower than evaluating KDE, which we study in this work; extending tKDC-style optimizations to one-class SVMs is an interesting opportunity for future work.

KDE is best suited for datasets of modest dimension [35, 51, 55]. In many of our motivating use cases, domain experts (or automated routines) often leverage a relatively small number of dimensions (cf. [3, 21, 59]). High-dimensional datasets suffer from the "curse of dimensionality" [13] where, in high-dimensional spaces, the distinction between nearby and far-away points becomes less pronounced, blurring the distinction between low and high density regions. For high-dimensional datasets (hundreds or thousands of dimensions), we expect users can use tKDC in conjunction with dimensionality reduction methods such as PCA [30] and sketch-based methods such as Locality Sensitive Hashing [2].

Fast Kernel Density Estimation. As a powerful distribution estimator, KDE is the subject of study both in statistics [55] and, recently, databases [60]. To illustrate similarities and differences with tKDC, we divide existing research in fast KDE computation, into two classes: algorithms that rely primarily on data transformations such as FFT and the Fast Gauss Transform, and algorithms that rely primarily on spatially grouping the data.

In the former class, methods based on grids and binning (such as "ks") can take advantage of FFT for very high performance [20, 51, 55]. However, many of these techniques do not provide accuracy guarantees and require building indices that scale exponentially with dimension. Other methods based on the Gauss transform provide better accuracy bounds [45, 58], but can require delicate parameter tuning [37] and also usually also scale exponentially poorly with dimension. [37] tries to address these issues but does not provide consistently better performance than simple tree-based methods in its evaluation.

In the latter class of fast KDE methods, other techniques rely on grouping points into clusters for faster evaluation [31, 60]. In particular, [60] builds an index with guarantees on accuracy: specifically, [60] allows a fixed *additive* error threshold ε as opposed to a threshold or data-point dependent bound. Other efforts leverage k-d and ball trees to derive density bounds [18, 26]. As described in Section 3, tKDC builds directly upon these techniques, whose data structures scale well to larger dimensions and provide good accuracy guarantees. However, existing k-d tree based KDE implementations focus on making density estimates, not classifications, and so are unable to take advantage of the cutoff threshold $t^{(p)}$ that is fundamental to tKDC's performance. In our evaluation, we achieve orders of magnitude speedups compared with these methods. tKDC does not make use of "dual-tree" techniques for grouping both query and training points [26] and integrating these with our pruning rules is a promising direction of future work.

6. CONCLUSION

Density classification is a recurring task in data analytics, and we introduce tKDC, which performs density classification via Kernel Density Estimation. tKDC makes use of pruning rules to classify point probability densities according to a quantile threshold while maintaining accuracy guarantees. This brings the runtime cost of evaluating a single density down to $O(n^{\frac{d-1}{d}})$, allowing tKDC to scale to a variety of dataset sizes and dimensionalities and offer orders of magnitude higher throughput over alternative methods.

Acknowledgements

We thank the many members of the Stanford InfoLab as well as Moses Charikar, John Duchi, and Greg Valiant for their valuable feedback on work. This research was supported in part by Toyota Research Institute, Intel, the Army High Performance Computing Research Center, RWE AG, Visa, Keysight Technologies, Facebook, VMWare, and the NSF Graduate Research Fellowship under grant DGE-114747.

7. REFERENCES

[1] S. Alam, F. D. Albareti, C. A. Prieto, F. Anders, et al. The eleventh and twelfth data releases of the sloan digital sky survey: Final data from sdss-iii. *The Astrophysical Journal Supplement Series*, 219(1):12, 2015.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, 2006.

[3] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. MacroBase: Prioritizing Attention in Fast Data. In *SIGMOD*, 2017.

[4] A. Baillo, A. Cuevas, and A. Justel. Set estimation and nonparametric detection. *Canadian Journal of Statistics*, 28(4):765–782, 2000.

[5] M. S. Bebbington and S. J. Cronin. Spatio-temporal hazard estimation in the auckland volcanic field, new zealand, with a new event-order model. *Bulletin of Volcanology*, 73(1):55–72, 2011.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] M. Blanton. Sdss galaxy map. http://www.sdss.org/science/orangepie/, June 2014.

[8] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *SIGMOD*, 2000.

[9] B. Cadre. Kernel estimation of density level sets. *Journal of multivariate analysis*, 97(4):999–1023, 2006.

[10] B. Cadre, B. Pelletier, and P. Pudlo. Estimation of density level sets with a given probability content. *Journal of Nonparametric Statistics*, 25(1):261–272, 2013.

[11] G. O. Campos, A. Zimek, J. Sander, R. J. G. B. Campello, B. Micenková, E. Schubert, I. Assent, and M. E. Houle. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data Mining and Knowledge Discovery*, 30(4):891–927, 2016.

[12] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[13] L. Chen. Curse of dimensionality. In *Encyclopedia of Database Systems*, pages 545–546. Springer, 2009.

[14] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[15] K. Cranmer. Kernel estimation in high-energy physics. *Computer Physics Communications*, 136(3):198 – 207, 2001.

[16] A. Cuevas, M. Febrero, and R. Fraiman. Estimating the number of clusters. *Canadian Journal of Statistics*, 28(2):367–382, 2000.

[17] A. Cuevas, M. Febrero, and R. Fraiman. Cluster analysis: a further approach based on density estimation. *Computational Statistics & Data Analysis*, 36(4):441 – 459, 2001.

[18] K. Deng and A. W. Moore. Multiresolution instance-based learning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1233–1239, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[19] L. Devroye and G. L. Wise. Detection of abnormal behavior via nonparametric estimation of the support. *SIAM Journal on Applied Mathematics*, 38(3):480–488, 1980.

[20] T. Duong et al. ks: Kernel density estimation and kernel discriminant analysis for multivariate data in r. *Journal of Statistical Software*, 21(i07), 2007.

[21] J. G. Dy and C. E. Brodley. Feature selection for unsupervised learning. *Journal of machine learning research*, 5(Aug):845–889, 2004.

[22] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD-96 Proceedings*, 1996.

[23] Ferdosi, B. J., Buddelmeijer, H., Trager, S. C., Wilkinson, M. H. F., and Roerdink, J. B. T. M. Comparison of density estimation methods for astronomical datasets. *A & A*, 531:A114, 2011.

[24] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[25] J. D. Gibbons and S. Chakraborti. *Nonparametric statistical inference*. Springer, 2011.

[26] A. G. Gray and A. W. Moore. Nonparametric density estimation: Toward computational tractability. In *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*, pages 203–211, 2003.

[27] P. Hall and M. Wand. On the accuracy of binned kernel density estimators. *Journal of Multivariate Analysis*, 56(2):165 – 184, 1996.

[28] R. Huerta, T. Mosqueiro, J. Fonollosa, N. Rulkov, and I. Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 2016.

[29] R. J. Hyndman. Computing and graphing highest density regions. *The American Statistician*, 50(2):120–126, 1996.

[30] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[31] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[33] J. Lei. Classification with confidence. *Biometrika*, 2014.

[34] M. Lichman. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2013.

[35] H. Liu, J. D. Lafferty, and L. A. Wasserman. Sparse nonparametric density estimation in high dimensions using the rodeo. In *AISTATS*, 2007.

[36] G. J. McLachlan and S. Rathnayake. On the number of components in a gaussian mixture model. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(5):341–355, 2014.

[37] V. I. Morariu, B. V. Srinivasan, V. C. Raykar, R. Duraiswami, and L. S. Davis. Automatic online tuning for fast gaussian summation. In *NIPS*, 2009.

[38] S. D. Newsome, J. D. Yeakel, P. V. Wheatley, and M. T. Tinker. Tools for quantifying isotopic niche space and dietary variation at the individual and population level. *Journal of Mammalogy*, 93(2):329–341, 2012.

[39] O. of Energy Efficiency & Renewable Energy (EERE). Commercial and residential hourly load profiles for all tmy3 locations in the united states.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[41] A. Perrot, R. Bourqui, N. Hanusse, F. Lalanne, and D. Auber. Large interactive visualization of density functions on big data infrastructure. In *LDAV*, 2015.

[42] L. T. Quakenbush, J. J. Citta, et al. Fall and winter movements of bowhead whales (balaena mysticetus) in the chukchi sea and within a potential petroleum development area. *Arctic*, 63(3):289–307, 2010.

[43] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD*, 2000.

[44] V. C. Raykar and R. Duraiswami. Fast optimal bandwidth selection for kernel density estimation. In *SDM*, 2006.

[45] V. C. Raykar, R. Duraiswami, and L. H. Zhao. Fast computation of kernel estimators. *Journal of Computational and Graphical Statistics*, 19(1):205–220, 2010.

[46] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. *Ann. Math. Statist.*, 27(3):832–837, 09 1956.

[47] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[48] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[49] E. Schubert, A. Zimek, and H.-P. Kriegel. *Generalized Outlier Detection with Flexible Kernel Density Estimates*, pages 542–550.

[50] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley Series in Probability and Statistics. Wiley, 2009.

[51] B. W. Silverman. Algorithm as 176: Kernel density estimation using the fast fourier transform. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(1):93–99, 1982.

[52] B. W. Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[53] D. Stoneking. Improving the manufacturability of electronic designs. *IEEE Spectrum*, 36(6):70–76, 1999.

[54] A. B. Tsybakov et al. On nonparametric estimation of density level sets. *The Annals of Statistics*, 25(3):948–969, 1997.

[55] M. Wand. Fast computation of multivariate kernel estimators. *Journal of Computational and Graphical Statistics*, 3(4):433–445, 1994.

[56] M. Wand and M. Jones. *Kernel Smoothing*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1994.

[57] M. Waskom, O. Botvinnik, drewokane, P. Hobson, et al. seaborn: v0.7.1 (june 2016), June 2016.

[58] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast gauss transform and efficient kernel density estimation. In *ICCV*, 2003.

[59] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[60] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li. Quality and efficiency for kernel density estimates in large data. In *SIGMOD*, 2013.

APPENDIX

A. RUNTIME ANALYSIS

In this section, we provide a formal exposition of the runtime analysis given in Section 3.8. Recall that we have a training dataset $X \in \mathbb{R}^{n \times d}$ with n points and d dimensions, sampled from an underlying distribution D. Here, we consider a single threshold $t = t_u = t_l$

and a tolerance of $\varepsilon = 0$, with no grid optimizations. This corresponds to tKDC with only the cutoff rule enabled. Even with only the cutoff rule—which is responsible for most of our speedup—tKDC has asymptotically improved performance.

We will start by establishing Lemma 1:

Lemma. The probability of a query point x falling inside R_X^{near} is proportional to $O\left(n^{-\frac{1}{d}}\right)$

Proof. We show how the precision provided by the k-d tree index density bounds improves (expanding the far region) as we add more points to the training set X. Note that tKDC may or may not make full use of the index before it selectively evaluates leaf nodes that may have a bigger impact on improving the upper and lower bounds, however this makes tKDC more efficient than an algorithm which strictly evaluates all index nodes before resolving any individual point contributions. By bounding the behavior of this more strict algorithm, we can conservatively bound the runtime of tKDC.

Let I_n denote the index on a training set of n points. If we double the number of points, each leaf in I_{2n} will become a parent node with two children, split along the trimmed midpoint in one dimension. After we double d times, each leaf in $I_{2^d n}$ spans half the range as its corresponding parent leaf in I_n along each dimension. By Taylor's theorem, for large n we can show that the precision Δ_n provided by these kernel density bounding box estimates for I_n is proportional to the maximum width w of the boxes [27]. Thus $\Delta_{2^d n} \approx \frac{1}{2}\Delta_n$ so $\Delta_n = O(n^{-1/d})$

Any query point x with density $p(x)$ far enough from the threshold t can be classified using only the index and is thus a "far" point in our previous nomenclature. More precisely, when $|p(x) - t| > \Delta_n$ then an index I_n is sufficiently precise to classify the point without traversing leaf nodes. Thus, the "near" region is $R_{X_n}^{near} = \{x : |p(x) - t| \leq \Delta_n\}$.

Now, let q be the *cumulative distribution function of the densities* $p(x)$ for $x \sim X$, i.e. $q(y) = \Pr[p(x) < y]$. Then, by Taylor's theorem as n grows and Δ_n shrinks, the derivative $q'(y)$ gives us a measure of how many points x have densities close to $p(x)$, where $2q'(t)dt \approx \Pr[t - dt < p(x) < t + dt]$. Letting $dt = \Delta_n$ we then have:

$$\Pr[x \in R_{X_n}^{near}] \approx 2q'(t)\Delta_n = O\left(q'(t)n^{\frac{1}{d}}\right)$$

□

Now we have proven Lemma 1, and we can solve a more precise version of the recurrence in Section 3.8.

$$F_{2n} \leq F_n + O\left(q'(t)n^{\frac{d-1}{d}}\right)$$

When $\frac{d-1}{d} > \log_2(1)$, we can use case 3 of the master theorem [14] to show that $F_n = O(q'(t)n^{\frac{d-1}{d}})$ when $d > 1$. Otherwise, when $d = 1$ and $\frac{d-1}{d} = 0$, we can use case 2 of the master theorem to show that $F_n = O(q'(t)\log(n))$.

Note that this more precise runtime expression (which encodes not just the dependence on n but also on t) shows that the runtime is proportional to $q'(t)$ the density of points near the threshold t, so we can compare with Figure 15 to see how the throughput decreases for larger thresholds with more "nearby" points than small tail thresholds.

B. ADDITIONAL EVALUATION

Figure 13 illustrates how the performance of the rkde algorithms depends on the radius threshold of nearby points considered. A smaller thresholds means more points can be pruned out from consideration when performing a range query, but also means that the resulting density estimate will be more inaccurate. In the plot, the radius is the distance threshold for pruning far-away points after scaling by the bandwidth, and in this test for $r \leq 1.2$ the error in the densities is on the order of the threshold t, so the results are highly unreliable for small r. In any case, rkde is unable to match tKDC's throughput while preserving any accuracy.

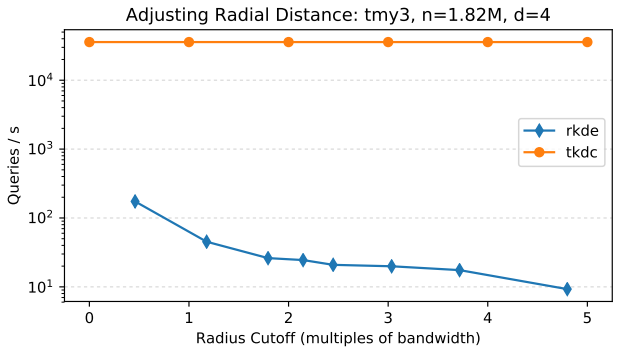


Figure 13: Scalability with radius threshold for rkde algorithm. Smaller radiuses allow for better performance at the cost of worse accuracy, but is still orders of magnitude slower than tKDC.

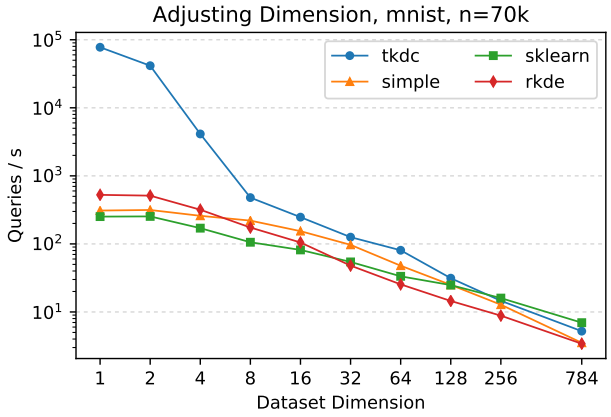


Figure 14: Scalability over data dimensionality on the mnist dataset. tKDC does not perform as well on small, high dimensional datasets, but remains competitive with other approaches.

Figure 14 presents an additional benchmark evaluating tKDC's performance on higher dimensional data, in this case the mnist dataset with up to 768 dimensions. For $d \leq 256$ we used a PCA to reduce the dimensionality since many of the pixels in mnist are almost always 0, while for $d = 768$ the native dimension we use the raw dataset. For $d \leq 256$ we also scale the bandwidth by $3\times$ the standard Scott's rule bandwidth to ameliorate underflow issues in this dataset, and for $d = 768$ use a bandwidth of $b = 1000$. For this relatively small $n = 70k$ dataset, tKDC scales relatively poorly with dimension since it's asymptotic advantage with n does not have a chance to kick in in higher dimensions, however it never degrades to the point where it is worse than a naïve computation.

Figure 15 illustrates how the performance of tKDC running with $\varepsilon = 0.1$ degrades for higher p, but remains better than sklearn, ks, and other baseline approaches. The pruning rules are more effective when there are relatively few query points near the threshold at very low and very high values. The relationship here is made more explicit in the runtime analysis in Appendix A, where we show that

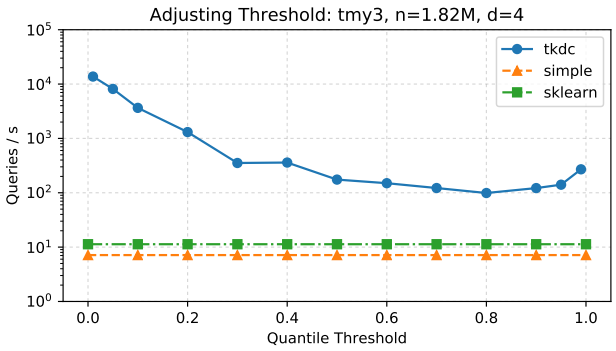


Figure 15: Throughput for different quantile boundaries: Performance is best for very low and very high thresholds, but remains an order of magnitude faster than sklearn and naïve methods which do not depend on p.

the runtime is proportional to the relative density of points near the threshold.

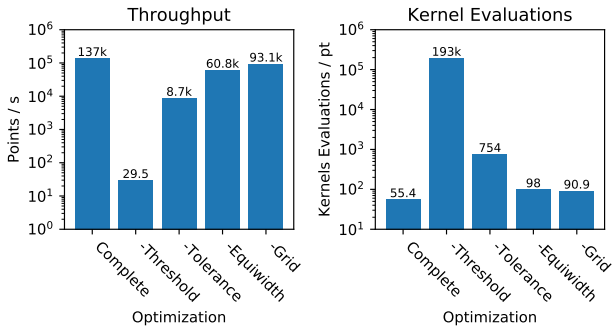


Figure 16: Lesion Analysis on 500k rows of a 4-d tmy3 dataset. Removing a single optimization at a time shows that no optimization is redundant.

Figure 16 shows the effect of removing each of our optimizations individually from the complete tKDC implementation. Compared to the complete suite, removing each optimization has an impact on the throughput, illustrating the contribution of each. Removing the threshold pruning rule in particular erases nearly all of the gains: it is the foundation of the performance improvements in tKDC.