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.

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 naively, the total runtime cost of density estima-
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 calculation. 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 expensive. To address this, we develop a sampling-based algorithm for accurately estimating a quantile target threshold (i.e., one useful for classification). tKDC uses quantile target thresholds to prune kernel evaluations. This additional pruning rule asymptotically reduces the complexity of computing the density of a single point from \(O(n)\) to \(O(n^{\frac{d+1}{d}})\).

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
with possibly widely varying magnitudes, precision is defined relative to 
approximate density classification. Our algorithm, tKDC, solves
unacceptably coarse for small thresholds
our goal is to make classifications for different
precise borderline decisions while bounding the severity of possible
algorithm to avoid the expensive computations required to make
translating bounded density regions into classification confidence
Identifying points in low and high density regions is key to en-
abling 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 distri-
bution 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 imple-
mentations [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
(\( \pm \epsilon \) 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 \( \epsilon \) 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 rel-
ative to \( t \). An absolute additive precision of \( \epsilon_{\text{abs}} = 0.01 \) would be unacceptably coarse for small thresholds \( t < \epsilon \).

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:

\[
\begin{align*}
\text{HIGH} \quad &\text{when } f(x_q) > t \cdot (1 + \epsilon) \\
\text{LOW} \quad &\text{when } f(x_q) < t \cdot (1 - \epsilon)
\end{align*}
\]

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 sub-
stantially, 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 \( \{ f(x) - f_0 : x \in X \} \):

\[
t^{(p)} := q_p(\{ f(x) - f_0 : x \in X \})
\]

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 speci-
cified, 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 prob-
abilistically estimating \( t^{(p)} \) in Section 3.5. The quantile threshold
estimation algorithm relies on sampling and thus has an adjustable
failure probability \( \delta \), but our density classification algorithm is
otherwise deterministic.

2.4 Kernel Density Estimation

Having defined density classification and the thresholds \( t^{(p)} \), we now
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 \( MSE \sim O(n^{- \frac{2}{d+2}}) \). 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 of
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-
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 \( u(p), l(p) \) 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 \( \epsilon \) 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 \( t(p) \) for \( t(p) \) by calculating the \( p \)-quantile \( t(p) \) 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 \( t(p) \).

In the following sections, we start by assuming that initial coarse bounds \( l(p), u(p) \) 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.
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

\[ f_{t}^{(p)}, t_{t}^{(p)} \leftarrow \text{BOUNDTHRESHOLD}(X) \]
\[ T \leftarrow \text{MakeIndex}(X) \quad \triangleright \text{Construct Spatial Index} \]
\[ D_{t} \leftarrow [\ldots] \quad \triangleright \text{Density estimates for } x_{t} \in X \]
for \( x_{t} \in X \) do
\[ f_{t}, f_{a} \leftarrow \text{BOUNDENSITY}(T, f_{t}^{(p)}, t_{t}^{(p)}, x_{t}) \]
\[ \text{append}(D_{t}, (f_{t} + f_{a})/2 - \frac{1}{N}K_{H}(0)) \]
\[ \bar{r}^{(p)} \leftarrow q_{p}(D_{t}) \quad \triangleright \text{Approximate threshold} \]
function \( \text{CLASSIFY}(x_{q}) \)
\[ f_{t}, f_{a} \leftarrow \text{BOUNDENSITY}(T, \bar{r}^{(p)}, \bar{r}^{(p)}, x_{q}) \]
if \( (f_{t} + f_{a})/2 > \bar{r}^{(p)} \) then
\[ \text{return HIGH} \]
else
\[ \text{return LOW} \]

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.

Algorithm 3: tKDC: Approximate Density Classification

- **Initialization**: \( f_{t}^{(p)}, t_{t}^{(p)} \leftarrow \text{BOUNDTHRESHOLD}(X) \)
- **Spatial Index Construction**: \( T \leftarrow \text{MakeIndex}(X) \)
- **Density Estimation**: \( D_{t} \leftarrow [\ldots] \)
- **Iteration**: for each \( x_{t} \in X \) do
  - \( f_{t}, f_{a} \leftarrow \text{BOUNDENSITY}(T, f_{t}^{(p)}, t_{t}^{(p)}, x_{t}) \)
  - \( \text{append}(D_{t}, (f_{t} + f_{a})/2 - \frac{1}{N}K_{H}(0)) \)
  - \( \bar{r}^{(p)} \leftarrow q_{p}(D_{t}) \)
- **Classification**: function \( \text{CLASSIFY}(x_{q}) \)
  - \( f_{t}, f_{a} \leftarrow \text{BOUNDENSITY}(T, \bar{r}^{(p)}, \bar{r}^{(p)}, x_{q}) \)
  - if \( (f_{t} + f_{a})/2 > \bar{r}^{(p)} \) then
    - return HIGH
  - else
    - return LOW

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 \( x_{min}^{l}, x_{max}^{l} \) for the points under a node and for each coordinate axis \( l \). 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 \( X_{q} \), the total kernel density contribution \( f^{(r)}(x_{q}) \) is given as:

\[ f^{(r)}(x_{q}) = \frac{1}{n} \sum_{x_{i} \in X_{q}} K_{H}(x_{q} - x_{i}) \quad (5) \]

\( f^{(r)}(x_{q}) \): is bounded by the smallest and largest distance vectors \( d_{max}, d_{min} \) from \( x_{q} \) to the bounding box of \( X_{q} \):

\[ \frac{|X_{q}|}{n} K_{H}(d_{max}) \leq f^{(r)}(x_{q}) \leq \frac{|X_{q}|}{n} K_{H}(d_{min}) \quad (6) \]

3.2 Iterative Refinement

As seen in the previous section, each node in the k-d tree has a 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 \( x_{q}^{(j)} \in X \) would be assigned to one of the child nodes depending on whether \( x_{q}^{(j)} < 3 \). 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 \( x_{min}, x_{max} \) for the points under a node and for each coordinate axis \( l \). 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 \( X_{q} \), the total kernel density contribution \( f^{(r)}(x_{q}) \) is given as:

\[ f^{(r)}(x_{q}) = \frac{1}{n} \sum_{x_{i} \in X_{q}} K_{H}(x_{q} - x_{i}) \quad (5) \]

\( f^{(r)}(x_{q}) \): is bounded by the smallest and largest distance vectors \( d_{max}, d_{min} \) from \( x_{q} \) to the bounding box of \( X_{q} \):

\[ \frac{|X_{q}|}{n} K_{H}(d_{max}) \leq f^{(r)}(x_{q}) \leq \frac{|X_{q}|}{n} K_{H}(d_{min}) \quad (6) \]

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.

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 \( K(d_{min}) \). 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 \( \{X_{q}\} \) with bounding boxes \( \{B_{i}\} \), we can bound the kernel density estimate \( f(x_{q}) \) with:

\[ f_{l} = \frac{1}{N} \sum_{i} \left| X_{i} \right| K_{H}(d_{max}(x_{q}, B_{i})) \]
\[ f_{h} = \frac{1}{N} \sum_{i} \left| X_{i} \right| K_{H}(d_{min}(x_{q}, B_{i})) \]

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:
Tolerance Rule

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 $\epsilon$ 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:

$$f_u^{(i)}(x_q) - f_l^{(i)}(x_q) < \epsilon t_{(p)}$$

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:

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

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 $t^{(p)} = \pm \epsilon t^{(p)}$.

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 $\pm \epsilon t^{(p)}$ 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 $n_e (K_l(d_{min}) - K_l(d_{max}))$ where $n_e$ 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 BoundingDensity 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 $\pm \epsilon t^{(p)}$ otherwise.

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 run after it hit the tolerance rule, so $f_u - f_l < \epsilon t^{(p)}$ and $\frac{1}{2} (f_l(x_q) + f_u(x_q))$ will be within $\pm \epsilon t^{(p)}$ 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 $\pm \epsilon t^{(p)}$ 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 $f_u - f_l < \epsilon t^{(p)}$. Thus, calculating quantities on $\{\frac{1}{2} (f_l(x) + f_u(x)) : x \in X\}$ allows us to estimate $t^{(p)}$ to within $\pm \epsilon t^{(p)}$ as well.

Thus, the BoundDensity procedure allows us to obtain bounds on both $f(x_q)$ and $t^{(p)}$ accurate to $\pm \epsilon t^{(p)}$ 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 $d^{(i)}_s$ be the $i$-th order static of $D_s$. Then, the binomial theorem gives us

```
Algorithm 2 Approximate Density Estimation

function BOUNDENSITY(T, t_{l}^{(p)}, t_{u}^{(p)})
  pq ← [T]  // Node Priority Queue
  f_{u} ← w_{min}(x, T)
  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 $f_{u} - f_{l} < \epsilon \cdot t_{l}^{(p)}$ then  // Tolerance Rule
      break
    if node is leaf then
      f_{cur} ← poll(pq)
      f_{l} ← f_{l} - w_{min}(child)
      f_{u} ← f_{u} + w_{max}(child)
      pq ← add(pq, child)
    else
      for child in children(node) do
        f_{l} ← f_{l} + w_{max}(child)
        f_{u} ← f_{u} + w_{min}(child)
      end
    end
  end
  return $f_{l}, f_{u}$
```

Equation 10 [25].
\[
Pr \left( d_x^{(p)} \leq d^{(np)} \leq d_x^{(u)} \right) = \sum_{i=1}^{n} \binom{n}{i} p^i (1 - p)^{n-i}
\]

For large \( n \) the binomial bound is well approximated by a normal distribution, so we can simplify the above equation:
\[
Pr \left( d_x^{(sp - z_\delta \sqrt{sp(1-p)}/2)} \leq d^{(np)} \leq d_x^{(sp + z_\delta \sqrt{sp(1-p)}/2)} \right) \geq 1 - \delta
\]

(11)

where the constant \( z_\delta \) is the \( \delta \)-th quantile of the normal distribution. For an acceptable failure rate \( \delta \), 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 \( \delta \) dictates how large 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 \( d^{(p)} \), then \( z_{0.99} = 2.576 \) so we have: \( Pr(\hat{d}(164) \leq t^{(0.01)} \leq \hat{d}(236)) \geq 0.99 \) and the 164th and 236th densities provide a confidence interval for \( \hat{d}(p) \).

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 \( \hat{d}(p) \), but to obtain bounds on \( \hat{d}(p) \) 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 estimators 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 \( X_s \subseteq X \) we can compute kernel densities \( f_{X_s} \) using data just from this subset.

\[
f_{X_s}(x_q) = \frac{1}{N_s} \sum_{x \in X_s} K_H (x_q - x)
\]

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

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

For example, if we have bounds \( t^{(p)}_l, t^{(p)}_u \) calculated from a KDE trained on \( X_s \), then we can use these bounds when calculating densities for a KDE trained on \( X_{4s} \), a subsample 4 times the size of \( X_s \). The BoundDensity routine returns density bounds that have precision \( \epsilon t^{(p)}_l \) for densities within the threshold bounds, so as long as enough of the new densities remain within the \( t^{(p)}_l, t^{(p)}_u \) 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 \( X_s \) (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 \( d^{(p)}, d^{(u)} \) the 1 - \( \delta \) confidence bounds for \( \hat{d}(p) \) to within \( \epsilon \cdot \hat{d}(p) \). 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 \( h_{\text{backoff}}, h_{\text{buffer}} \) 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 \( h_{\text{growth}} \) do not affect correctness.

### 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 routine 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 \( f_l > t^{(p)}_u \) or \( f_u < t^{(p)}_l \) and we can precisely classify a point \( x \) or else \( f(x) \) is near the threshold and \( f_u - f_l < t^{(p)}_u \).

Thus, assuming that \( t^{(p)}_u, t^{(p)}_l \) are valid bounds for \( \hat{d}(p) \), then the \( p \)-quantile of the densities \( d_s, q_p(D_s) \), is an estimate \( \bar{d}(p) \) for \( \hat{d}(p) \) that is accurate to within \( \epsilon \cdot \bar{d}(p) \). Ignoring constant factors of \( \epsilon \cdot \bar{d}(p) \), this means that the Classify routine correctly classifies all points with densities more than \( \epsilon \cdot \bar{d}(p) \) away from \( \bar{d}(p) \), and solves the density classification problem (Problem 1) for \( \hat{d}(p) \).

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

---

**Algorithm 3 Bootstrapped Threshold Bound**

```python
function BoundThreshold(X)
    t_l^{(p)} ← 0  # Threshold bounds
    t_u^{(p)} ← ∞  # Num training points
    r ← r_0  # Num query points
    s ← s_0

    while r ≤ N do
        X_r ← sample(X, r)
        X_s ← sample(X_r, s)
        Build kdtree on X_r
        Recalculate bandwidth
        \{f_l^{(p)}, f_u^{(p)}\} ← BoundDensity(t_l^{(p)}, t_u^{(p)}, X_s)
        \{d_l^{(p)}, d_u^{(p)}\} ← sorted(\{(f_l^{(p)} + f_u^{(p)})/2, \frac{1}{3\sqrt{N}}KH(0)\})
        # Density estimates, correcting for self-contribution
        l ← sp - z_{(1-\delta)} \sqrt{sp(1-\delta)}/2
        u ← sp + z_{(1-\delta)} \sqrt{sp(1-\delta)}/2
        if d_u > t_u^{(p)} then
            t_u^{(p)} ← t_u^{(p)} · h_{\text{backoff}}
            else if d(l) < t_l^{(p)} then
                t_l^{(p)} ← t_l^{(p)} · h_{\text{buffer}}
                else
                    t_l^{(p)} ← d(l) · h_{\text{buffer}}
                    t_u^{(p)} ← d(u) · h_{\text{buffer}}
                    r ← max(r · h_{\text{growth}} · N)
            return \{d_l^{(p)}, d_u^{(p)}\}
```
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 \( t_h \) 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 dataset 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 \( G(x_q)/N \cdot Kt(d_{diag}) \) 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 it 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 \( i \)-th 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 \( X \subseteq \mathbb{R}^{n \times d} \) 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 \( \epsilon \) and \( \delta \)) in this analysis.

**Theorem 1.** For a query point \( x \) drawn from \( D \), tKDC runs in expected \( O(n^{d/2}) \) 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 naive 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 \( (f_1, f_\infty) \) 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.

**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 \( R^\text{far} \subseteq \mathbb{R}^d \). 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 \( R^\text{far} \). 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 \( R^\text{near} \) is proportional to \( O\left(n^{-\frac{1}{2}}\right)\).

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 \( R^\text{near}_n \), \( R^\text{near}_{2n} \). 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 \( O(n^{-\frac{1}{2}}) \) 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 \( F^\text{near}_n \) be the costs of evaluating far and near points respectively for training set \( X_n \), then we can derive the recurrence:

\[
F_{2n} \leq F^\text{far}_{2n} + \Pr( x \in R^\text{near}_{2n} ) \cdot F^\text{near}_{2n} \\
\leq F_n + O\left(n^{-\frac{1}{2}}\right) \cdot O(2n) \\
\leq F_n + O\left(n^{-\frac{2}{2}}\right)
\]

By the master theorem [14], the runtime is then \( O(n^{d/2}) \) for \( d > 1 \) and \( O(\log(n)) \) for \( d = 1 \).
while the Kernel Smoothing \( \text{ks} \) R package \([20]\) is written in C and

Alternative Algorithms.

Thus, we measure throughput by amortizing the training time across

t the performance should be even better.

throughput for performing tasks such as outlier detection using

tKDC on a dataset by constructing a k-d tree and then estimating

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

Thus, we implement an algorithm that performs a range query

around the query point using some 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 \( \varepsilon = 0.01r \) 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 \( \text{ks} \)

on low dimensional datasets \( d < 10 \). \( \text{ks} \) 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,
\( \text{ks} \) does not provide accuracy guarantees, as seen in Section 4.3.
In contrast, the other baselines can provide moderate speedups over the

Table 2: Algorithms used in evaluation

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

Table 3: Datasets used in evaluation

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

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 \( f(x) \) 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.

\(^1\)Source code available at https://github.com/stanford-futuredata/tkdc

4. EV ALUATION

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.
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 \( e_t(p) \) 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 \( e = 0.01 \) error parameter, tKDC has nearly perfect accuracy, only making incorrect classification for points within \( e_t \) of the threshold. ks accuracy degrades considerably in 4-dimensions 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 \(O(n^{-d/2})\) runtime bound derived in Section 3.8. In fact, the measured throughput exceeds the \(O(n^{-1})\) 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 \(O(n^{-d/2})\) 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 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
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 $O(n^{1/3})$, 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


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 $R^\text{near}_x$ is proportional to $O\left(n^{-\frac{d}{2}}\right)$

**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 $\Delta_n$ 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 $R^\text{near}_x = \{x : |p(x) - t| \leq \Delta_n\}$.

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 $\Delta_n$ 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:

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

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

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

When $\frac{d-1}{d} > \log_2(1)$, we can use case 3 of the master theorem [14] to show that $F_n = O(q'(t)n^{\frac{d-1}{2}})$ 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 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 $d = 768$ we 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.