Applying Dual Tree KDE to Gene Interaction Network Integration

Melroy Saldanha
Chester Shiu
CS229

Abstract

In high dimensional spaces a naïve approach to calculating joint probability densities is computationally intractable (O(MN) where M = # training points, N = # query points). We have implemented a variant of a nonparametric machine learning algorithm, tree-based Kernel Density Estimation, to solve this problem. We will refer to this implementation as dual-tree. We have found our implementation to be very efficient over large datasets (O(sqrt(MN)), but it does not scale as well in high dimensions. While dual-tree would be quite useful for computational statisticians and data miners, our particular motivation comes from a problem in bioinformatics, merging protein interaction networks.

Introduction

A number of experimental techniques have generated evidence about gene-gene and gene-protein interactions. This data has been used to create hypothetical interaction networks that are of great biological interest. However, the networks tend to be derived from a single type of data source and do not uniformly cover the same sets of genes and/or proteins. It would be highly useful if there was a robust algorithm to integrate the various networks into one coherent network, complete with the probability of each potential interaction. Serafim Batzoglou's lab has developed such an algorithm[1], but it is limited by its ability to calculate joint probability densities over large datasets with large dimensions. Thus, the application of dual-tree will address the algorithm's weakest point. The resulting upgraded algorithm will represent a major advancement in the field of protein network integration (particularly in accuracy and scalability).

The most naïve approach to kernel density estimation is to sum up the prior probability of each query point with every data point in the training set. For M queries and N training examples this yields O(MN), which is too slow for most interesting problems. One commonly used approximation is to divide the hypothesis space into bins, treat the center of each bin as a point, and multiply the resulting answer by the number of points in that bin. Unfortunately this is a poor approximation for large N.

Approach

A much better implementation is dual-tree[2]. We can speed things up considerably without loosing too much accuracy if we take into account the structure of the data. First, we assume that each point is fitted with its own Gaussian with its own parameters (hence the non-parametric aspect of the

algorithm – the final distribution across the entire set could look like anything). Next, we observe that Gaussians decay exponentially, so only points close to the query point will make non-infinitesimal contributions to the prior probability sums. Thus, with some (calculable and bounded) error ε we can surround each query point with a bounding box and consider only those training points in the bounding box. To efficiently implement this bounding box, we can keep track of the lower extreme and upper extreme vertex of the bounding box for each node in the mrkd-tree as one of its sufficient statistics[3,4]. If we know all of the query points before runtime, we can achieve even greater speedup by fitting the query points to another mrkd-tree. The comparison would then be between the nodes on each tree instead of a query point to a node. In essence, dual-tree is simply building these two trees and doing a iterative deepening traversal of each tree. This takes O(sqrt(MN) time.

Building a kd-tree. Suppose we are given N points in R^k. To organize them we will build a kd-tree. There are many ways to do this; this is the procedure we used:

(a) Sort all N points along each of the k dimensions to build an auxiliary N by k matrix of sorted indices. This takes kO(N log(N)) using quicksort.

(b) For each of the k dimensions, calculate the range $R_i = x_{i,max} - x_{i,min}$, where i ranges over each dimension from 1 to k.

(c) Partition along the dimension with the greatest range. Partition this dimension at the median, keeping half in one (N/2 by k) submatrix and half in another submatrix of equal size.

(d) Repeat this for each submatrix until we have at most N_{min} points in each submatrix, where $N_{min} \geq 1$ and is the minimum particle count for each node of the kd-tree.

Caching sufficient statistics. We will cache the mean μ; the covariance matrix Σ; and the coordinates of the bounding box $(\vec{l}, \vec{u})$ of the points contained in that node/leaf, where $\vec{l}$ is the point in the box closest to $(-\infty, \ldots, -\infty)$ and $\vec{u}$ is the point in the box closest to $(\infty, \ldots, \infty)$. After building the initial kd-tree we can do post-traversal on it and calculate each parent's statistics in terms of its children.

(a) To calculate μ for a parent with two children that contain N1 and N2 of their own children:

$$\mu = \frac{N_1 \mu_1 + N_2 \mu_2}{N_1 + N_2}$$

(b) Σ can be calculated similarly.

(c) The new bounding box can be found by

$(\vec{l},\vec{u})$ = (min($\vec{l}_{child1},\vec{l}_{child2}$),max($\vec{u}_{child1},\vec{u}_{child2}$))

where min and max are along each dimension.

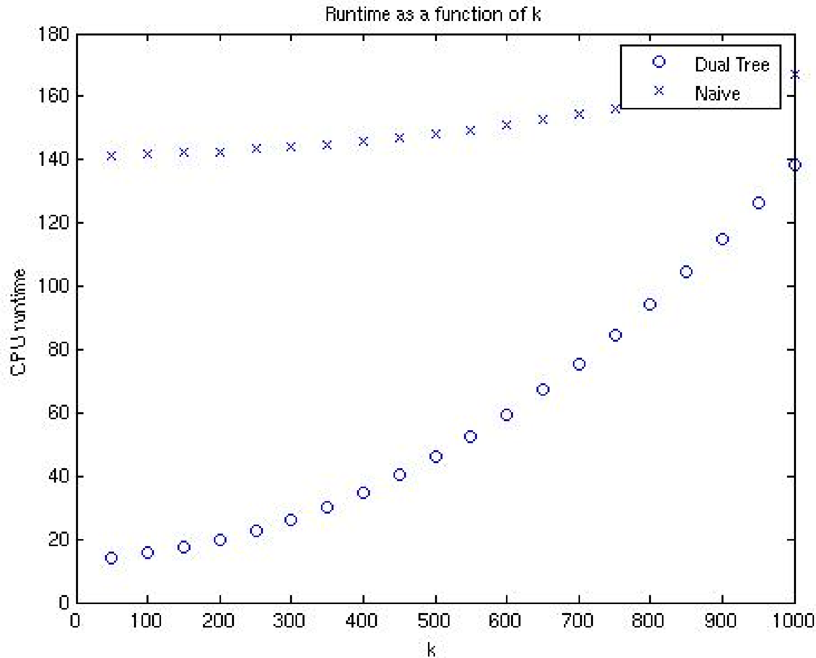
Bounding box comparison. To determine whether a training node makes a significant contribution to a query node, we must determine the shortest distance between their respective bounding boxes. The key observation here is that each dimension can be minimized interpedently, so for $\vec{l}_1,\vec{l}_2,\vec{u}_1,\vec{u}_2$ to find the shortest distance in dimension x, we find min($\left|\vec{l}_{1x}-\vec{l}_{2x}\right|,\left|\vec{l}_{1x}-\vec{u}_{2x}\right|,\left|\vec{u}_{1x}-\vec{l}_{2x}\right|,\left|\vec{u}_{1x}-\vec{u}_{2x}\right|$). Hence, to find the minimum distance between two bounding boxes can be done in O(k) time, where k is the dimensionality of the data.

Estimate maximum contribution. Once we know the minimum distance between the bounding boxes of two nodes, we can calculate the maximum contribution by applying the Gaussian kernel to that minimum distance and multiplying by the number of training points in the box. In other words, the maximum possible contribution of those training points to the density occurs if all the training points were crowded in one area as close as possible to the query points.

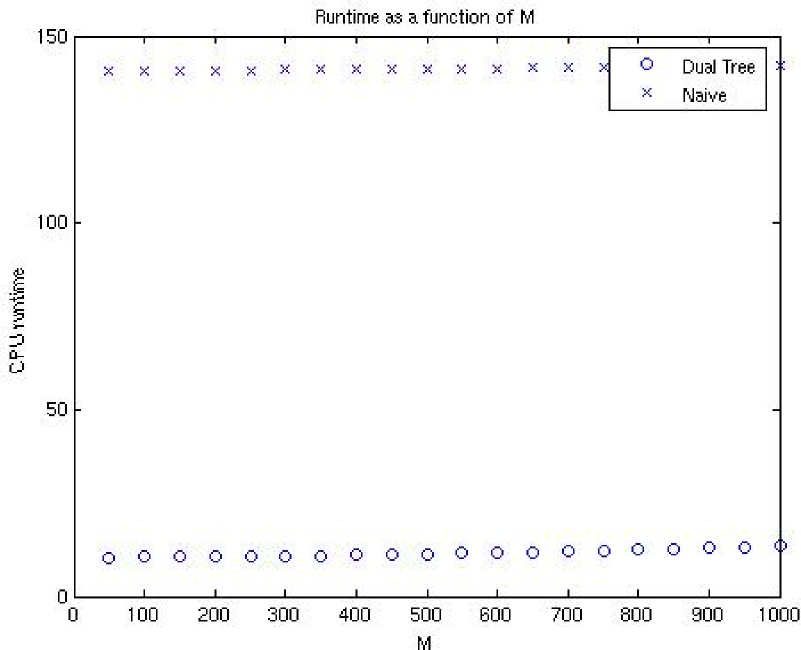
Dual Tree Algorithm. The algorithm runs an iterative deepening approach in which for every level 'i' in the query tree and for every level 'j' in the training tree, we compare every query node found at that level 'i' with every node found at level 'j' in the training tree. When comparing two individual nodes, we estimate the maximum contribution as described above, and if that contribution is less than some tolerance ε, we prune away the training node (and implicitly its children) and subtract the maximum contribution from ε. Once ε < 0 we can no longer prune nodes without increasing error so we explicitly calculate all probabilities from then on. Note that ε is inherited from the parent, but the two children can ultimately "use up" different amounts of ε depending on what training nodes they prune. For the other case, where we have found the two nodes to be close enough, we just repeat the comparison for the query node with the children of that training node. Once we have reached the leaves, we directly calculate the densities between each point in the query node with each point in the training node, using a Gaussian kernel. Since we have managed to prune away distant training points that contribute little to the density, we have considerably reduced our calculations and thereby greatly improved the overall running time. Note that even with a small ε tolerance (0.001), on average many nodes are pruned.

Results

As can be seen on the graph that follows, dual-tree does not do well in high dimensional spaces. The problem is that for really high dimensional spaces most of the complexity comes from the number of dimensions rather than the number of points, so pruning away points does not result in as much speedup.

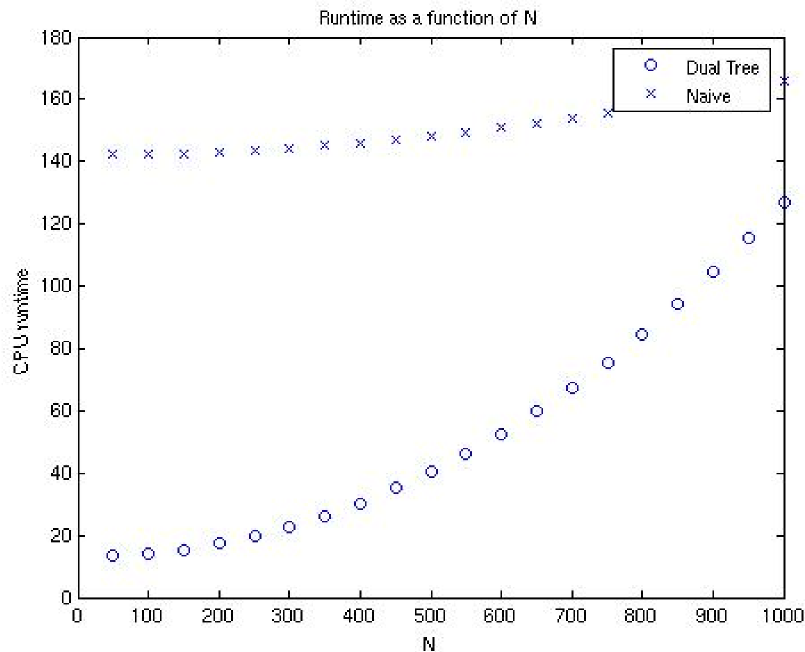


However, when dimensionality and the number of training points is held constant, dual-tree is considerably faster than a naïve O(MN) approach and retains that edge even for large M.



The most curious result is that as N increases, dual-tree performs worse asymptotically than a naïve approach. There is still a speedup for N < 1500 or so, but after that dual-tree definitely performs more slowly. This was re-verified on another dataset with 5000 training points. This suggests an inefficiency in the way that we are traversing and pruning the training tree. There might be issues with memory garbage collection at the interface between our C++ code and the R

interface that invokes it. We are continuing to revise our code to improve efficiency and are confident that observed asymptotic performance can be improved.



Future work will consist primarily of optimizing the performance of the code and experimenting with other pruning methods.

Acknowledgements

We collaborated closely with Balaji Subbaraman Srinivasan in Serafim Batzoglou's lab on this project. It should be noted that implementations of dual-tree do exist, but their performance leaves something to be desired, especially for large datasets, hence our efforts to rewrite the algorithm in C++ and expose an R interface so that it can be used as a general purpose tool (and Serafim's lab in particular in this case.)

---

[1] Srinivasan BS, Novak AF, Flannick JA, Batzoglou S, and McAdams HH. Integrated Protein Interaction Networks for 11 Microbes. In submission.

[2] A Gray and A Moore, Rapid evaluation of multiple density models, AISTATS, 2003.

[3] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209-226, September 1977.

[4] K. Deng and A.W. Moore. Multiresolution instance-based learning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 1233-1239, San Francisco 1995.