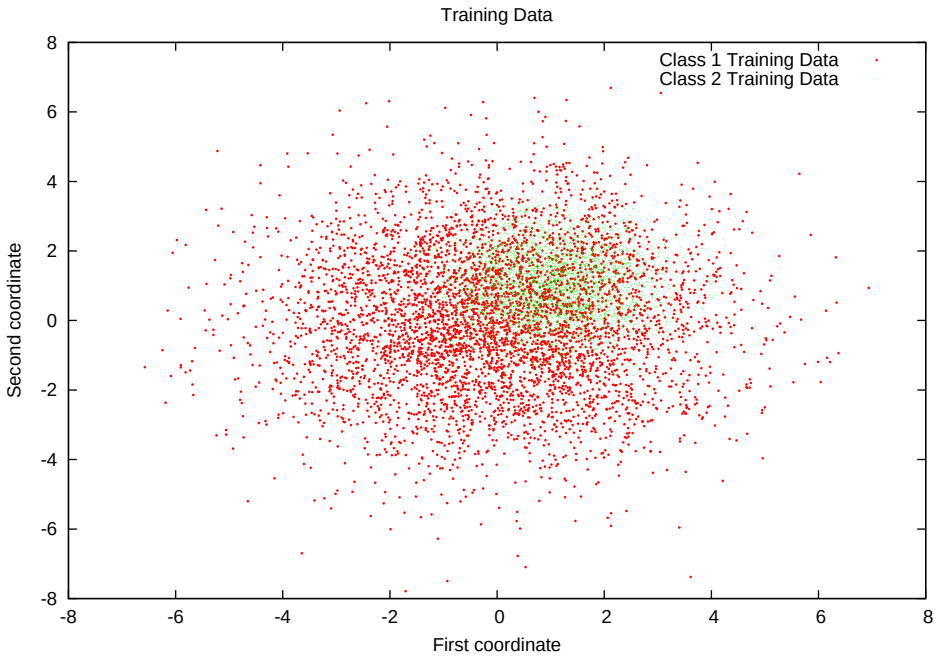
CS531 Assignment 4

Bryan Topp

March 30, 2014

- Scatter Plot of Data



(I tried for a very long time to get a good overlay of estimated Gaussian isocontours. I couldn't do it.)

- Ideal Bayes Error

I estimated the error rate of a minimum-error classifier numerically using a C program. First, I estimate the parent distribution means and covariances from the training data sets. These are simply computed as the sample mean and sample covariance matrix for each set of samples. Then, I numerically integrate Gaussians with such parameters over a fixed grid to approximate the Bayes error. For each of 2048 points spanning 4 times the sample variance (that is, the diagonal terms of the estimated covariance matrix) in each dimension, I evaluate each of the Gaussian functions. The ideal classifier would pick the greater of the two at each point - thus, the lesser of the two (multiplied by the area of summation, to give an approximate volume) is accumulated as error. Following the integration, the two error rates (denoted class-1-as-2 and class-2-as-1) are averaged, as we are assuming equal priors. This gives the ideal Bayes error if we were to build a minimum-error classifier having a model of the distributions and estimating the maximum-likelihood parameters for them.

The program runs in approximately 1.3 seconds on a Core i7 920 system and uses about 3MB of RAM.

The results of this estimation are as follows:

Estimating over bounds -16.265510, -15.674212 to 16.205234, 15.694166

Using 2048 * 2048 total grid points = 4194304

	Sample Mean	Sample Covariance Matrix
Class 1	-0.030138, 0.009977	4.058843, 0.061895, 0.061895, 3.921047
Class 2	1.008376, 0.993050	1.019485, 0.012932, 0.012932, 0.994920

Error volume for classifying class 1 as 2: 0.312500.

Error volume for classifying class 2 as 1: 0.126870.

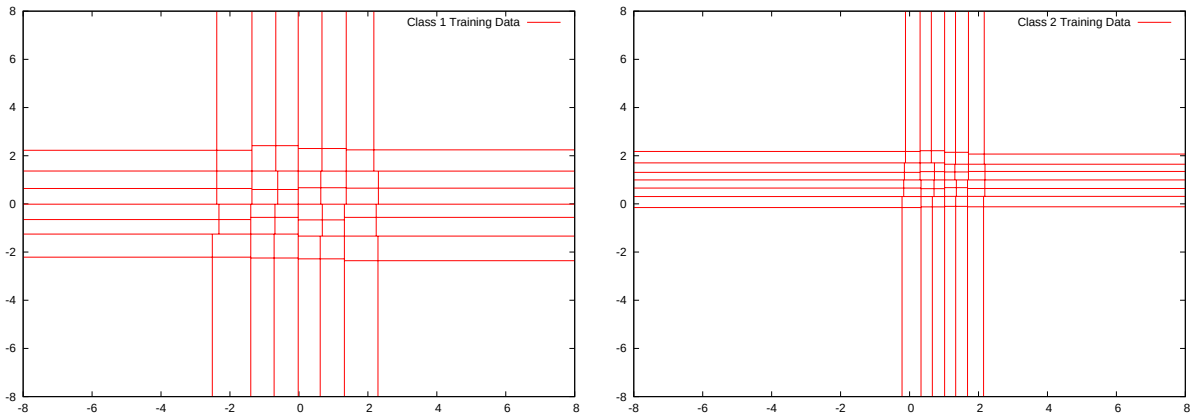
Overall Bayes error given equal priors: 0.219685.

- Parzen Windows classifier

  In implementing the Parzen Windows classifier, I decided to pre-process the training data to improve the asymptotic scaling of the classification algorithm. I assume that the classifier will be trained once, and then used many times to classify data (i.e., the size of the training set is much smaller than the size of all items to be classified), so this will similarly improve the overall "speed" of the classifier. Additionally this approach helps if, for example, the classifier is trained once using high-end hardware and then deployed on low-end hardware.

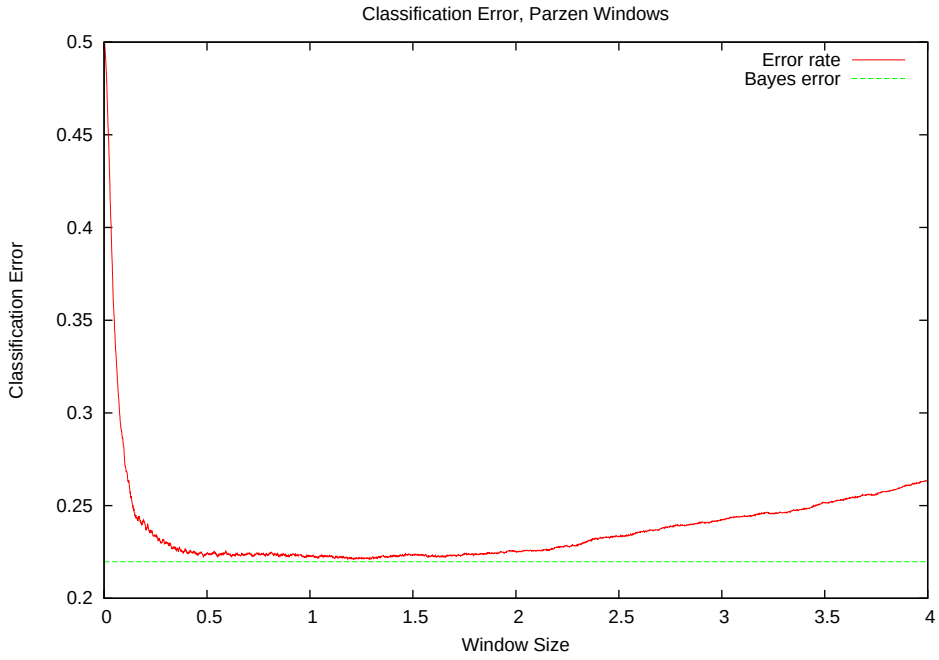
  Each class training set is recursively subdivided into a 2-dimensional $k-d$ tree. The internal nodes of this tree specify an x and y coordinate, which split a space into 4 axially-divided subspaces. The subspaces are the children of such a node. The algorithm stops when a subspace contains fewer than l training samples, where l is tunable and set at 100 for demonstration. Each splitting point's coordinates are chosen based on the individual coordinate medians of the points in the subspace.

  The given training sets, when split like this, appear as such:

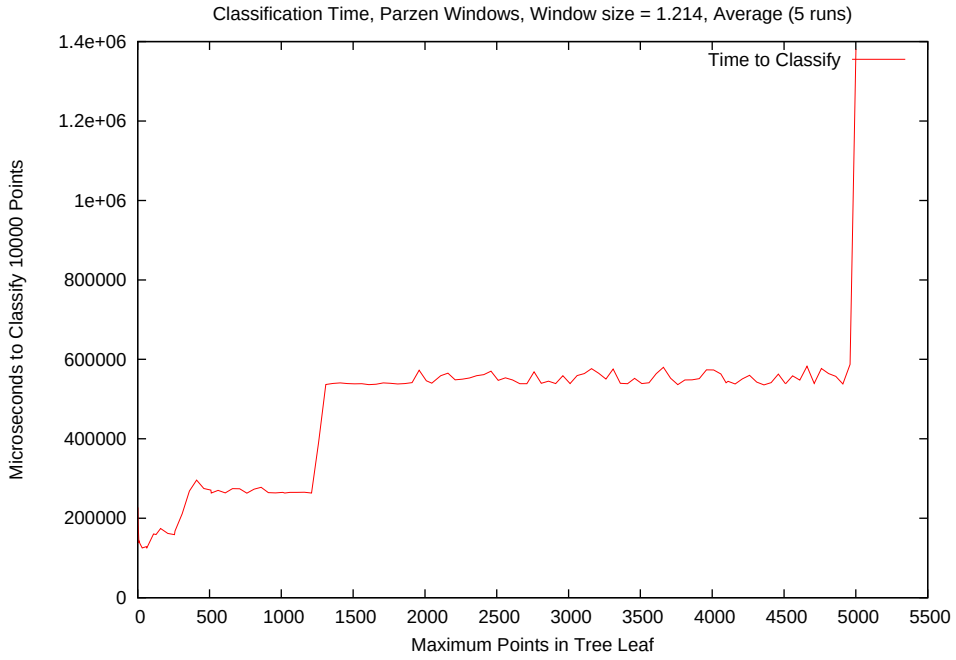


  Given a windowing function with finite support, we no longer have to evaluate all training points per classification. We can search the tree to find all subspaces which intersect the window, and only consider the points within. The ability to search using the tree structure means that each classification instance no longer scales with $O(d)$ versus the size of the training dataset, but with $O(\log d)$. When the training dataset becomes sufficiently large and the window sufficiently small, this could lead to significant improvements. For small training sets or large windows, the benefit is less.

  After implementing the classifier and confirming that the tree subdivisions did not alter the results (as they are only an optimization, and should not fundamentally alter the algorithm), I evaluated the accuracy versus the window size. I used a linear search from 0 to 4 with a step size of $\frac{1}{1000}$. The results are as follows:



Window sizes between 0.5 and 2 give good results. The minimum error seen was with window size 1.214, giving an error of 0.221100. This is within 1% of the Bayes error. Given the window size resulting in the best classification, I evaluated the run-time of the classifier as I varied the maximum leaf size of the tree. Note that tree leaf size 1 crashes with a stack overflow. Results are as follows:

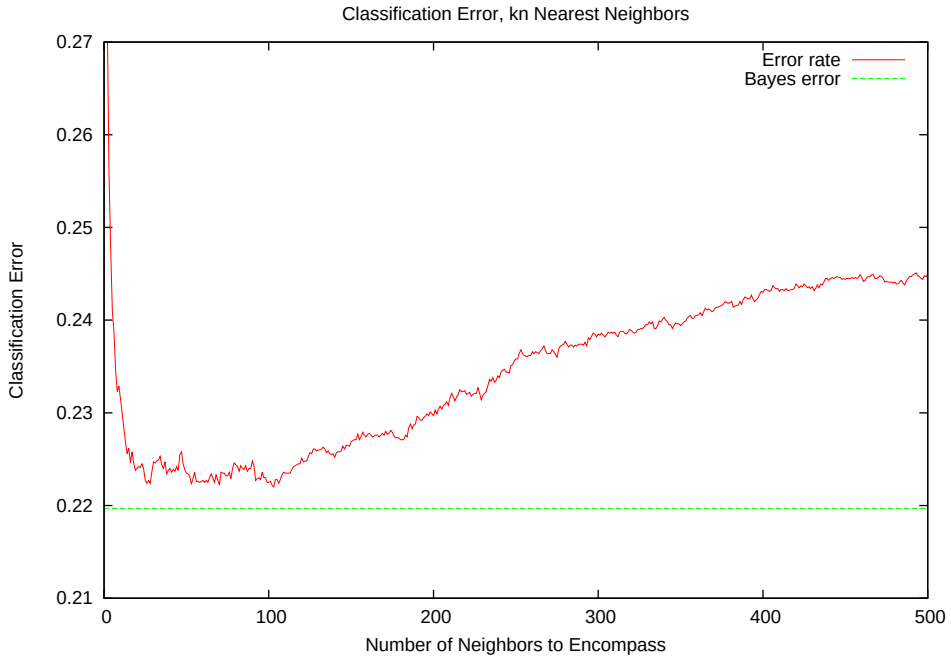


The case where the maximum leaf size is 5000 results in a simple linear search, and every single data point is evaluated with the window function for each classification. This is the worst case, taking 1380585μ sec to classify the data sets. The best case is observed at a leaf size of 32, taking 125082μ sec. The improvement is considerable - the tree can make classification over 10 times faster.

(The time to build the tree is not included in these results. Even including this - taking the total run time of the application - we see an improvement in these cases, less precisely, from 1.590s to 0.186s.)

- K_n Nearest Neighbors

  This algorithm is significantly harder to accelerate with pre-processing. I had considered some type of additional data on each training sample, possibly specifying the distance to the next nearest sample, or something of the sort - the amount of data needed would potentially be very large though. General tree structures do not work well either. I settled on an implementation which performs a simple search through the training samples for a class, finding the k nearest to an evaluation point. The time to find the smallest k values of a set of n (a selection algorithm) is generally $O(n)$, which is not too bad. In particular, I used the quickselect algorithm. The K_n Nearest Neighbors classification indeed takes much longer to run than the Parzen Windows method. A single run to classify 10000 points takes multiple seconds even on a modern machine with an optimizing compiler; the results below took many minutes to run and culminated in a shutdown due to a critical CPU temperature.



  The trend continues to the right, roughly linearly, up to at least 2200 points. The graph is zoomed to show the interesting behavior; all values around 20 to 120 give good results. The best case in this example is to find 103 neighbors, giving an error rate of 0.222, around 1% more than the Bayes error. The worst case is when only using 1 point (0.3055 error), in which case the classifier becomes equivalent to a Nearest Neighbor algorithm - naturally, whichever class has a closer neighbor will have a smaller volume and thus a higher estimated density.

- Nearest Neighbor

  This algorithm does not require too much exposition. Each classification causes a single linear search through all training data (unlike the earlier methods, both classes of training data are considered simultaneously). The sole nearest training point to the evaluation point is used to determine the class. There are no parameters and nothing special about my implementation. The error rate using the given data is 0.305500. This is 40% worse than the Bayes error. Still, it is impressive for such a simple classification algorithm. Examining the decision boundary results (given later in the report), it becomes evident that Nearest Neighbor is equivalent to both N Nearest Neighbors and K_n Nearest Neighbors when their parameters are each set to 1.

- N Nearest Neighbors

  I implemented this algorithm with a sort, rather than a selection algorithm. With K_n Nearest Neighbors, I only need to find the kth nearest training point for each class, so a given evaluation incurs 2 quickselect runs. In N Nearest Neighbors, I need the 1st through the nth nearest training points for both classes, so each evaluation would incur n quickselect runs. Therefore, it is faster to use a single quicksort and take the first n elements.

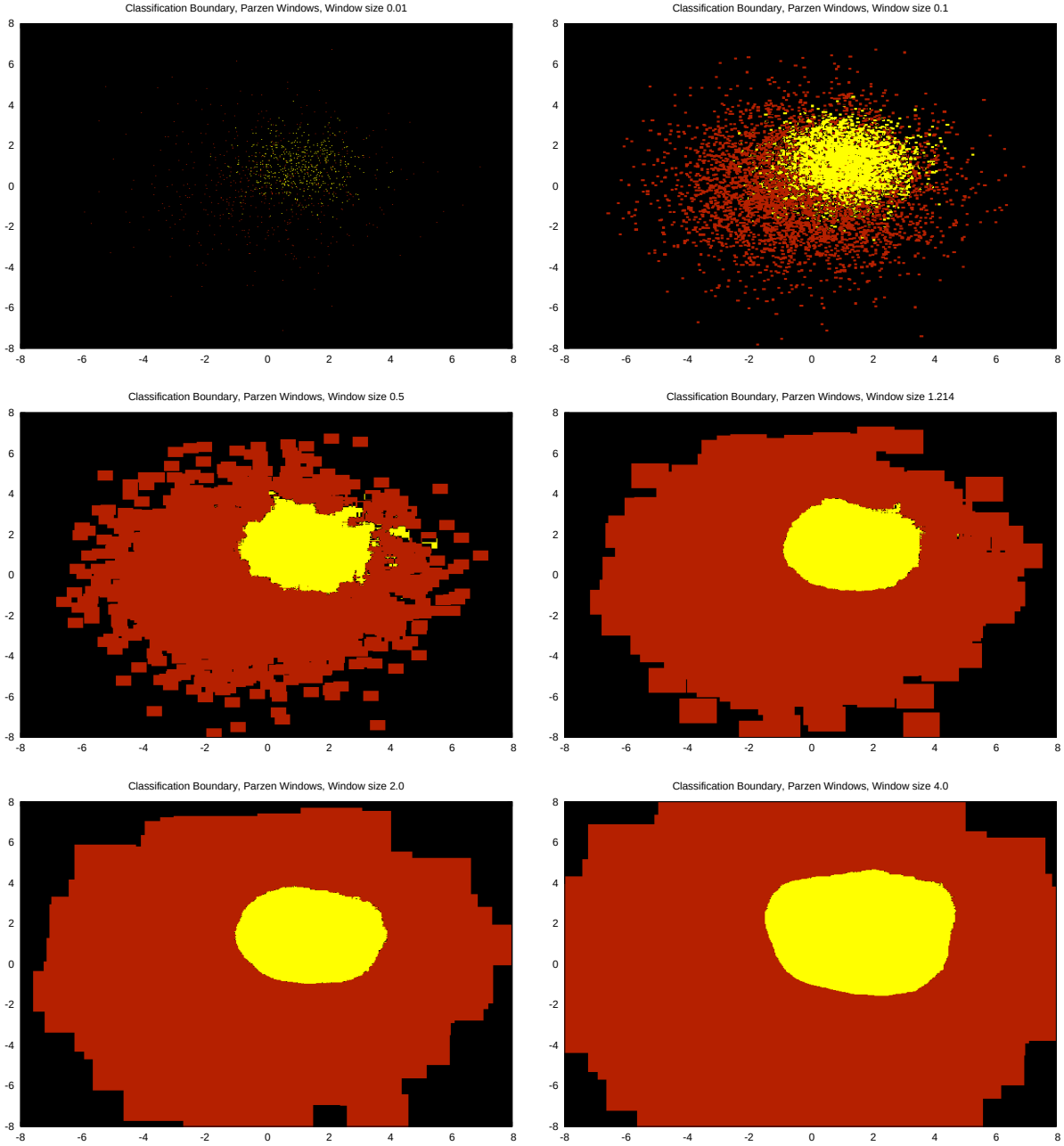
  My implementation of this algorithm takes some steps to avoid "ties". If two training points of different classes are equidistant from an evaluation point, they are ignored. Additionally, when the N nearest neighbors "vote" on the best class, ties are broken by dropping the farthest point until the algorithm devolves into a simple Nearest Neighbor, which is quite unlikely to run into the same problem.

  Results are as follows:

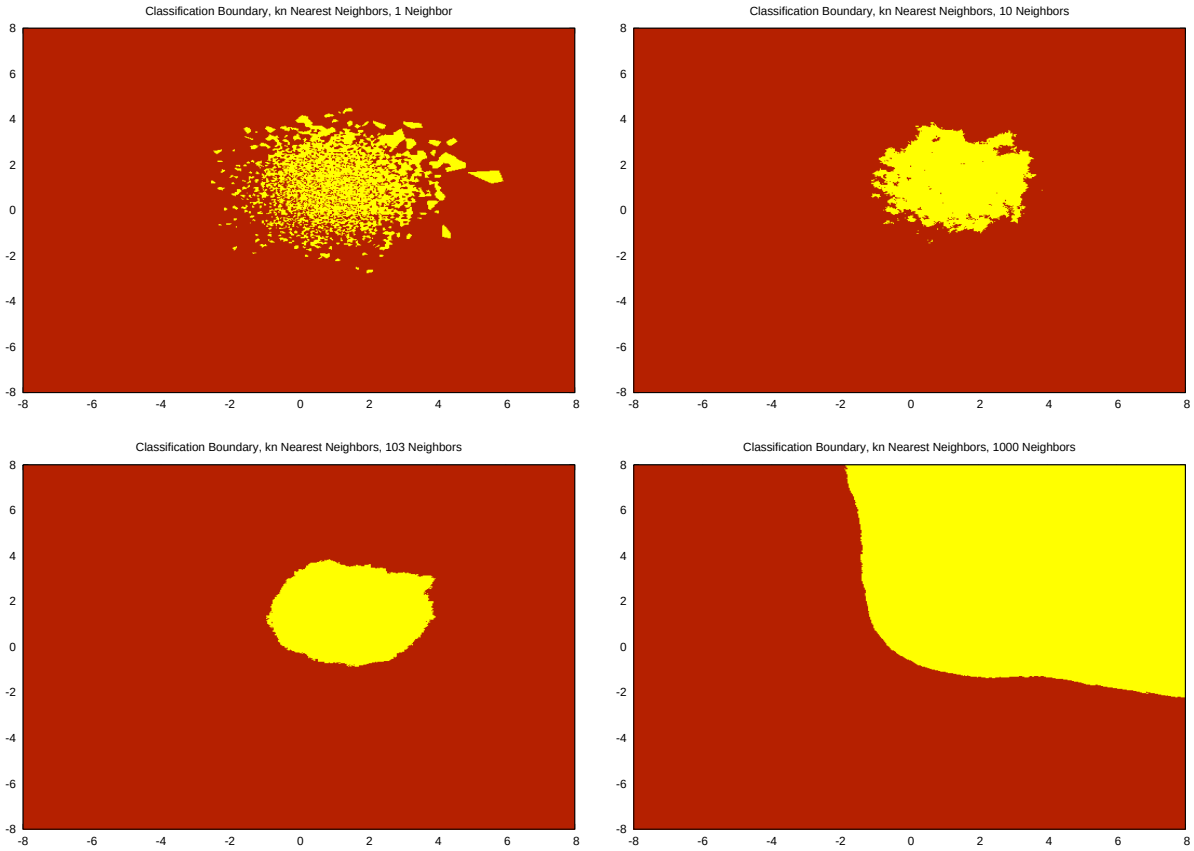
  The trend continues upwards as more neighbors are brought into each vote. Values around 50 to 250 seem to give good results. The best result was seen when taking a vote of the closest 205 neighbors, giving an error rate of 0.222, around 1% more than the Bayes error and the same as the optimal K_n Nearest Neighbors parameter. Similarly, the algorithm devolves to Nearest Neighbor when only considering 1 point.

- Comparison of Decision Boundaries

  The Parzen Windows method is quite tunable, and gives interesting results across a variety of window sizes. Below, various settings are shown. Black areas are those in which there is a tie and we could make a decision randomly. Red areas represent those where c_1 will be chosen. Yellow represent those where c_2 will be chosen. The detail around the boundary can be seen to wash out when the window size is too large. Similarly, small windows can create spotty boundaries.

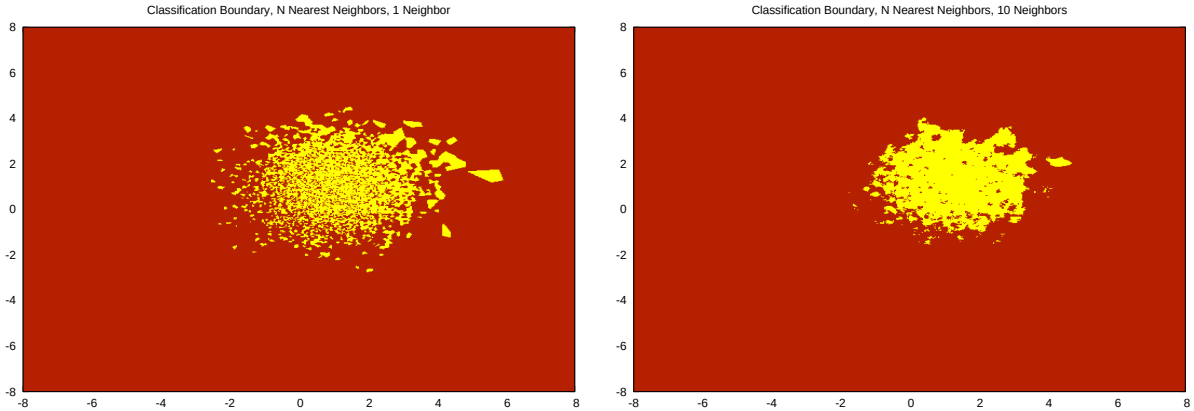


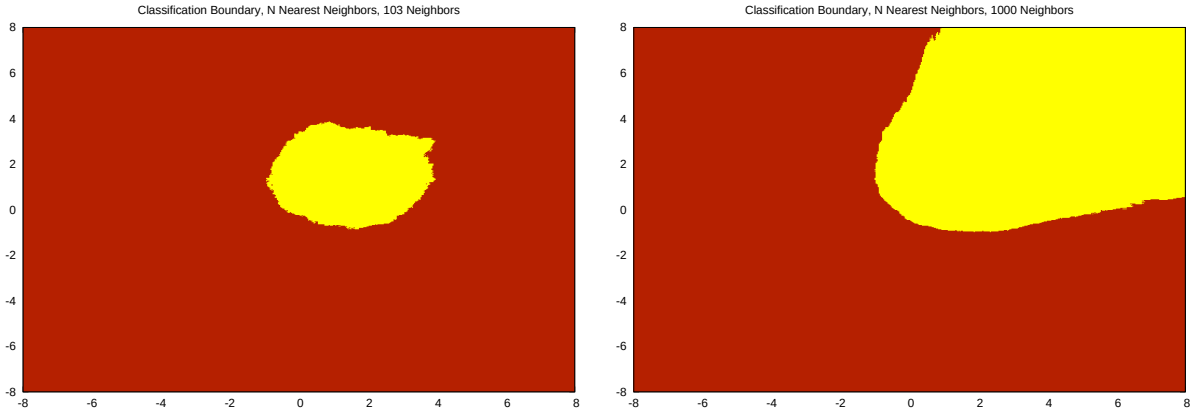
The K_n Nearest Neighbors algorithm is also tunable. Various decision boundaries are shown below. Because the volume size adapts based on surrounding data, there are no longer large areas of "ties". Such cases will be rare, as few training points will ever even be equidistant from an observation.



Interestingly, as samples become sparse and many are required, the decision boundary distorts to give samples of c_2 more weight than it should.

The N Nearest Neighbors algorithm is tunable as well. In the $N = 1$ case, it is equivalent to K_n Nearest Neighbors and Nearest Neighbor. All three reduce, in this case, to picking the class with the single closest training sample.





It is interesting to note that the c_2 outliers are not eliminated as readily by this algorithm. Additionally, the distortion suffered by selecting too many neighbors seems not to be as severe as with the K_n Nearest Neighbors algorithm.

- Algorithmic Complexity

  All of the classifiers scale linearly with the number of dimensions in the problem, and of course with the number of points to be classified. The size of the training set has an interesting impact, however.

  The Parzen Windows method considers a fixed, limited spatial area in each classification, so it is subject to acceleration with a spatial tree structure. For small window sizes, it scales with $\Theta(\log N)$ versus the size of the training sets. The benefits of the tree become less apparent as the window size starts to encompass more of the data (or rather, forces traversal of more of the tree), and the worst case becomes $\Theta(N)$.

  The K_n Nearest Neighbors algorithm is not as immediately amenable to tree structures, but is still relatively fast. To find a volume encompassing the k nearest neighbors, we only need to know the radius to the kth nearest neighbor. So, a quickselect algorithm is appropriate, and runs in $\Theta(N)$ average time versus the size of the training sets.

  The N Nearest Neighbors algorithm is less friendly, and essentially requires a sort for each classification. For k nearest neighbors, all neighbors (from both training sets!) from the closest to the kth closest must be considered. If k is of the same magnitude as N, which the data indicated it should be, then using quickselect k times to grab this many points give a complexity of nearly $\Theta(N^2)$ versus the size of the (combined) training set. A quicksort at $\Theta(N \log N)$ is thus the most obvious solution. (I have not looked thoroughly into quickselect; perhaps it could be written to run in approximately $\Theta(n \log k)$ time when selecting k points.)

- Conclusion

  Given that the three algorithms (not counting Nearest Neighbor) produce roughly the same accuracy with the optimal tuning results, Parzen Windows would seem to be the best approach in this type of problem. It is many orders of magnitude faster, it seems less prone to unpredictable variance based on the tuning parameter, and it performs in fact slightly better.