HW1 K-Nearest Neighbors Algorithm Analysis

Keith G. Williams 800690755

February 08, 2016

# Algorithm Descriptions

## *k*-Nearest Neighbors

For *k*-Nearest Neighbors, each test tuple is iterated over, and its distance to every other training point is checked. Since the training examples are in a numpy array, numpy's broadcasting feature is utilized to make these distance calculations fast. For only the minimum distance is required, so for 1-NN, the label of the closest tuple is found and appeneded to the testY label array. For , the minimum distance is not sufficient, so the resulting distance array is sorted, and the first distances are sliced and labeled. The mode label is applied to the test point (majority vote) and appended to the testY label array.  
Since this approach assumes features are represented as tuples in and euclidean distance is used, less relevant features will be equally weighted with more relevant features and may lead to less accurate results.

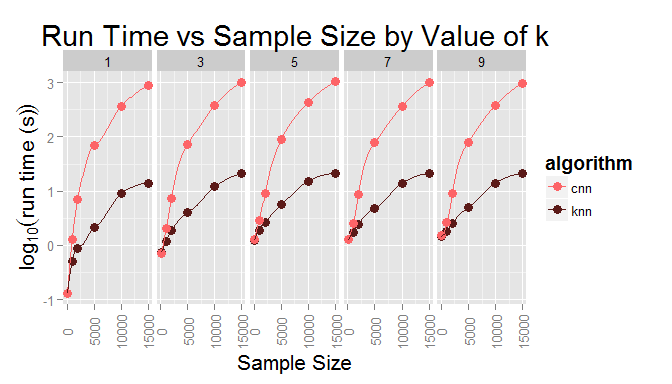
## Condensed Nearest Neighbors

To minimize the memory requirements for *k*-NN, a condensed training set can be found such that the decision boundary is approximately the same for the condensed subset as the full training set. The points in the condensed subset are called the prototypes.  
First, pairwise distances are calculated for each -d point in the training set. These distances are stored as an matrix, where element is the distance from point to point in the training set. Though all pairwise distances *might* not be necessary to create a condensed set, calculating them up front avoids recalculating the same distance between two arbitrary points multiple times during the choosing process. Second, the condensed subset is seeded with one random prototype from each class in the training set. The indexes for each prototype are stored as members of a boolean array of length , where if point is a prototype in the condensed subset. This boolean array can also be used as a mask to slice the euclidean distance matrix to get the distance of every point in the training set to each prototype in the condensed set during the choosing process, and it can be used to slice trainY to get the class label for each prototype in the condensed set. Next, the remaining training points are labeled according to 1-NN, where each point gets the label of its nearest prototype. This label is found by retrieving the index in the sliced euclidean distance matrix where element is the minimum value along column . This index is used to index the label array such that label is the label of prototype . Finally one pass is made over the remaining training points. The predicted label is compared to the true label, and if they are not equal, the point is added to the prototype set, and each remaining point is relabeled according to 1-NN against the new prototype set.  
A single pass over the training points after initialization is chosen, so that once a point is labeled correctly, it is never checked again. However, this approach is sensitive to the order in which the points are presented. Further, there are very likely redundant points in the condensed subset. If one wanted to remove these redundancies, each point's utility could be checked by running leave-one-out 1-NN to check if any labels change when each point is removed from the condensed set. The post-pruning approach would require longer training runtimes, but would improve memory requirements for later tests on novel points.

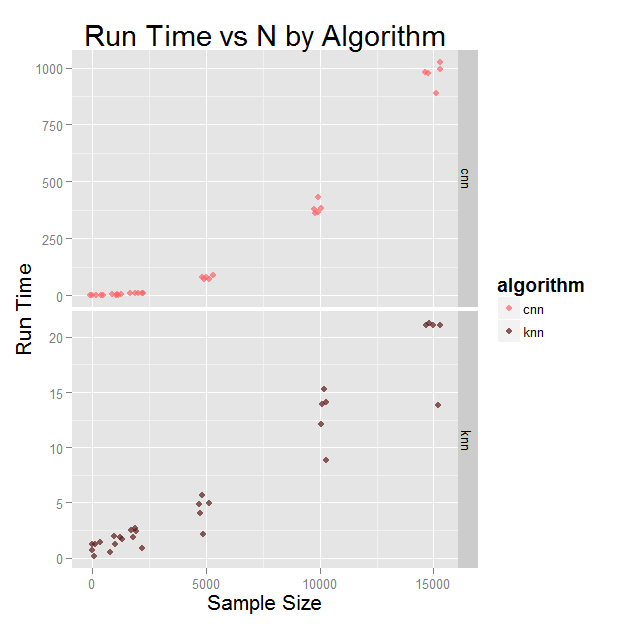
# Experimental Results

60 experiments were run to test the effects of sample size and on run time and accuracy for my implementations of *k*-Nearest Neighbors and Condensed Nearest Neighbors. Sample sizes of 100, 1000, 5000, 10,000 and 15,000 and values of 1, 3, 5, 7, and 9 were tested on each algorithm. The results of each experiment are included in the appendix.

## Run Time Analysis

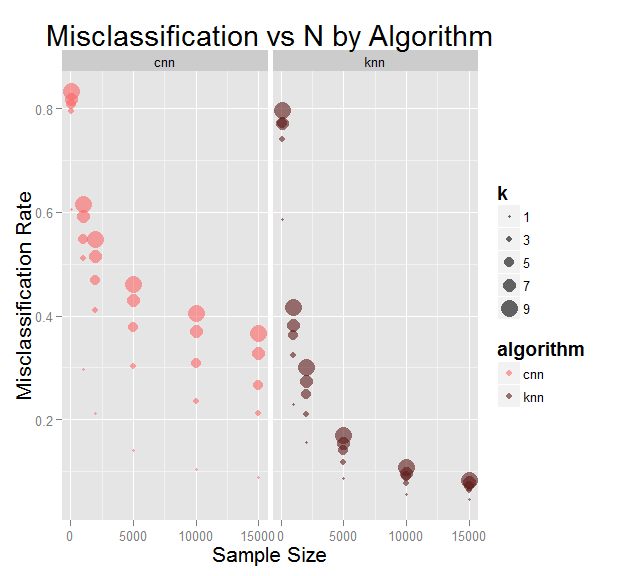


From the above plot, one can see condensed NN is an order of magnitude slower than *k*-NN for the same value of and . Further, by comparing each value of for the same , one can see that there is no obvious trend for the effect of on run time. One important exception is for vs . Since the check for nearest neighbors is qualitatively different for , a small improvement in run time is observed. Since only the minimum distance value is needed rather than the closest distances, there is no need to sort neighbor distances for .  
To more directly investigate the effect of sample size on run time, the below plots sample size versus run time for each algorithm:



One can also see from this plot that the effect of is minimal. Additionally, this plot shows condensed NN is quadratic in n, while *k*-NN is sub-quadratic.

## Classification Accuracy



From the above plot, it can be concluded that sample size is the most important factor in classification accuracy; for both algorithms misclassification rate is inversely related to sample size. Further, for each sample size, condensed NN has a much higher misclassification rate. Though, it should be noted that for , condensed NN is much more competitive with *k*-NN. Finally, across all sample sizes, for both algorithms, increasing the value of consistently increases the misclassification rate.

## Confusion Matrix Exemplar

Below is the Confusion Matrix for *k*-NN on 15000 training examples, where . Rows of the matrix are true labels, while columns are predicted labels. The misclassification rate is good at 7%, but a few errors (in the off diagonal) tell an interesting story. It appears the classifier has trouble distinguishing 'F' from 'P', 'K' from 'X' and 'K' from 'H'. Given that the symbols for these letters are similar, this makes sense.

**Predicted Label**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | 203 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 160 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 6 | 3 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| C | 0 | 0 | 168 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 4 | 0 | 202 | 0 | 0 | 0 | 7 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 1 | 2 | 0 | 165 | 1 | 5 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 9 |
| F | 0 | 1 | 1 | 0 | 1 | 175 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| G | 0 | 1 | 3 | 2 | 3 | 1 | 187 | 2 | 0 | 0 | 1 | 1 | 2 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| H | 0 | 3 | 0 | 6 | 4 | 0 | 3 | 143 | 0 | 0 | 7 | 0 | 1 | 0 | 3 | 0 | 2 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| I | 0 | 1 | 0 | 1 | 0 | 2 | 0 | 0 | 193 | 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| J  **True Label** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 5 | 174 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| K | 0 | 2 | 0 | 1 | 2 | 0 | 1 | 14 | 0 | 0 | 148 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 7 | 0 | 0 |
| L | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 197 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| M | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 182 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| N | 1 | 1 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 3 | 179 | 1 | 0 | 0 | 5 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| O | 0 | 1 | 1 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 167 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| P | 1 | 1 | 0 | 1 | 0 | 12 | 0 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 184 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Q | 0 | 0 | 0 | 1 | 1 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 1 | 204 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| R | 0 | 2 | 0 | 5 | 1 | 2 | 0 | 6 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 1 | 1 | 185 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| S | 0 | 2 | 0 | 0 | 4 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 184 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| T | 0 | 1 | 0 | 3 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 171 | 0 | 1 | 0 | 1 | 2 | 0 |
| U | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 208 | 1 | 0 | 0 | 0 | 0 |
| V | 0 | 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 159 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 162 | 0 | 0 | 0 |
| X | 1 | 1 | 0 | 1 | 5 | 0 | 0 | 0 | 0 | 1 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 170 | 0 | 1 |
| Y | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 1 | 175 | 0 |
| Z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 189 |

# Appendix

### Figure 1: Experimental Data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| algorithm | k | sample\_size | accuracy | run\_time |
| knn | 1 | 100 | 0.4144667 | 0.1307928 |
| knn | 3 | 100 | 0.2583333 | 0.7292185 |
| knn | 5 | 100 | 0.2290667 | 1.2488839 |
| knn | 7 | 100 | 0.2290667 | 1.2610384 |
| knn | 9 | 100 | 0.2036000 | 1.4479178 |
| cnn | 1 | 100 | 0.3936000 | 0.1288576 |
| cnn | 3 | 100 | 0.2059333 | 0.7170977 |
| cnn | 5 | 100 | 0.1917333 | 1.2970113 |
| cnn | 7 | 100 | 0.1814000 | 1.3117525 |
| cnn | 9 | 100 | 0.1671333 | 1.5044048 |
| knn | 1 | 1000 | 0.7696667 | 0.5119107 |
| knn | 3 | 1000 | 0.6767333 | 1.2067960 |
| knn | 5 | 1000 | 0.6374000 | 1.9201712 |
| knn | 7 | 1000 | 0.6182000 | 1.7210886 |
| knn | 9 | 1000 | 0.5830000 | 1.8373649 |
| cnn | 1 | 1000 | 0.7032000 | 1.2692344 |
| cnn | 3 | 1000 | 0.4891333 | 2.0451665 |
| cnn | 5 | 1000 | 0.4513333 | 2.9106198 |
| cnn | 7 | 1000 | 0.4070667 | 2.4982342 |
| cnn | 9 | 1000 | 0.3842000 | 2.7023353 |
| knn | 1 | 2000 | 0.8439333 | 0.8746457 |
| knn | 3 | 2000 | 0.7893333 | 1.8637551 |
| knn | 5 | 2000 | 0.7508000 | 2.6908534 |
| knn | 7 | 2000 | 0.7252667 | 2.4525257 |
| knn | 9 | 2000 | 0.6990667 | 2.5392322 |
| cnn | 1 | 2000 | 0.7882667 | 6.8980086 |
| cnn | 3 | 2000 | 0.5898000 | 7.4310129 |
| cnn | 5 | 2000 | 0.5306667 | 9.1844432 |
| cnn | 7 | 2000 | 0.4846667 | 8.6263285 |
| cnn | 9 | 2000 | 0.4530000 | 9.1075020 |
| knn | 1 | 5000 | 0.9124667 | 2.1074052 |
| knn | 3 | 5000 | 0.8826667 | 4.0118147 |
| knn | 5 | 5000 | 0.8602000 | 5.7141940 |
| knn | 7 | 5000 | 0.8450000 | 4.8391042 |
| knn | 9 | 5000 | 0.8302667 | 4.9839498 |
| cnn | 1 | 5000 | 0.8587333 | 69.8826424 |
| cnn | 3 | 5000 | 0.6977333 | 70.9902930 |
| cnn | 5 | 5000 | 0.6218667 | 88.8768771 |
| cnn | 7 | 5000 | 0.5692000 | 77.2234150 |
| cnn | 9 | 5000 | 0.5382000 | 78.8709295 |
| knn | 1 | 10000 | 0.9441333 | 8.8740223 |
| knn | 3 | 10000 | 0.9226000 | 12.1150699 |
| knn | 5 | 10000 | 0.9094000 | 15.2938325 |
| knn | 7 | 10000 | 0.9045333 | 13.9410005 |
| knn | 9 | 10000 | 0.8927333 | 14.0818811 |
| cnn | 1 | 10000 | 0.8965333 | 359.5721113 |
| cnn | 3 | 10000 | 0.7647333 | 381.7616871 |
| cnn | 5 | 10000 | 0.6912667 | 432.0180691 |
| cnn | 7 | 10000 | 0.6302000 | 366.8824201 |
| cnn | 9 | 10000 | 0.5947333 | 378.9983289 |
| knn | 1 | 15000 | 0.9547333 | 13.8323979 |
| knn | 3 | 15000 | 0.9371333 | 21.0294983 |
| knn | 5 | 15000 | 0.9295333 | 21.0915528 |
| knn | 7 | 15000 | 0.9214667 | 21.0278783 |
| knn | 9 | 15000 | 0.9173333 | 21.2452001 |
| cnn | 1 | 15000 | 0.9117333 | 888.0537620 |
| cnn | 3 | 15000 | 0.7876000 | 995.0610164 |
| cnn | 5 | 15000 | 0.7330000 | 1024.7707833 |
| cnn | 7 | 15000 | 0.6726000 | 981.9318193 |
| cnn | 9 | 15000 | 0.6343333 | 976.5080718 |