Machine Problem 1:
K-Nearest Neighbor on MNIST

By Vincenzo Marconi

Problem Statement

For this assignment we are to use the K Nearest Neighbors (KNN) algorithm to determine the digit value of an image of a handwritten digit. We are using the MNIST data set of handwritten digits which contains 60,000 images of digits that are 28 by 28 pixels with greyscale values. With KNN we are given an image of a digit and we are to determine what digit the image contains. With 60,000 pictures we divide our data set into a training set, a test set, and a validation set, based on one of three test protocols.

In test protocol 1, we partition the data set into 50,000 training examples and 10,000 test examples. We calculate the KNN of each element of the test set with those of the training set for an arbitrary K value.

In test protocol 2, we divide the data set into 40,000 training examples, 10,000 validation examples, and 10,000 test examples. We run KNN on the validation examples with a range of K values and we tune our K to find some value X that is the K with the minimum error. We then run KNN on our test set with this X value.

In test protocol 3, we perform a 5-fold and 10-fold cross validation on the data set. In the 5-fold, cross validation we partition our dataset into 2 sets: a training set of 48,000 samples and a validation set of 12,000 samples. We run KNN on set of K values, and we perform it 5 times with different validation sets picked from partitioning the 60,000 samples into training and validation sets in 5 different ways. For the 10-fold cross validation, we perform the same tasks as our 5-fold but by partitioning our data into 10 sections instead of 5.

Implementation

For this program, I used C++ with Apple's Accelerate Framework which is builtin into the MacBook. Specifically I used the LAPACK or Linear Algebra Package inside the Framework to perform all vector operations.

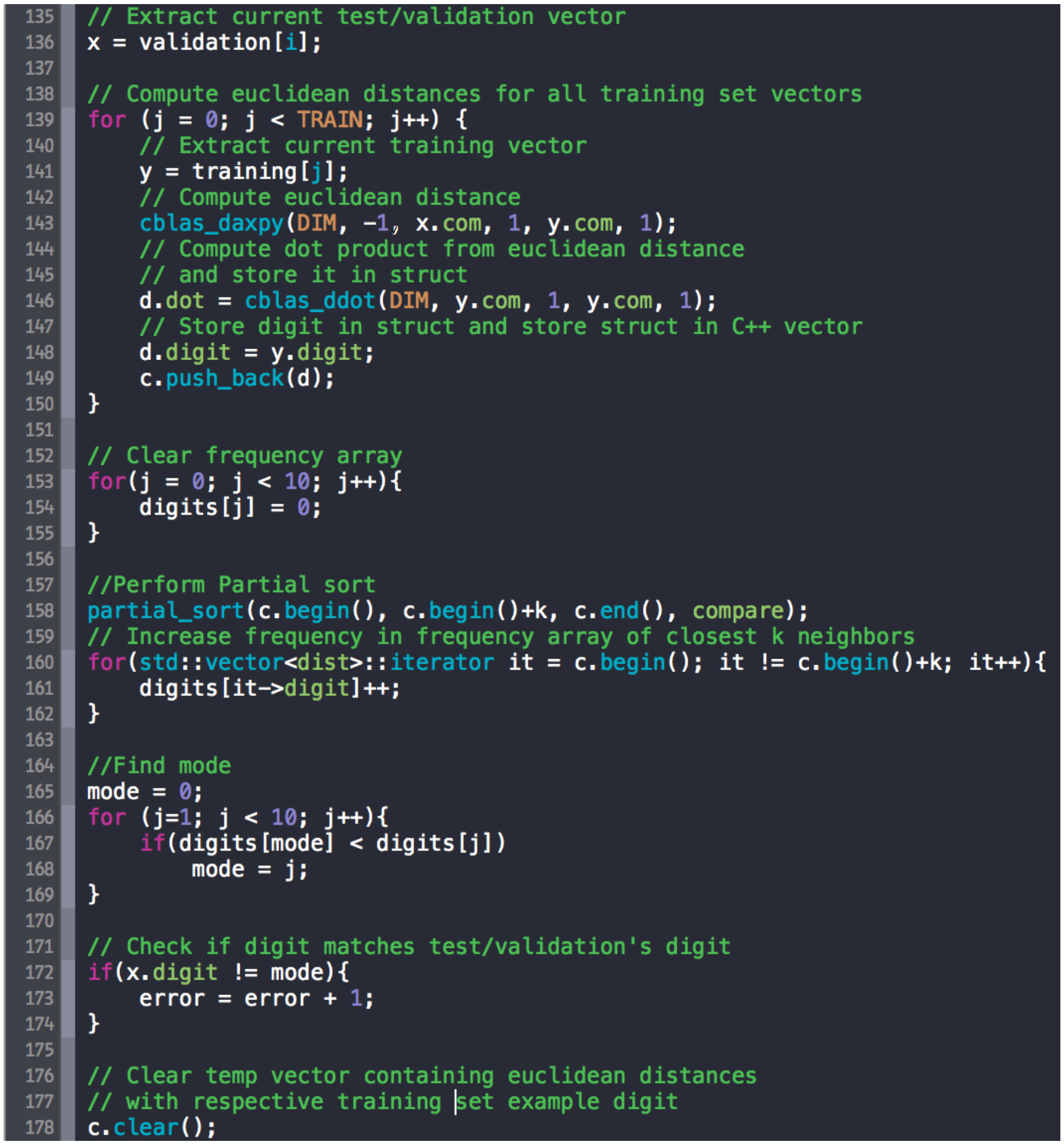
To find the K nearest neighbor, I performed the euclidean distance between each vector of the test/validation set with each vector of the training set, and saved the square of the value of the euclidean distance into a temporary C++ vector with the given training vector's digit; the euclidean distance and the digit of the given training vector is stored in a struct which is what is saved in the temporary C++ vector. I then proceeded to do a partial sort (which is built into C++) on the vector with a comparison function that finds the K minimum euclidean distances. The partial sort itself performs a selection sort on the K smallest values of the given array, which gives it a complexity of the product of the size of the training set and the size of K. With this in mind, the performance of my program varied between 6 minutes and 10 minutes. The program ran for 6 minutes on K=1, about 7 minutes on K=5, and finally a little over 10 minutes with K=50,000. To find the pattern in the K values the program ran a little over 14 hours.

Using the following with correspondence to Figure 1:

T = size of training set
K = the number of nearest neighbors of KNN
α = the complexity of performing scalar multiplication with vector addition, computing the dot product of two vectors, and 3 other simple operations (lines 140-149)
Ω = the complexity of performing auxiliary operations (lines 153-155, 165-178)

The complexity of my algorithm was:

$$O(\ (\alpha + K)T + \Omega\ ) = O(\ (\alpha + K)T\ )$$



```
// Extract current test/validation vector
x = validation[i];

// Compute euclidean distances for all training set vectors
for (j = 0; j < TRAIN; j++) {
    // Extract current training vector
    y = training[j];
    // Compute euclidean distance
    cblas_daxpy(DIM, -1, x.com, 1, y.com, 1);
    // Compute dot product from euclidean distance
    // and store it in struct
    d.dot = cblas_ddot(DIM, y.com, 1, y.com, 1);
    // Store digit in struct and store struct in C++ vector
    d.digit = y.digit;
    c.push_back(d);
}

// Clear frequency array
for(j = 0; j < 10; j++){
    digits[j] = 0;
}

//Perform Partial sort
partial_sort(c.begin(), c.begin()+k, c.end(), compare);
// Increase frequency in frequency array of closest k neighbors
for(std::vector<dist>::iterator it = c.begin(); it != c.begin()+k; it++){
    digits[it->digit]++;
}

//Find mode
mode = 0;
for (j=1; j < 10; j++){
    if(digits[mode] < digits[j])
        mode = j;
}

// Check if digit matches test/validation's digit
if(x.digit != mode){
    error = error + 1;
}

// Clear temp vector containing euclidean distances
// with respective training set example digit
c.clear();
```

Figure 1: Code segment used to find K Nearest Neighbor for every element of the test or validation set.

Results

Test Protocol 1

I started out by testing KNN with very different K values: 1, 11, 100, 1000, 10000, 25000, 40000, 50000. I noticed as seen in Table 1 that as K became bigger so did the error, so I resorted to trying the first 30 odd values. From those results it is clear that K values from 1 to 5 had the minimum, and in this case specifically K=5 was the minimum with 2.83 % error while higher K values produced higher errors.

Test Protocol 2:

From the results of the previous test, I decided to try the first 10 values for K for the test containing the validation set. As seen in Table 2, the similar results to Test Protocol 1 manifested. The smaller values now ranged between K values of 1 to 5 with the smallest being K=5. From 5, higher K values only made the error higher. With a minimum error of 2.87%, I ran KNN on the test set with K=5 and got an error of 7.20 %. I believe this occurred mainly because of how the K value was tuned along with the population size. In this test protocol the vectors of the test set are now limited to 10,000 less than the previous test protocol, and tested with a K value that was tuned on a different set of vectors.

Test Protocol 3:

From the 5-Fold cross validation test, tuning the K value between 1 and 10 for KNN made a very subtle difference. Figures 3.2 and 3.4, display values for the X axis that oscillate between about 3.0 and 4.0 percent error, but as K increases there is a subtle increase in error in every other K value. The Y axis shows a little bit of uniformity in the errors when a partition was selected to be the validation set; it is noticeable though that when the second set of 6000 images are selected for the validation set the highest error occurs. The data overall in the 5-Fold cross validation appears to be needing a bit more data.

With the 10-Fold cross validation test, we then tune K between 1 and 15 for the KNN; the data and visualizations now begin to show more of where the error lies. First, from Figure 4.2, we can see on the Y axis that the partition that we select as our validation set does not seem to affect the error in a predictable way as the K value. It is clear from Figure 4.3, that as the K value increases so does the error. The data exhibits a bump in the first 2 K values in the error but then minimizes error at K=3 with an average value of 2.88 to then increase as K increases. At K=3, the results also appear less variable with a standard deviation of .32 which is the most stable.

Overall, the partitioning of data at these sizes of 60,000 samples into Training, Validation, and Testing appears to affect the error in both the size of the folds and which fold is selected. The data strongly suggests that higher K values produce higher errors. Finally, K values between 3 and 5 appear to be producing the least amount of error and a more stable distribution.

Table 1

K Value	Percentage Error
1	2.89
3	3.9
5	2.83
7	2.92
9	2.95
11	3.08
13	3.22
15	3.36
17	3.41
19	3.47
21	3.51
23	3.68
25	3.83
27	3.89
29	3.95
31	3.93
33	4.08
35	4.22
37	4.31
39	4.37
41	4.42
43	4.59
45	4.64
47	4.68
49	4.74
51	4.76
53	4.8
55	4.84
57	4.9
59	4.95
100	5.53
1000	12.91
10000	38.69
25000	64.13
40000	85.11
50000	89.36

Table 2

K Value	Percentage Error
1	3.05
2	3.57
3	2.92
4	2.91
5	2.87
6	3.05
7	3.09
8	3.11
9	3.16
10	3.23

Test Protocol 3: 5-Fold Cross validation Figures

Table 3

K values	1	2	3	4	5	6	7	8	9	10
Fold 1	3.00	3.57	2.83	3.09	3.02	3.21	3.19	3.39	3.44	3.52
Fold 2	2.98	3.74	2.95	3.22	3.02	3.13	3.08	3.35	3.25	3.45
Fold 3	3.08	3.83	3.12	3.22	3.11	3.22	3.35	3.48	3.52	3.59
Fold 4	3.07	3.66	3.11	3.26	3.10	3.27	3.29	3.49	3.52	3.70
Fold 5	3.18	3.60	2.96	2.99	3.12	3.18	3.21	3.18	3.18	3.28
Mean	3.06	3.68	2.99	3.16	3.07	3.20	3.22	3.38	3.38	3.51
Standard Deviation	0.007	0.10	0.11	0.10	0.04	0.04	0.09	0.11	0.14	0.14

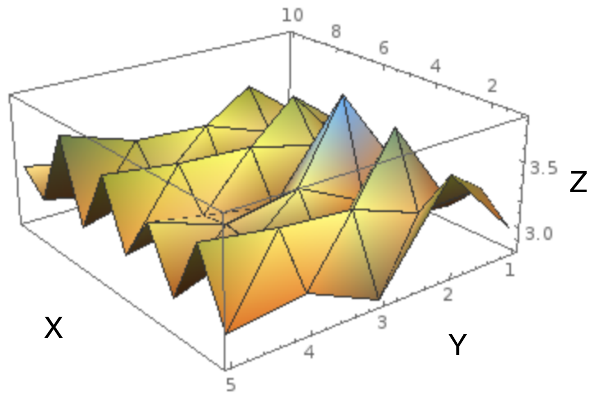


Figure 3.1

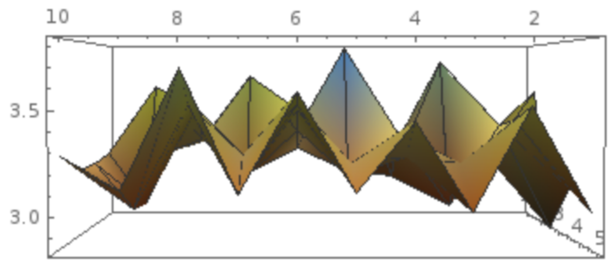


Figure 3.2

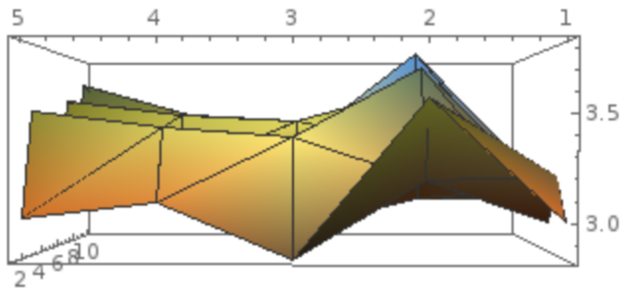


Figure 3.3

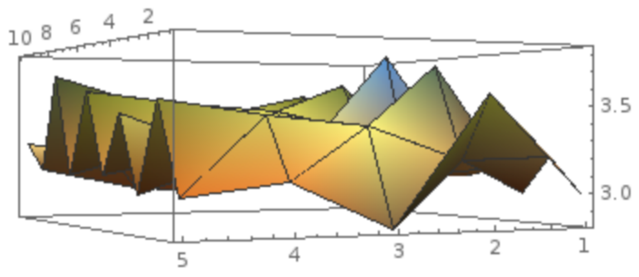


Figure 3.4

Note: all 4 graphs use the same data; they are just angled differently. For the graphs: Z axis is percentage error, and X and Y axis are K value and Number of Folds. Use Table 3 to distinguish axis.

Test Protocol 3: 10-Fold Cross validation Figures

Table 4

K values	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fold 1	2.48	3.30	2.45	2.70	2.63	2.93	2.88	3.00	2.98	3.18	3.17	3.37	3.32	3.45	3.55
Fold 2	3.15	3.82	3.12	3.30	3.28	3.40	3.22	3.55	3.73	3.73	3.70	3.63	3.70	3.78	3.88
Fold 3	3.08	3.78	3.05	3.28	3.23	3.30	3.33	3.48	3.55	3.70	3.62	3.68	3.70	3.80	3.78
Fold 4	2.70	3.32	2.57	2.73	2.60	2.77	2.67	2.85	2.80	2.98	2.97	3.13	3.12	3.18	3.22
Fold 5	3.17	3.67	2.98	3.05	2.98	3.15	3.22	3.35	3.42	3.50	3.60	3.40	3.63	3.53	3.68
Fold 6	2.80	3.60	2.85	3.13	2.87	2.93	3.05	3.18	3.27	3.28	3.35	3.38	3.43	3.60	3.67
Fold 7	2.87	3.23	2.85	2.97	2.88	3.08	2.92	3.00	3.20	3.30	3.30	3.40	3.47	3.50	3.62
Fold 8	3.07	3.77	3.17	3.32	3.20	3.25	3.30	3.45	3.50	3.63	3.70	3.78	3.80	3.83	3.83
Fold 9	3.82	4.15	3.42	3.53	3.78	3.78	3.78	2.85	3.82	3.98	4.05	4.22	4.22	4.35	4.35
Fold 10	2.38	2.87	2.38	2.42	2.42	2.53	2.48	2.57	2.52	2.47	2.55	2.63	2.57	2.73	2.68
Mean	2.95	3.55	2.88	3.04	2.99	3.11	3.09	3.23	2.28	3.38	3.40	3.46	3.50	3.58	3.63
Standard Deviation	0.39	0.35	0.32	0.33	0.38	0.33	0.35	0.36	0.39	0.41	0.41	0.40	0.42	0.41	0.42

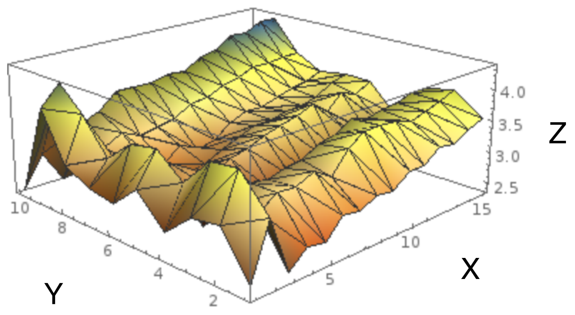


Figure 4.1

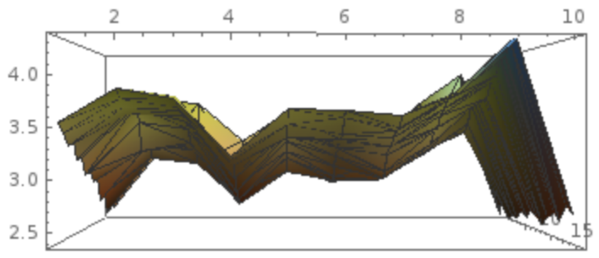


Figure 4.2

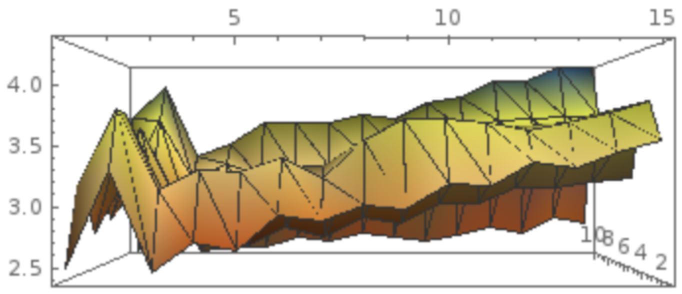


Figure 4.3

Note: all 4 graphs use the same data; they are just angled differently. For the graphs: Z axis is percentage error, and X and Y axis are K value and Number of Folds. Use Table 3 to distinguish axis.