DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Project No. 01:

PROJECT 1: K-NN AND CONDENSED K-NN

submitted to:

Professor Richard Souvenir
CIS 4526/55256: Fundamentals of Machine Learning
Temple University
College of Science and Technology
1947 North 12th Street
Philadelphia, Pennsylvania 19122

September 19, 16

prepared by:

James Novino
Email: tue73681@temple.edu

1. ABSTRACT

Using the Letter Recognition Data Set from the UCI Machine Learning Repository, which contains 20,000 vectored example a basic version of K-NN was implemented in python. This version of K-NN was trained on a randomly subsampled version of the data set. A simplified version of the condensed K-NN was also implemented.

2. ALGORITHM

The 20,000 examples in the UCI Machine Learning Repository where divided into two sets a training set and a test set. The trainset contained the feature vectors in trainX, which was of randomly sampled based on a tuning parameter n. The training set also contained a corresponding list of labels stored in trainY, the label set was also sampled based on n but the same values sampled in trainX would always be sampled from trainY. The testing data was stored into testX and testYCorrect which contained 5000 samples. There were two algorithms implemented in this assignment the first was the Basic K-NN (k-nearest neighbor) algorithm of which the steps where:

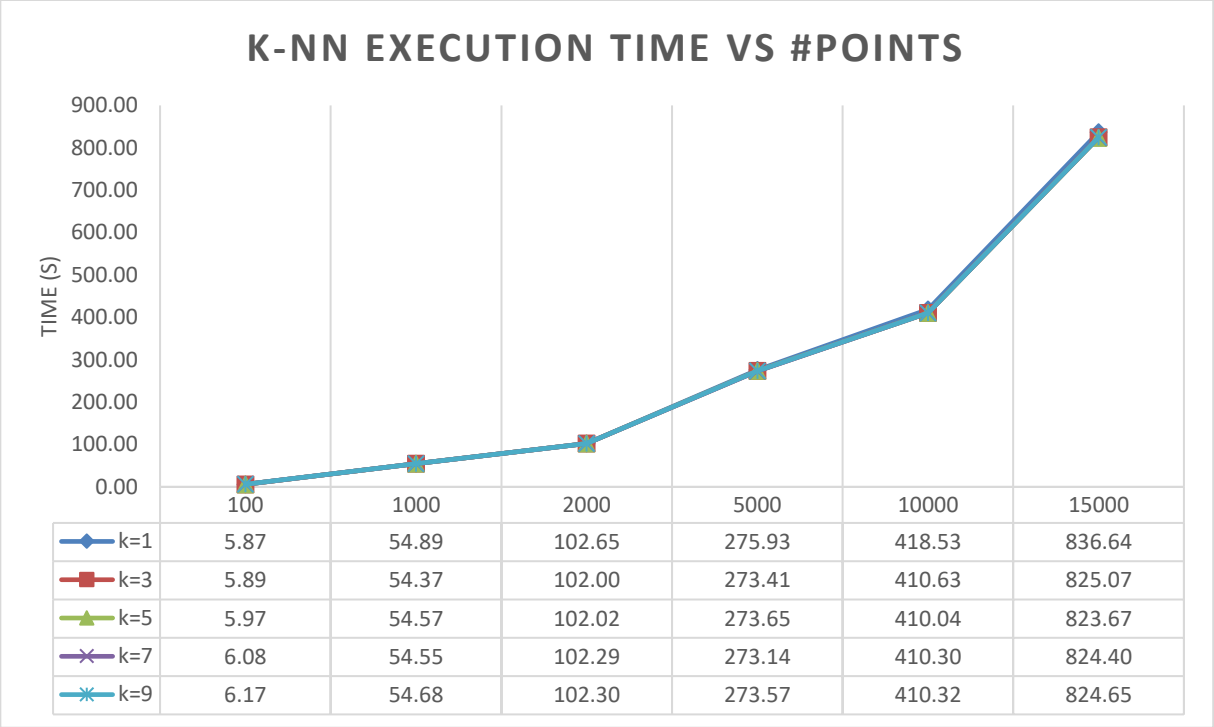
1) Load training data and testing data
2) Calculate distances for every point in trainX versus testX[i]
3) Select smallest k-distances
4) Find the most frequent label in the k-distances
5) Repeat 2-4 for the remaining points in testing data

The second algorithm was C-NN (condensed nearest neighbor) algorithm which is used to minimize the number of points in the training set to improve testing performance while maintaining the same prediction accuracy. The condensing algorithm steps can be seen below.

1) Select one point at random from trainX/trainY
2) run testknn using that one point as the training set
3) check results of testY against trainY
4) take all incorrect points select one at random append to training set
5) return to step 2, continue until returns no incorrect predictions

3. PREFORMANCE

The performance for the basic K-NN algorithm varied significantly but the big-O was O(Cn). The runtime was plotted for 30-tests n=[100,1000,2000,5000,10000,15000], k=[1,3,5,7,9].



	100	1000	2000	5000	10000	15000
k=1	5.87	54.89	102.65	275.93	418.53	836.64
k=3	5.89	54.37	102.00	273.41	410.63	825.07
k=5	5.97	54.57	102.02	273.65	410.04	823.67
k=7	6.08	54.55	102.29	273.14	410.30	824.40
k=9	6.17	54.68	102.30	273.57	410.32	824.65

Figure 1

What is interesting about the chart above due to the way the implementation was done there was not really any change in execution performance with increases in K. This was because all of the distance values were calculate and then sorted so based on K, the K number of labels were simply stored a string and then the maximum letter occurrence was used to assign the test label. The O(Cn) was verified by doing the following math.

$$O(Cn) = O(100) = 5.868\,s$$

$$O(1000) = O(10 * 100) = 58.68\,s$$

$$O(5000) = O(50 * 100) = 293.4\,s$$

The quick calculations show a clear relationship between the calculated run time and the measured run-time, the reason the measured execution time and the calculated execution time has a lot to do with minor implementation details and computer specific optimization in the complier. But overall the measured performance has a clear correlation with the calculated performance. The performance of the algorithm was slow requiring ~14 minutes on the largest test cases, this isn't necessarily a problem for just running K-NN but became more of a problem for the condensing algorithm which needs to run the algorithm n times which requires a non-trivial amount of time. In order to improve performance instead of iteratively calculating the Euclidean distance for each test point vs each point in the training set to an optimized calculation that is much faster.

```
244     for i in range(0,len(trainX)):
245         dst[i] =  np.linalg.norm(test-trainX[i,:])
246     return(dst)
```

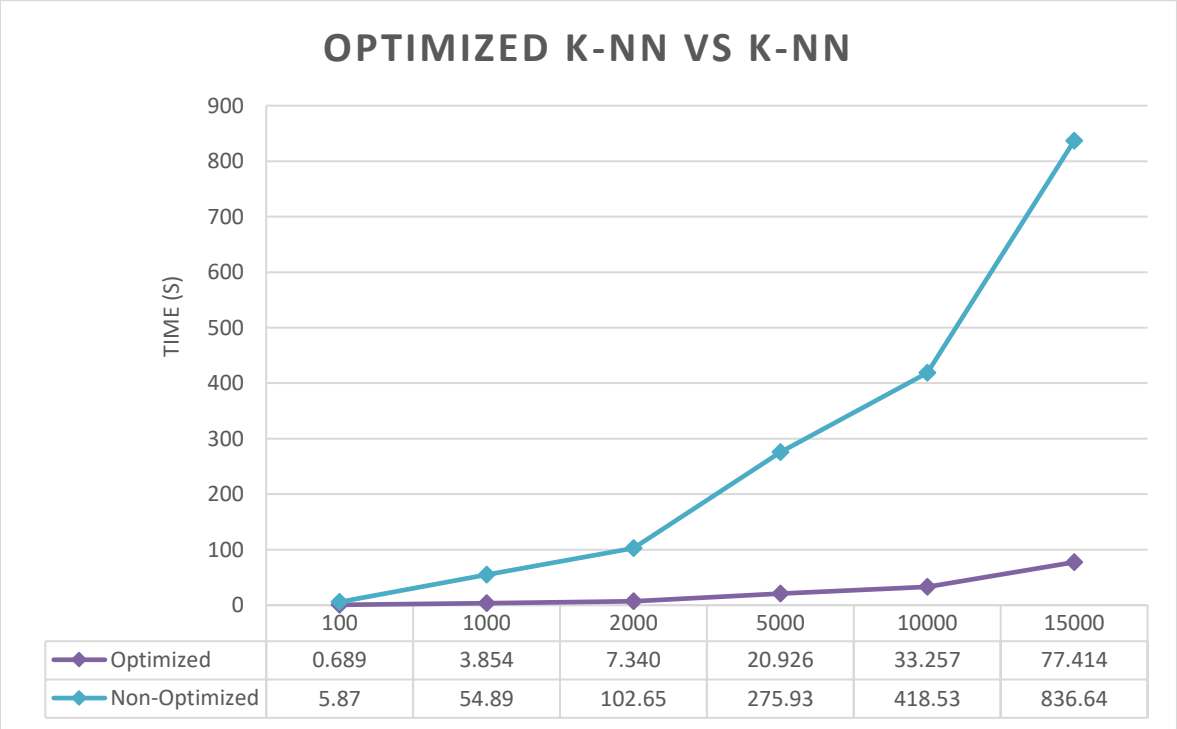
Figure 2

The code above calculates the Euclidean distance between the test point and each point in the training set and returns an array of distances. This was optimized into the code below.

```
238     new_test = np.array([test,]*1)
239     #Calculate distance from testpoint on every point in training set
240     dst = scipy.distance.cdist(trainX, new_test)
241     return(dst.reshape((len(dst)),))
```

Figure 3

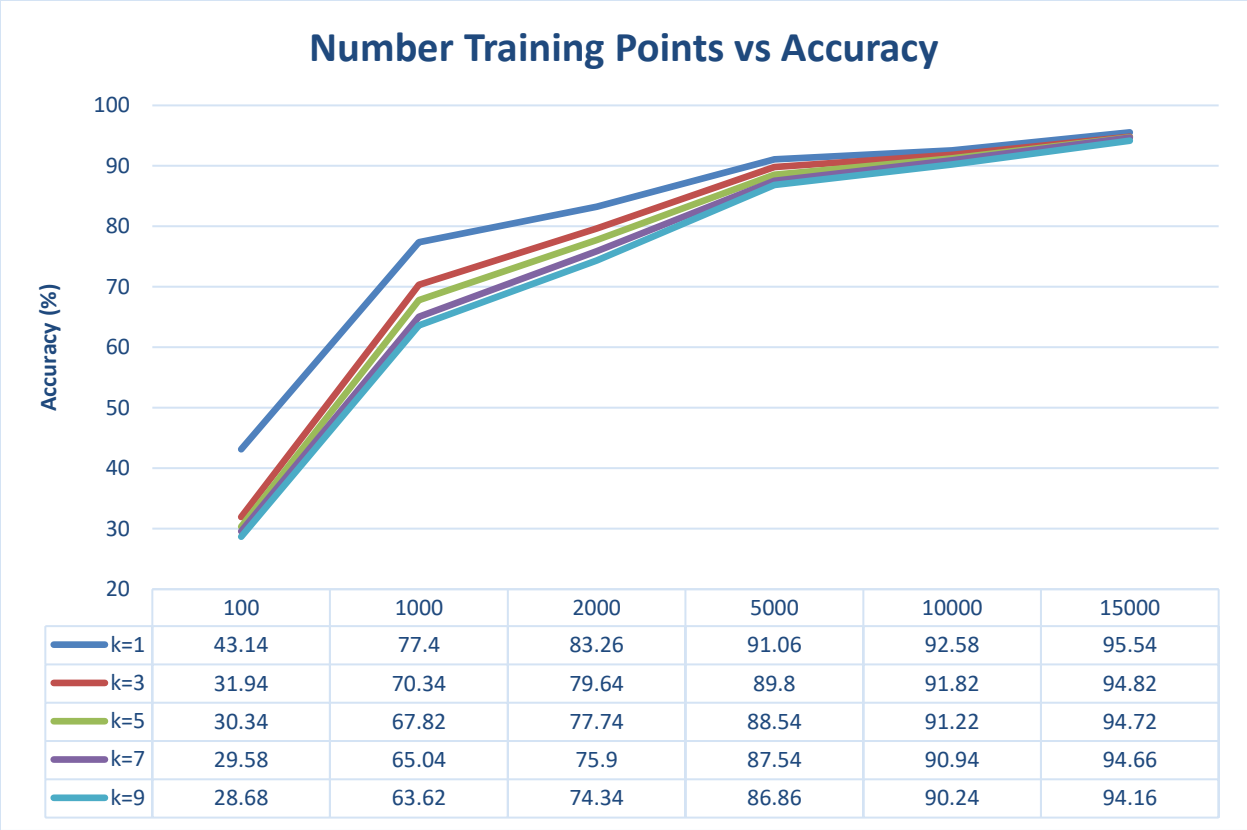
This code reduces the need to iterate over the array for every test point and instead relies on matrix multiplication to calculate the distance the reshaping is only to keep the expected shape of (n,). The performance increase can be seen below.



	100	1000	2000	5000	10000	15000
Optimized	0.689	3.854	7.340	20.926	33.257	77.414
Non-Optimized	5.87	54.89	102.65	275.93	418.53	836.64

Figure 4

The chart above only shows k=1, since based on the previous set of test the execution time was consistent between different values of k. The improvement was roughly a factor of 10, which results in a meaningful difference in execution time for the largest cases from 14 mins to ~1min. The prediction accuracy performance was also plotted for all 30 test cases and can be seen below.

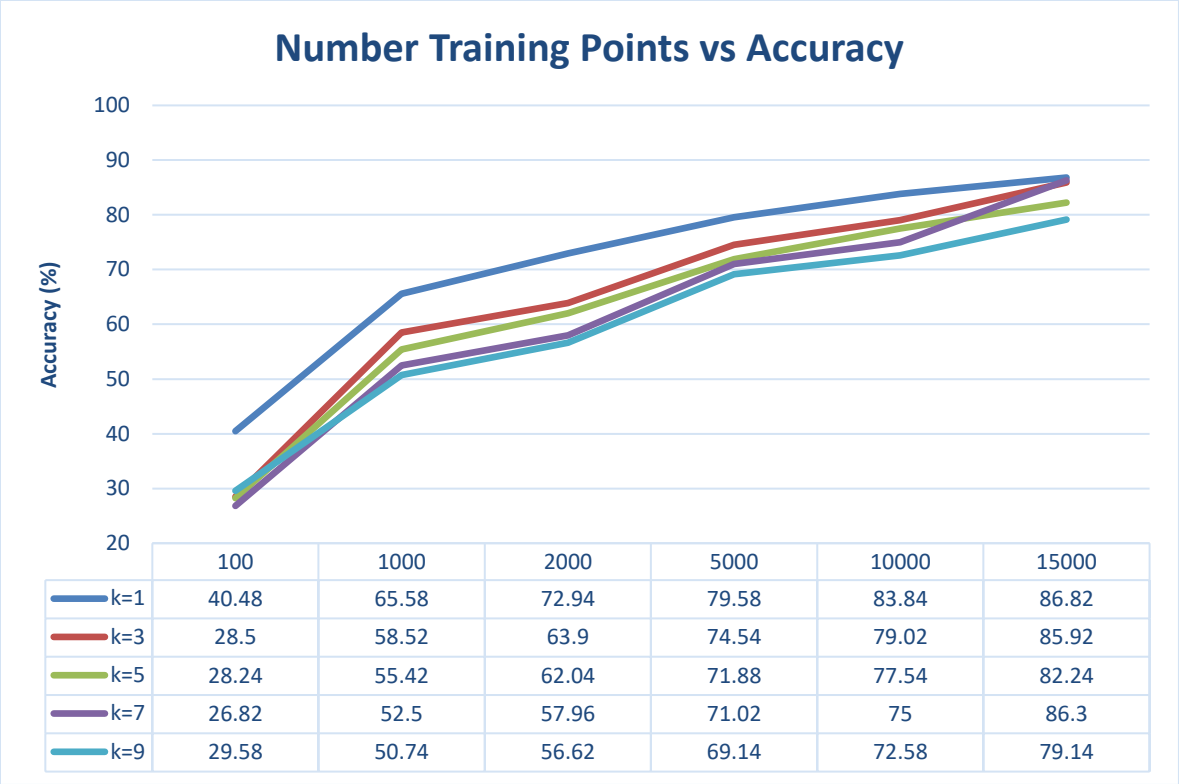


	100	1000	2000	5000	10000	15000
k=1	43.14	77.4	83.26	91.06	92.58	95.54
k=3	31.94	70.34	79.64	89.8	91.82	94.82
k=5	30.34	67.82	77.74	88.54	91.22	94.72
k=7	29.58	65.04	75.9	87.54	90.94	94.66
k=9	28.68	63.62	74.34	86.86	90.24	94.16

Figure 5

The performance of the classifier varied at lower values of N, but trended towards 95% accuracy at the maximum number of training points (15,000). The reason that there is a clear difference in accuracy between N[100,1000,2000,5000] is due to the way the points where sampled. Each of these training sets where sampled randomly from the total available training set. This leads to performance differences in the smaller training sets, if this test were run 100 times the performance would normalize to fit a common trend, with minor differences in performance. The reason there are not huge improvements with increasing k is due to the fact the data had a high entropy and was evenly distributed among labels.

In order to better understand the accuracy of predictions a confusion matrix was generated for the case N=1000, k=1. The confusion matrix can be found in the appendix, but the result of the confusion matrix was that 77.4% of the predicted values were true positives that is a total of 3870 out 5000. The accuracy was calculated by summing the diagonal and then dividing by the total. The condensing algorithm was run on the same number of test cases and the performance was plotted below.



	100	1000	2000	5000	10000	15000
k=1	40.48	65.58	72.94	79.58	83.84	86.82
k=3	28.5	58.52	63.9	74.54	79.02	85.92
k=5	28.24	55.42	62.04	71.88	77.54	82.24
k=7	26.82	52.5	57.96	71.02	75	86.3
k=9	29.58	50.74	56.62	69.14	72.58	79.14

Figure 6

The accuracy performance was not as good as the non-condensed version of k-nn especially looking at higher values of n. However, the time performance was much better, the graph below condensed execution time was plotted.

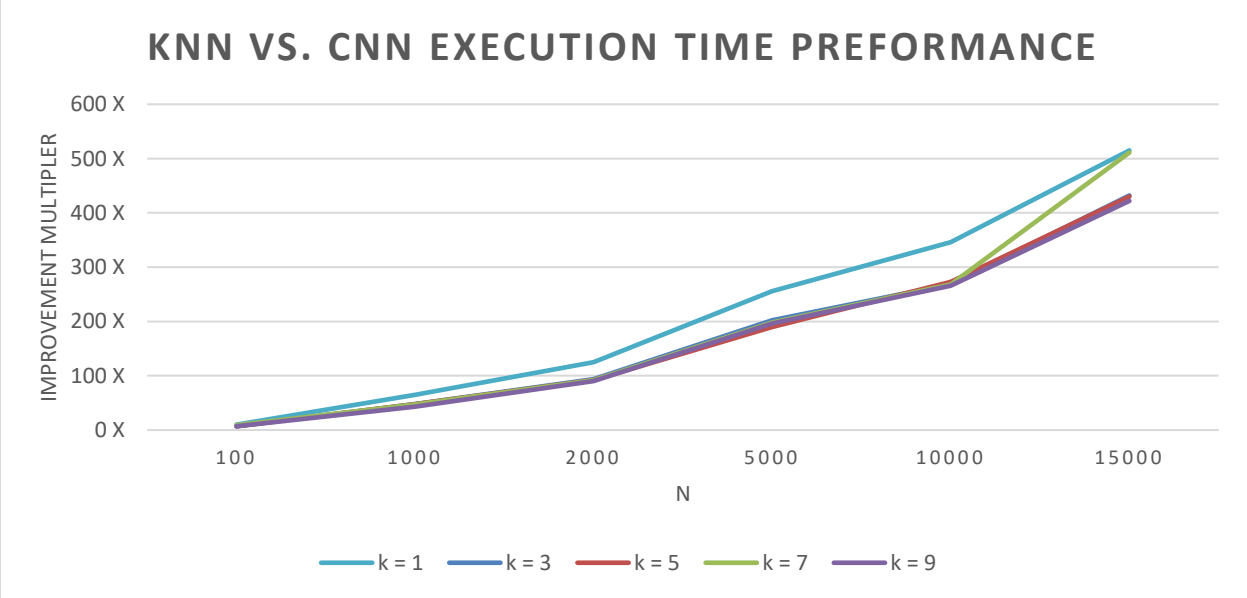
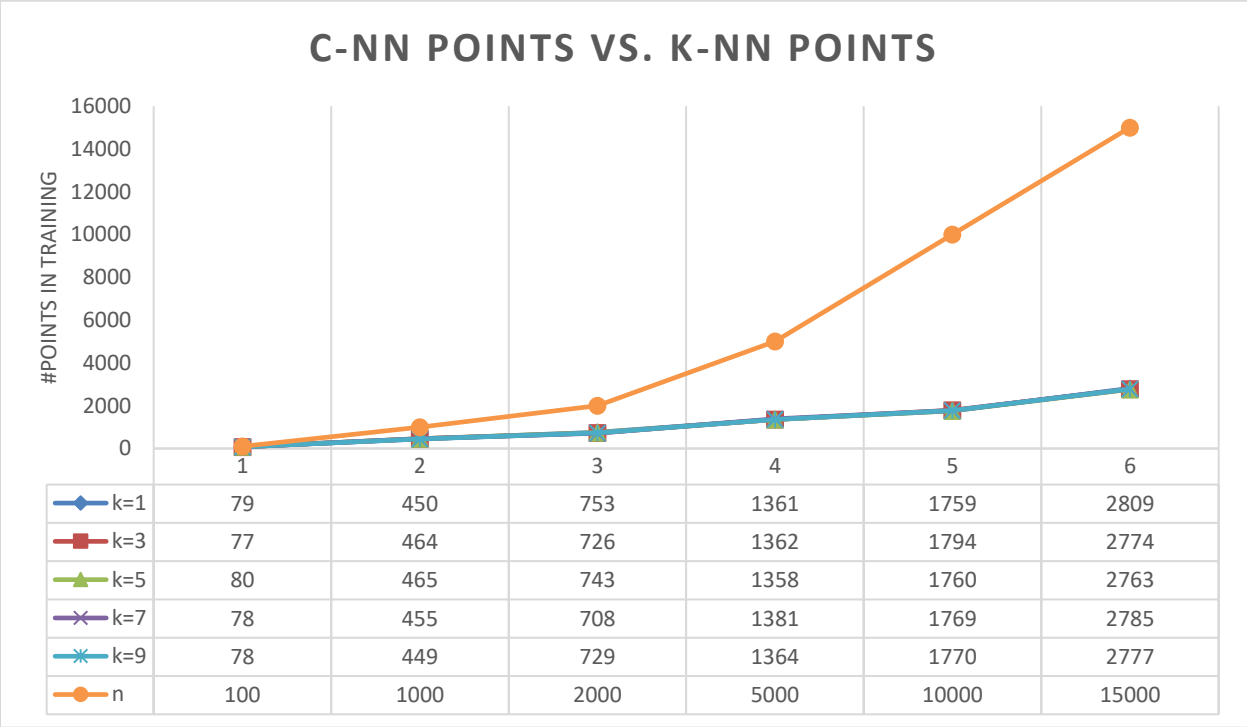


Figure 7

The plot above shows the multiplication factor of improvement between the cnn testing time and the regular

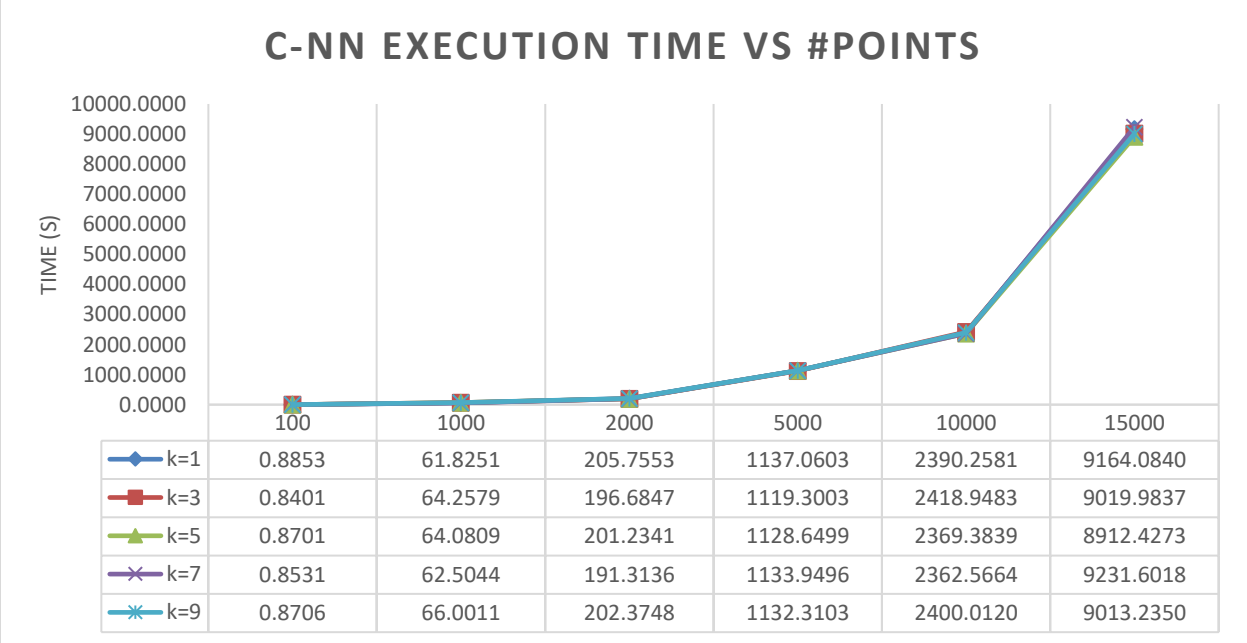
knn testing time. The number of points that the CNN ran against during testing is in the figure below.



	1	2	3	4	5	6
k=1	79	450	753	1361	1759	2809
k=3	77	464	726	1362	1794	2774
k=5	80	465	743	1358	1760	2763
k=7	78	455	708	1381	1769	2785
k=9	78	449	729	1364	1770	2777
n	100	1000	2000	5000	10000	15000

Figure 8

The line n represents the number of points K-NN ran against. As shown by the figure above the improvement in the number of points was huge on order of magnitude of improvement. One of the biggest issues was the execution time required to actually do the condensing.



	100	1000	2000	5000	10000	15000
k=1	0.8853	61.8251	205.7553	1137.0603	2390.2581	9164.0840
k=3	0.8401	64.2579	196.6847	1119.3003	2418.9483	9019.9837
k=5	0.8701	64.0809	201.2341	1128.6499	2369.3839	8912.4273
k=7	0.8531	62.5044	191.3136	1133.9496	2362.5664	9231.6018
k=9	0.8706	66.0011	202.3748	1132.3103	2400.0120	9013.2350

Figure 9

The execution time for condensing has an interesting trend with the time required for condensing increasing exponentially with increase in the numbers of n.

4. CONCLUSION

Both versions of the nearest neighbor algorithm had fairly successful performance with larger training sets. The K-NN algorithm had good performance as far as accuracy with higher values of n, the accuracy was approaching ~95% for prediction accuracy on the test set which is very good considering the limited number of data points. The condensed nearest neighbor didn't have as good accuracy results but had much better performance as far as execution time to make predictions which can be seen in figure 6 and 7. The accuracy for prediction for C-NN was approaching ~87% which is almost 10% less than K-NN, obviously the tradeoff was prediction time. Without optimization the original k-nn was taking roughly 1000+ secs to predict on the largest set, but after optimization was roughly 6s, which represented a significant improvement and was mainly accomplished by minimizing indexing and copying of arrays in the implementation which had huge effects on performance. The condensing algorithm had a large amount of time required to actually run the condensing algorithm which was running with a worst case of $O(n^3)$. Improvements can be made to the condensing algorithm performance by caching the distances calculated as training points are appended to the training set to prevent additional work and improving execution time greatly. Overall, while C-NN was supposed to be more efficient it turns out that K-NN can be optimized without the use of KD-Trees to have a excellent performance even on large data sets with high dimensions.

5. APPENDIX

Predicted \ Actual	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	182	6	0	0	0	1	1	2	0	0	0	4	0	0	0	0	3	0	3	0	0	0	0	0	0	0
B	0	136	0	1	7	0	5	6	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
C	3	0	140	0	6	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	1	18	0	127	0	3	0	14	2	3	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	6	4	0	141	1	7	1	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	6	0	0	4	133	0	1	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	8	6	3	10	3	151	3	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	13	0	1	1	2	3	113	0	1	11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	0	2	1	2	1	6	1	0	185	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
J	0	1	0	0	3	3	0	4	14	152	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K	0	8	1	3	4	1	7	13	0	0	126	0	2	0	0	0	1	11	1	0	0	1	0	17	1	1
L	4	3	0	2	4	0	0	2	0	1	0	182	0	0	0	1	1	1	1	0	0	1	0	2	0	0
M	0	1	0	0	0	0	1	9	0	0	2	0	165	1	0	0	0	0	2	0	2	1	0	0	1	0
N	0	10	0	5	0	0	0	14	0	0	0	0	5	152	0	0	1	2	0	0	1	0	0	0	0	0
O	0	2	0	1	0	1	4	4	0	0	0	0	0	4	145	4	18	1	0	0	5	0	0	1	0	0
P	0	6	0	7	0	12	6	1	0	0	0	1	0	0	0	163	0	0	1	0	0	1	0	0	2	0
Q	3	4	0	1	12	1	9	0	0	0	1	1	0	1	0	0	163	0	1	0	0	2	0	0	2	2
R	0	17	0	9	1	0	2	13	0	0	11	1	0	2	1	0	0	149	3	1	0	0	0	7	0	0
S	3	24	0	1	12	0	3	3	1	1	1	1	2	0	0	1	1	3	123	1	0	0	0	0	0	6
T	0	5	0	0	4	5	2	4	0	0	1	0	1	0	0	1	0	0	0	148	0	0	0	3	24	1
U	0	1	1	0	0	0	3	10	0	1	0	0	2	1	5	0	0	0	0	0	187	1	0	1	0	1
V	0	4	0	0	0	0	8	0	0	0	1	0	1	0	0	1	2	0	0	0	2	141	4	0	10	0
W	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	159	0	1	0
X	0	5	0	3	17	0	0	0	1	2	17	2	0	0	1	0	0	7	0	3	1	0	0	123	1	7
Y	0	0	0	3	0	1	2	4	0	1	0	0	1	0	0	2	2	0	0	24	0	0	1	1	131	0
Z	0	7	0	0	11	1	0	0	3	1	1	0	0	0	0	0	0	0	6	1	1	0	0	7	0	153

Table 1: Raw Confusion Matrix