ICS485 MACHINE LEARNING

ASSIGNMENT01 - K NEAREST NEIGHBOR

OCTOBER 12, 2019

GROUP 04
Faris Hijazi - Baraa Fael – Mohammed Al Sayed

Table of Contents

I. Introduction:

In this assignment we introduce our implementation of K-nearest neighbor (KNN) algorithm using Mnist database. Mnist is a huge collection 28x28- pixel images of hand-written digits between zero and nine. In our implementation, we first process the figures and convert them to array of vectors where each of which is 784 pixel long. Then, we divide the array of figure into two sets, training data set and test data set. Afterwards, to implements and train our model, we further divide the training data set into two sets, namely, trainX and valX, each of which has a corresponding array of the same length indicating the label of each vector (figure) in the array.

Throughout the experiment, we implement the KNN algorithm with different Lp norms (L1 and L2) and K values. We then run our implementation variants on the trainX and valX sets to check for the values of the hyperparameter that yield the best score. Once we select the best values for our hyperparameters, we run our algorithm on the test data set and show the results using confusion matrix. Please do note that the L_infinity distance and weighted KNN are all implemented.

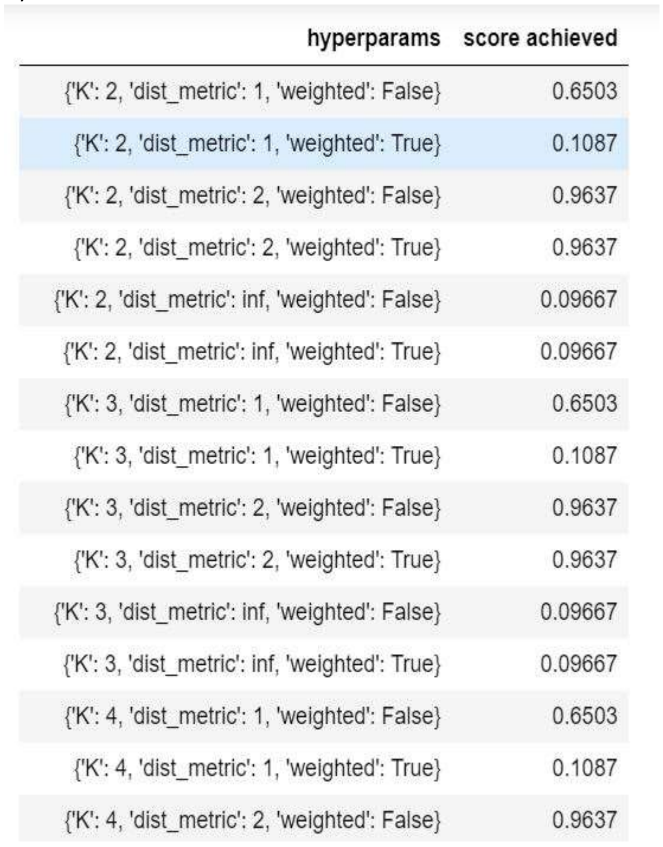
II. Document Outline:

In this document, we attempt only to show the output of our implementation and to provide answers to the questions appended with the assignment. For the KNN variants' implementation and further details, the reader must refer to the notebook (.ipynb file) attached with this report.

III. Hyperparameter Combinations and Scores:

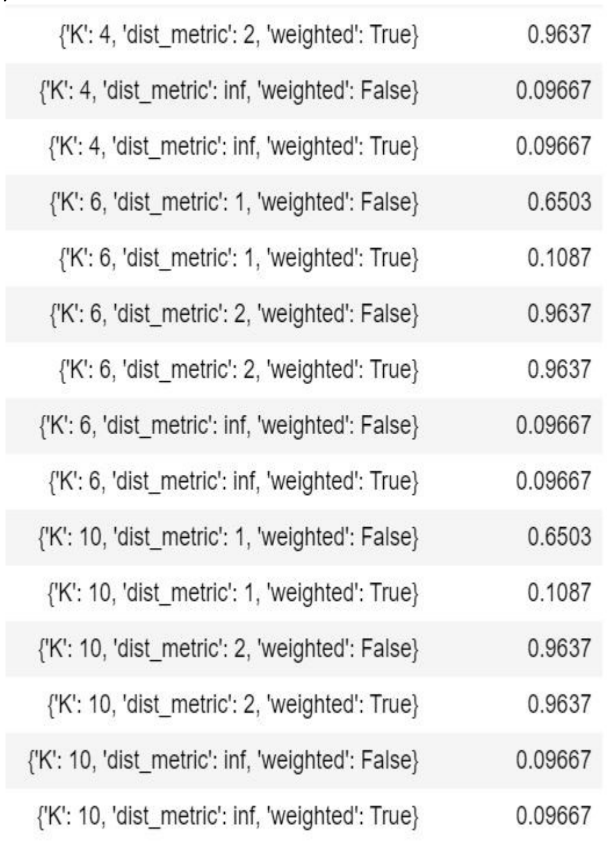
Following are different combinations of K values (K= 2,3,4,6), Lp norms (p = 1,2, infinity) and weighted/unweighted technique and their corresponding scores:

1)



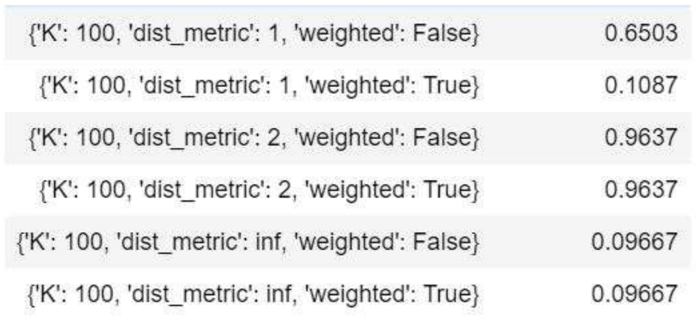
hyperparams	score achieved
{'K': 2, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 2, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 2, 'dist_metric': 2, 'weighted': False}	0.9637
{'K': 2, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 2, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 2, 'dist_metric': inf, 'weighted': True}	0.09667
{'K': 3, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 3, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 3, 'dist_metric': 2, 'weighted': False}	0.9637
{'K': 3, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 3, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 3, 'dist_metric': inf, 'weighted': True}	0.09667
{'K': 4, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 4, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 4, 'dist_metric': 2, 'weighted': False}	0.9637

2)



{'K': 4, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 4, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 4, 'dist_metric': inf, 'weighted': True}	0.09667
{'K': 6, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 6, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 6, 'dist_metric': 2, 'weighted': False}	0.9637
{'K': 6, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 6, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 6, 'dist_metric': inf, 'weighted': True}	0.09667
{'K': 10, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 10, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 10, 'dist_metric': 2, 'weighted': False}	0.9637
{'K': 10, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 10, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 10, 'dist_metric': inf, 'weighted': True}	0.09667

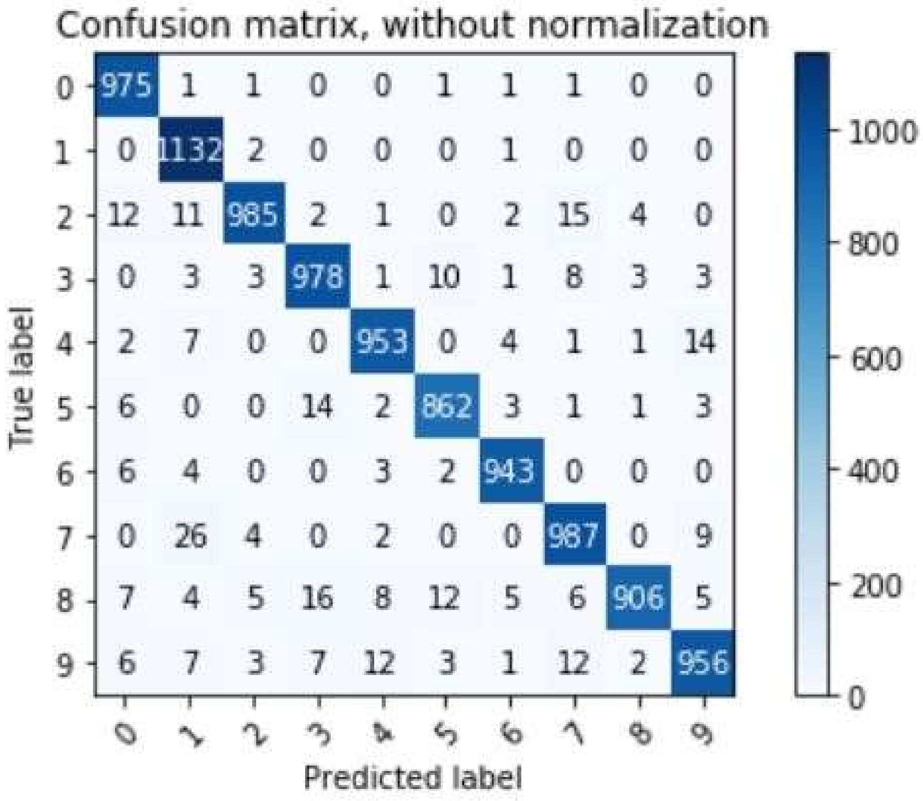
3)



{'K': 100, 'dist_metric': 1, 'weighted': False}	0.6503
{'K': 100, 'dist_metric': 1, 'weighted': True}	0.1087
{'K': 100, 'dist_metric': 2, 'weighted': False}	0.9637
{'K': 100, 'dist_metric': 2, 'weighted': True}	0.9637
{'K': 100, 'dist_metric': inf, 'weighted': False}	0.09667
{'K': 100, 'dist_metric': inf, 'weighted': True}	0.09667

IV. Test Set confusion matrix:

Following is the confusion matrix with of the test set (final test). We can see that 7 is sometimes confused with 1, 8 with 3, 4 with 9 etc. However, the overall prediction is fairly good.



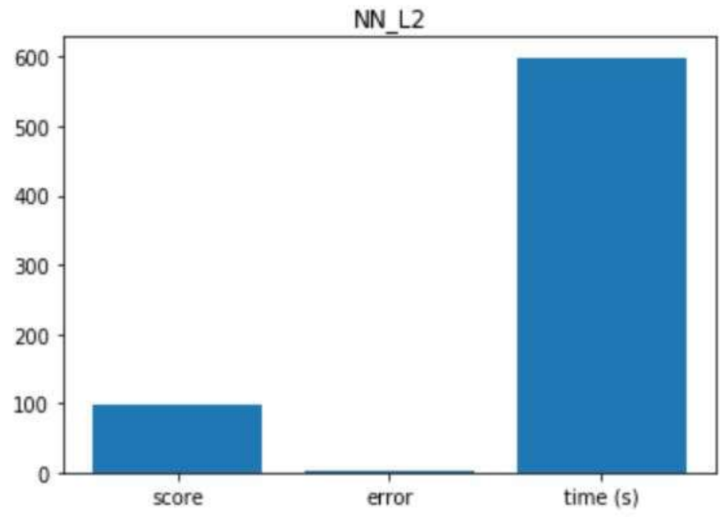
	0	1	2	3	4	5	6	7	8	9
0	975	1	1	0	0	1	1	1	0	0
1	0	1132	2	0	0	0	1	0	0	0
2	12	11	985	2	1	0	2	15	4	0
3	0	3	3	978	1	10	1	8	3	3
4	2	7	0	0	953	0	4	1	1	14
5	6	0	0	14	2	862	3	1	1	3
6	6	4	0	0	3	2	943	0	0	0
7	0	26	4	0	2	0	0	987	0	9
8	7	4	5	16	8	12	5	6	906	5
9	6	7	3	7	12	3	1	12	2	956

V. Questions and Answers:

1. What is the error rate on the validation set for NN_L2?

    - Error rate for **NN_L2** is **2.6%.** Below, we plot the score, error and time for the sake of comparison and visual illustration:

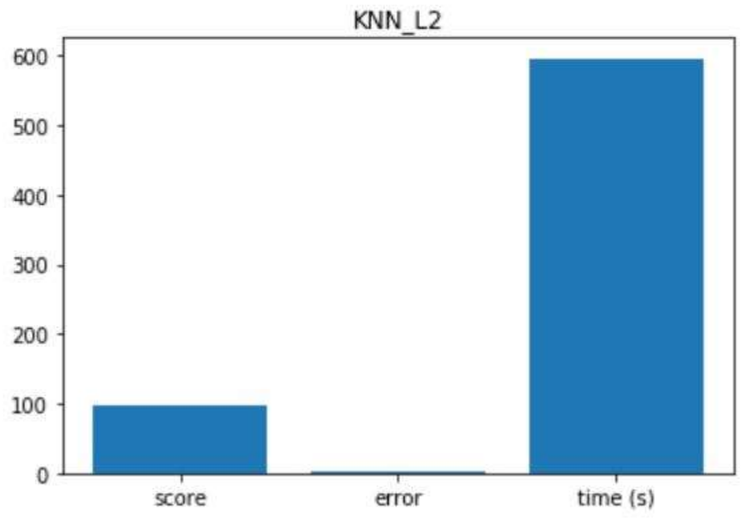
```
['name: NN_L2', 'score: 0.974', 'error: 0.026', 'time: 599.298']
```



2. What is the best error rate on the validation set for KNN_L2?

    - Error rate for **KNN_L2** is **3.6%.** Below, we plot the score, error and time for the sake of comparison and visual illustration:

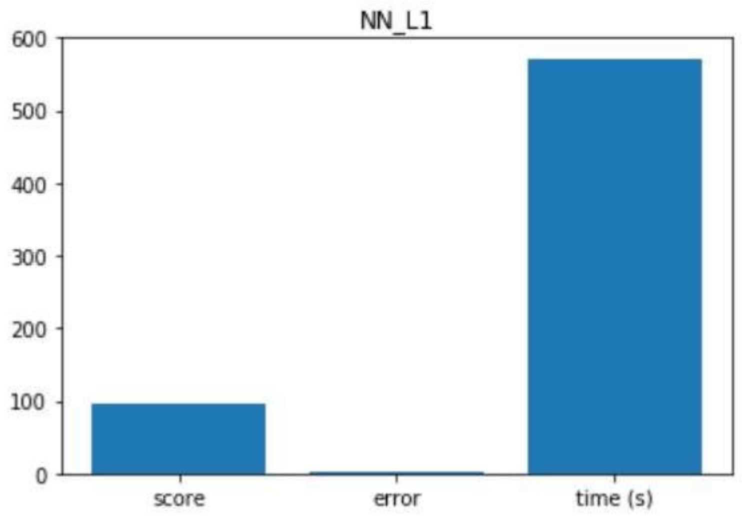
```
['name: KNN_L2', 'score: 0.964', 'error: 0.036', 'time: 595.991']
```



3. What is the error rate on the validation set for NN_L1?

   - Error rate for **NN_L1** is **2.6%.** Below, we plot the score, error and time for the sake of comparison and visual illustration:

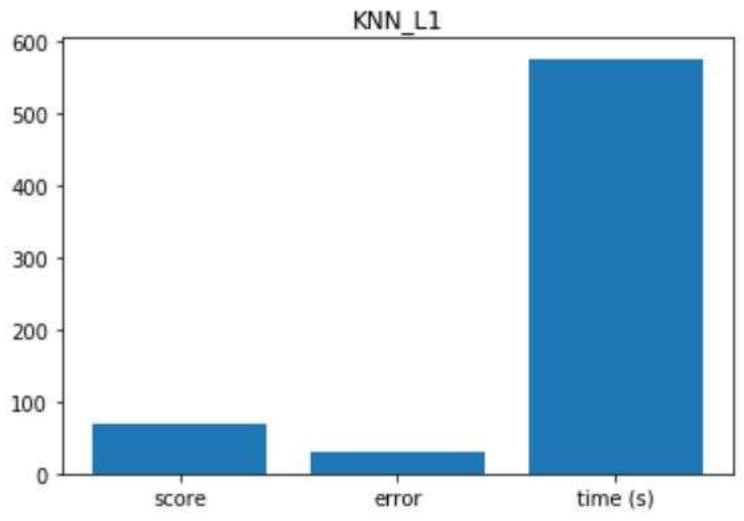
```
['name: NN_L1', 'score: 0.974', 'error: 0.026', 'time: 571.526']
```



4. What is the best error rate on the validation set for KNN_L1?

   - Error rate for K**NN_L1** is **30.1%.** Below, we plot the score, error and time for the sake of comparison and visual illustration:

```
['name: KNN_L1', 'score: 0.699', 'error: 0.301', 'time: 575.489']
```



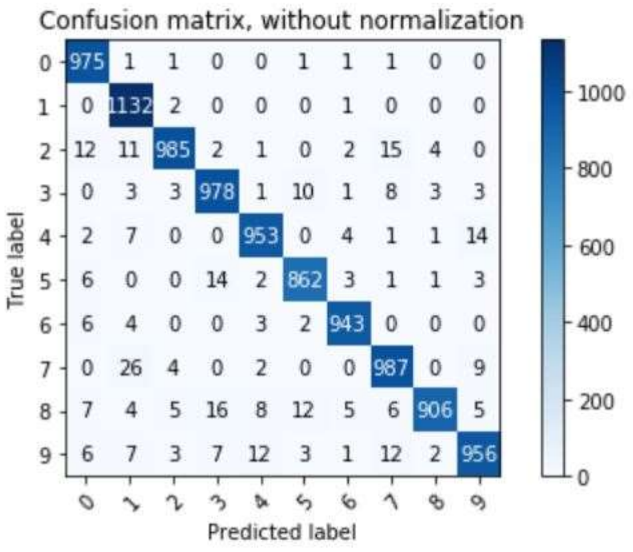
5. What is the error rate on the test set?

   - From the above hyperparameter combinations and score section, the best score we got on the test set is 96.77% leaving us with an **error rate** of **3.23%**
   - This was achieved using the below set of hyperparameters. It is also worthwhile to mention that there were combinations that had a very close error and score rates.

```
best_params {'K': 6, 'dist_metric': 2, 'weighted': True}
```

6. Which digit class has the most errors? And, with which class does it get the most confused? Do you have ideas to fix this problem?

   - The class with the most errors is class 8 (digit 8). It was confused the most with other digits that is was confused at least five times with any other class.
   - However, number 7 had highest confusions across the classes and that is with number 1 as shown below:
   - How to fix it: this happens because there are many *features* (pixels) that overlap between the two (1 and 7), they both have a vertical line.
     One way to moderate this is to use less overlapping features in our model. One example could be the sum of pixels in each column of the 28x28 pixel images, this way the nature of the features themselves has less overlap (because the number of pixels at the top for a 7 will be much higher than those of a 1) (but may cause other issues with other numbers like the 9 and the 7).



	0	1	2	3	4	5	6	7	8	9
0	975	1	1	0	0	1	1	1	0	0
1	0	1132	2	0	0	0	1	0	0	0
2	12	11	985	2	1	0	2	15	4	0
3	0	3	3	978	1	10	1	8	3	3
4	2	7	0	0	953	0	4	1	1	14
5	6	0	0	14	2	862	3	1	1	3
6	6	4	0	0	3	2	943	0	0	0
7	0	26	4	0	2	0	0	987	0	9
8	7	4	5	16	8	12	5	6	906	5
9	6	7	3	7	12	3	1	12	2	956

VI. Optimization (vectorization)

In terms of performance and vectorizing the code, we first attempted to make the KNN function work for a single testpoint at a time (`pred=KNN_predict(evalx)` where `evalx` is a single image and `pred` is a single prediction), while this was easier to implement, it did decrease performance.

We implemented this completely without looking at any resources.

We then attempted to vectorize the code, and we had to change the single-prediction implementation, and now `KNN_predict()` predicts multiple testpoints at a time. Now the bottleneck were the for-loops in the `KNN_predict()`, the ones that calculated the distances. After some research, I came across some resources [1](https://ljvmiranda921.github.io/notebook/2017/02/09/k-nearest-neighbors/) [2](https://medium.com/@souravdey/l2-distance-matrix-vectorization-trick-26aa3247ac6c) that had a no-loop implementation:

```py
dists = -p * np.dot(a, b.T) + np.sum(b**p, axis=axis) + np.sum(a**p, axis=axis)[:, np.newaxis]
```

where `axis = 1` for multiple predictions and p is the LP metric (L1, L2...).

This comes from the following formula:

$$d_2(\mathbf{I}_1, \mathbf{I}_2) = ||\mathbf{I}_1 - \mathbf{I}_2|| = \sqrt{||\mathbf{I}_1||^2 + ||\mathbf{I}_2||^2 - 2\mathbf{I}_1 \cdot \mathbf{I}_2}$$

VII. Contribution:

Contribution	
Faris Hijazi	65% hyper param testing and optimizing
Baraa Fael	35% kdtree and documentaiton
Mohammed Alsayed	0%

VIII. References:

- [1](https://ljvmiranda921.github.io/notebook/2017/02/09/k-nearest-neighbors/)
- [2](https://medium.com/@souravdey/l2-distance-matrix-vectorization-trick-26aa3247ac6c)