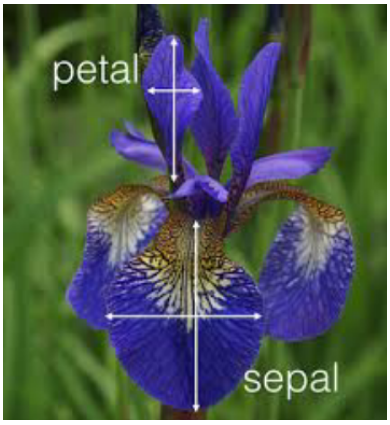
Classification project

Andreas Tufte
Martin Kraft

April 26, 2021



Department of Electronic Systems

Summary

This report is a part of the evaluation in the course TTT4275 Estimation, Detection and Classification hosted by the Department of Electronic Systems at NTNU Trondheim. The course aims to give insight into the fundamental statistical and algorithmic methods for signal processing and data analysis.

This report focus on the classification part of the course, and will apply some of the theory on the given tasks. By using both a discriminant and template based classifier on two different data sets, we achieved results giving insight into the different classifiers attributes in relationship to the attributes of the data set. These exploration is all a part of what is known as supervised learning.

First, we describe an introduction to classification and the tasks, present relevant theory, implementation in Python and provide results.

Contents

1 Introduction

The project is of educational purpose. Meaning that the goal is to archive insight into the classification types and data set attributes, rather than optimal performance. In general however, classification serves a great interest to the society outside academia. Reliable classification is fundamental to several fields like, robotics, autonomous vehicle and medical imaging. Humans have evolved a strong ability to classify objects and phenomena in their environment, however we struggle with speed and large data sets. By devising strong classification tools, more accurate and faster performance is possible.

We have in the report collected necessary theory in the theory section 2. This theory is then used in the two parts giving the experimentation. The first task, concerning the classification of Irises, with both the implementation and results, is included in section 3.

The last task uses an neareast-neighbour-classifier to classify handwritten numbers from the MNIST dataset. The implementation and results of this is described in section 4. Lastly, a conclusion of the results is found in section 5.

The given project description is found in [1]. Some of the code used to implement the classifiers is found in the appendix, it may also be found in the github repository[1].

[1] https://github.com/Martinakraft99/TTT4275_Project_Classification_1_01

2 Theory

Contents relevant for both tasks are given in section 2.1 about supervision. In section 2.2 and 2.3, we discuss theory applied for implementing the linear classifier in the Iris task. Lastly, section 2.4 and 2.5 discuss nearest-neighbor classifier and methods to reduce training data relevant to the task with handwritten numbers.

2.1 Supervision

A fundamental distinction within machine learning is between supervised and unsupervised learning. Supervised learning is about giving a labeled set of data to and algorithm, and asking the algorithm to find a way distinguishing between the classes corresponding to the labels. After the algorithm has been trained on a set of data, the performance may be tested on another set of data. Classification fall naturally under the supervised learning section. In unsupervised learning, an algorithm is design to take in unlabeled data, and find connections between the data points. The most common method is called clustering.

2.2 Linear classifier

A linear classifier is discriminate classifier, which aims to separate classes by some linear decision border. The linear classifier is based upon the discriminant function, g, defined as

$$g(x) = \mathbf{W}x + \omega_0, \tag{1}$$

where x is a single column vector of size $1 \times D$, representing a single sample with D features. The matrix $\mathbf{W}$, is of dimensions $C \times D$, where C is the number of classes. It represents the weighting given to the sample features per class. ω_0 is an offset-vector of size $1 \times C$.

The discriminant function is a vector of dimensions $1 \times C$, where each element represents a value the corresponding class. The decision rule implemented is that a sample, x, belongs to the class corresponding to the largest entry in g. By tuning $\mathbf{W}$ and ω_0, we wish to obtain a g giving correct predictions (label).

A common way of tuning, is using the mean square error (MSE), which is defined as

$$\text{MSE} = \frac{1}{2}\sum_{k=1}^{N}(g_k - t_k)^\top(g_k - t_k), \tag{2}$$

where $k = 1, 2...N$ is the number of samples used to correct, and t_k is the true label vector. The true label vector is a column vector of binary values, having 1 at the index of the true label of sample x and 0 elsewhere.

In order to compare g and t, we would ideally transfer g into a binary vector, using the *heaviside* function, however, as we want continuous derivatives, we use the *sigmoid* function as a close approximation. The *sigmoid* is defined as

$$g_{ik}(z) = \frac{1}{1 + e^{z_{ik}}} \tag{3}$$

where z_k is the previously called g_k. To minimize MSE, it is common to use an numerical optimization algorithm to update $\mathbf{W}$, such as gradient descent.

$$\mathbf{W}(m) = \mathbf{W}(m-1) - \alpha \nabla_{\mathbf{W}} \text{MSE} \tag{4}$$

where $\alpha \nabla_{\mathbf{W}} \text{MSE}$ is the gradient of MSE in the direction of $\mathbf{W}$ and m the iteration number. The constant α is often refereed to as the learning rate. The value of this determine how fast the algorithm will converge to an optimal $\mathbf{W}$.

2.3 Linear separability

An important quality for the performance of a linear classifier, is the linear separability of the data. If the data is linearly separable, then one may separate the classes using a linear hyperplane, as shown in in 1.

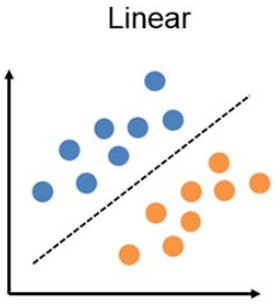


Figure 1: Two classes separated by a decision border. Note that in 2D, the hyperplane becomes a line. Image from [2].

If the dataset is almost linearly separable, then a linear classifier will not perform perfectly, but could have have an acceptable error rate. Otherwise a nonlinear classifier, or a more advanced, such as an SVM, may be used.

2.4 KNN-classifier

A nearest-neighbor (NN) classifier is a supervised classifier with training data needed at runtime. New data is classified based on the class of their nearest resembles to existing data. The intuition is shown in figure 2 with n as number of features $\{q_i\}$ in the data.

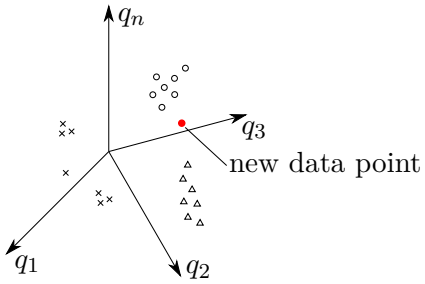


Figure 2: Nearest neighbor classifier. In this example, the new data point resembles features from that of the label circle.

The distance metrics used in our classifier is the euclidean distance

$$||\Delta \mathbf{q}|| = \sqrt{\sum_{i}^{n} (\Delta q_i)^2}, \tag{5}$$

where $\Delta \mathbf{q}$ denotes a difference in two points. As the distances is relatively compared, it is beneficial to work with the euclidean distance squared (not performing the square root). This saves computational time.

An extension to the nearest-neighbor classifier is the K-nearest classifier (KNN) where the K nearest points are identified, and the mode of these classes is used as classification.

2.5 Clustering of reference data

To save computational time, it is common to perform clustering of the reference data. Groups of data points with similar features and class is approximated to single-cluster centers. This reduces the number of reference points, but at the cost of loosing accuracy.

Formally, we want to group each class into M sets $\{S_i\}$ such that the within-cluster variance of points $\mathbf{q}$ to the cluster center μ_i is minimized

$$\mathbf{S} = \arg\min_{\mathbf{S}} = \sum_{i=1}^{M} \sum_{\mathbf{q} \in S_i} ||\mathbf{q} - \mu_i||^2. \tag{6}$$

There already exists libraries for performing clustering, and we used `Kmeans` from the library `sklearn` in Python. The algorithm for finding a local minimum is based on algorithm 1 given in appendix B.

3 Linear Classification of Irises

In this task, we use a linear classifier to classify a dataset of Irises into three different classes based upon some of their features.

The three classes of interest are called Iris Setosa, Iris Versicolor and Iris Virginica. The features used to classify are their sepal and petal dimensions, see front page. The distribution of the features among the classes for this given dataset, is shown in the histogram in figure 3.

The first task is focused on the basics of a linear classifier, and attributes of a data set, while the second is more focused on linear separability of individual features. All code is implemented in python.

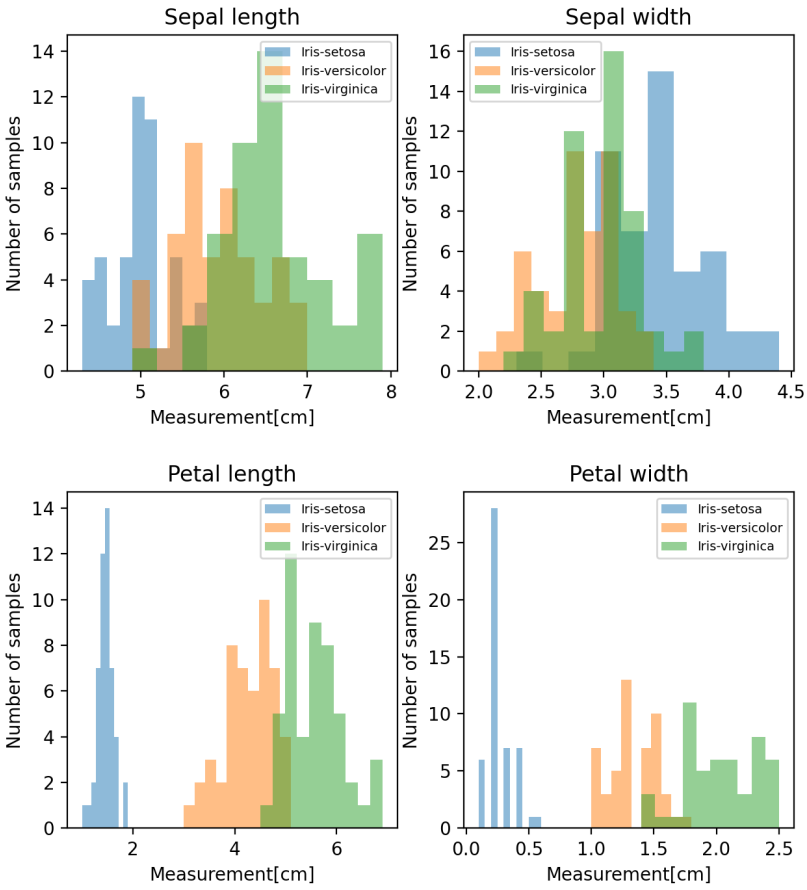


Figure 3: Histogram of Iris data set

3.1 Implementation

The task was implemented using python, especially the numpy-library. Loaded the dataset and split it into a training- and test set. The classification itself

was performed by implementing (1) and picking the label corresponding to the largest element in g. This was then compared to the true labels, and, if wrongly classified, appended to a vector of wrong classifications. This vector was then fed back through the procedure to update the **W**-matrix, shown in (4). Used an break condition on the Euclidean norm of **W**, the number of iterations and the size of the wrong vector, to break the update loop.

The obtained weight matrix was then used to classify the samples in the test set.

3.2 Results

In the first task the data was spilt with the first 30 samples of each class in the training set, and the remaining 20 in the test set. Using a learning rate of $\alpha = 0.01$ resulted in a error rate at 0% and 5% for the training and test set respectively. From the resulting confusion matrix for the training set shown in 1 and test set shown in 2, it clear that the classifier has the most issues classifying the *Versicolor* and *Virginica*.

Changing the sets, so that the last 30 samples is used for training, and first 20 for testing resulted in an overall worse performance, with error rates at 2% and 0% for training and test respectively, and results as shown in table 3 and 4.

It is expected that the classifier has the highest error for *Versicolor* and *Virginica*, as it is apparent from the histogram 3 that these are the two classes with the most overlapping features. Why the first split is performed worse than the second is probably due the some difference in linear separability between subsets in the dataset.

For the second task, the sepal width feature was removed due to it had the most overlap across the three classes, as seen in 3.

This resulted in an error rate of 6.7% for the training- and 1.7% for the test set. The resulting confusion matrices 5 and 6 is similar to the ones from the first experiment.

The second most overlapping feature is the sepal length. Removing this yields an error rate of 6.7% for the training set, and 3.3% for the test set, with confusion matrices shown in 7 and 7.

Finally removing the petal length feature, we are left with only the petal width. This gives a error rate of 13.3% and 10% for the training- and test set respectively. Results shown in 9 and 10.

Looking at the confusion matrices from the four experiments, it is clear that the classifier performs worse when features are removed, even though those features were less linearly separable than the remaining. This might be surprising, as one could expect that the less linearly separable features only becomes noise degrading the performance of the close to linearly separable features. However, as long as no features are completely linearly separable, this is not the case. Rather, it seems that one of the mantras within

True Labels			
Setosa	30	0	0
Versicolor	0	30	0
Virginica	0	0	30
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 1: Confusion matrix training on first 30 samples.

True Labels			
Setosa	19	1	0
Versicolor	0	19	1
Virginica	0	1	19
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 2: Confusion matrix testing on last 20 samples.

classification, namely ”The more data the better”, holds true.

True Labels			
Setosa	30	0	0
Versicolor	0	29	1
Virginica	0	1	29
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 3: Confusion matrix training on last 30 samples.

True Labels			
Setosa	20	0	0
Versicolor	0	20	0
Virginica	0	0	20
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 4: Confusion matrix testing on first 20 samples.

True Labels			
Setosa	30	0	0
Versicolor	0	24	6
Virginica	0	0	30
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 5: Confusion matrix training on first 30 samples, with sepal width removed.

True Labels			
Setosa	20	0	0
Versicolor	0	20	0
Virginica	0	1	19
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 6: Confusion matrix testing on last 20 samples, with sepal width removed.

True Labels			
Setosa	30	0	0
Versicolor	0	24	6
Virginica	0	0	30
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 7: Confusion matrix training on first 30 samples, with sepal length and width removed.

True Labels			
Setosa	20	0	0
Versicolor	0	20	0
Virginica	0	2	18
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 8: Confusion matrix testing on last 20 samples, with sepal length and width removed.

True Labels			
Setosa	29	1	0
Versicolor	0	19	11
Virginica	0	0	30
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 9: Confusion matrix training on first 30 samples, with only pedal width.

True Labels			
Setosa	20	0	0
Versicolor	0	20	0
Virginica	0	6	14
Predicted Labels	*Setosa*	*Versicolor*	*Virginica*

Table 10: Confusion matrix testing on last 20 samples, with only pedal width.

4 Nearest neighbor classifier of numbers

A database, called MNIST with pixtures of handwritten numbers 0-9 have dimension 28x28 pixels and are in 8-bit greyscale.

4.1 Implementation

Code written in Python, see appendix B. The library numpy is also used in this task. In addition, njit and the clustering algorithm `Kmeans` from sklearn library is used.

4.2 Results

After running NN-classifier and KNN-classifier with and without clustering on all 10000 test images, we got the results summarized in table 11. The clustering was performed into $M = 64$ clusters.

Classifier	Clustering	Time	Error rate	Confusion matrix
NN	No	1463 s (24 min)	3.09%	Table 12
KNN (K=7)	No	2722 s (45 min)	3.06%	Table 13
NN	Yes	13.8 s*	4.71%	Table 14
KNN (K=7)	Yes	17.3 s*	6.23%	Table 15

Table 11: Classification of test images.

* needs clustering, which can be done off-line, took 306.2 s

The KNN and the NN-classifier performed with the same error rate on the whole training set before clustering. From performance viewpoint, we look at the time running at Google Colab. The NN-classifier was about twice as fast, averaging about 0.15 s per image.

After clustering, the reference data points was reduced from 60000 to 640, or reduced to about 1% of the original set. The NN-classifier used about 1% of the original time, whereas the KNN-classifier used considerable less than 1% of the original time. This is a reasonable find.

Our NN-algorithm do not sort the data points and uses linearly time $\mathcal{O}(n)$. Our KNN-algorithm however, needs to sort the data points in addition, which runs in linearithmic time $\mathcal{O}(n \log n)$. This appoints to more time "saved" when reducing the reference data points.

Both KNN and NN-classifier performed worse after clustering. The NN-classifier had about 50% more errors, and to our surprise, KNN had about 100% more errors. This might be due to the classes being "strongly" overlapping in the 784-dimensional feature space or using a local minimum (non-optimal solution) in the clustering algorithm. Lower error rate might be tuned by choosing a different value of K.

Predicted\True	0	1	2	3	4	5	6	7	8	9
0	973	0	7	0	0	1	4	0	6	2
1	1	1129	6	1	7	1	2	14	1	5
2	1	3	992	2	0	0	0	6	3	1
3	0	0	5	970	0	12	0	2	14	6
4	0	1	1	1	944	2	3	4	5	10
5	1	1	0	19	0	860	5	0	13	5
6	3	1	2	0	3	5	944	0	3	1
7	1	0	16	7	5	1	0	992	4	11
8	0	0	3	7	1	6	0	0	920	1
9	0	0	0	3	22	4	0	10	5	967

Table 12: Confusion matrix NN-classifier using whole training set.

Predicted\True	0	1	2	3	4	5	6	7	8	9
0	974	0	11	0	1	5	6	0	6	5
1	1	1133	8	3	8	0	3	25	4	6
2	1	2	988	2	0	0	0	3	6	3
3	0	0	2	976	0	8	0	0	11	6
4	0	0	1	1	945	2	3	1	7	8
5	1	0	0	12	0	866	2	0	12	4
6	2	0	2	1	5	4	944	0	1	1
7	1	0	16	7	1	1	0	989	6	11
8	0	0	4	4	1	2	0	0	916	2
9	0	0	0	4	21	4	0	10	5	963

Table 13: Confusion matrix KNN-classifier using whole training set.

Predicted\True	0	1	2	3	4	5	6	7	8	9
0	966	0	9	0	0	5	6	1	5	5
1	1	1129	7	0	5	1	3	19	1	4
2	4	2	977	6	4	1	1	6	5	5
3	0	0	9	944	0	17	0	0	16	6
4	1	0	1	1	917	2	1	7	5	20
5	4	0	0	27	3	850	2	1	21	4
6	3	2	2	0	8	4	940	0	1	1
7	0	0	11	7	2	1	1	965	5	26
8	0	1	15	17	3	5	3	4	910	7
9	1	1	1	8	40	6	1	25	5	931

Table 14: Confusion matrix NN-classifier after clustering.

Predicted\True	0	1	2	3	4	5	6	7	8	9
0	954	0	16	0	1	5	10	0	5	7
1	1	1127	8	2	16	3	5	38	2	8
2	3	2	958	9	2	0	4	15	4	3
3	1	1	8	951	0	30	1	1	27	12
4	0	0	2	2	900	3	6	7	5	30
5	9	0	0	20	0	833	9	0	28	4
6	11	3	3	0	10	9	922	0	3	1
7	1	0	12	6	2	2	0	927	6	20
8	0	2	25	15	2	5	1	1	887	6
9	0	0	0	5	49	2	0	39	7	918

Table 15: Confusion matrix KNN-classifier after clustering.

After highlighting the most common errors in the confusion matrices, both classifiers before clustering made about the same mistakes, such as classifying 7 when 2 is correct or classifying 9 when 4 was correct. After clustering, the same errors were more profound, but the NN and KNN differed with some of the most common errors. They errors look more ”randomly” or evenly distributed.

10 random correctly classified numbers are given in figure 4 and 10 misclassified numbers are given in figure 5. As humans, we agree with the classified numbers. However, we think there is some ambiguity in some of the misclassified numbers. These include the first 3 as 5, 2 as 7 and 7 as 4 in figure 5. It seems that when the numbers are slightly skewed or the pencil markings is dragged at the end, both the classifier and we as humans struggle. However, most of the unclassified numbers are still easy to differentiate as humans.

Figure 4: 10 correctly classified numbers.

Figure 5: 10 misclassified numbers (guess in red).

5 Conclusion

From the theoretical standpoint, the results is in large part as expected. The linear classifier performs sub-optimally on a non-separable data set, and this performance worsen with less features. This is reasonable, as the a linear classifier on a non-separable data set find a border leading to the least errors, and with less data, it will set that border less accurately.

The difference in results coming from the two different splits of the iris data set, is also expected, as it is unlikely that two different subsets of an data set has the same qualities. This differences would however have a less effect if the data set was larger.

The linear classifier performed as good as one could expect it to do, however even at best it was not perfect. The error rate is even at best far higher than what would be acceptable for most real world applications. Using a more reliable and common classifier in the industry, like SVM or naive bayes, would be a useful exercise.

Classification with nearest neighbor classifiers on the MNIST dataset for handwritten numbers provided a reasonable low error rate, but had a slow running time when uitilizing the whole training data set. Clustering reduced the running time, but at the expense of higher error rate. A practical trade-off on accuracy and running time should be addressed before using such a classifier in "real world". The results also showed that KNN did not perform better than NN in our configuration.

When coding in python, we learned to use njit to decrease running time, which was crucial to run the NN-classifiers on the big datasets, but did not matter as much for the iris task. Without njit, we would have used about 10 times longer to compute. Here, we also learned that off-line optimizing such as euclidean distance squared and clustering provides faster execution.

In general the project was useful to get some knowledge of fundamentals in classification theory, like clustering, supervised and unsupervised learning, and linearity. It sparked an interest for further exploration in the field.

References

[1] Department of Electronic Systems NTNU. TTT4275 Project, 2021.

[2] Linear classifier with softmax, 2017.

Appendices

A Linear Classifier in Python

Listing 1 shows how a linear classifier is implemented. Listing 2 and 3 shows how a linear classifier is trained.

Listing 1: Python-code for a linear classifier

```
def classify(Weight, data):

    wrong = []
    correct = []
    for sample in data:

        sample_features = np.zeros((n_features, 1))

        for i in range(n_features - 1):
            sample_features[i][0] = sample[i]

        g = np.matmul(Weight, sample_features)

        label = np.argmax(g)
        if (label == 0):
            if sample[n_features] != 'Iris-setosa':
                sample.append(0)
                wrong.append(sample)
            else:
                correct.append(sample)

        elif (label == 1):
            if sample[n_features] != 'Iris-versicolor':
                sample.append(1)
                wrong.append(sample)
            else:
                correct.append(sample)

        elif (label == 2):
            if sample[n_features] != 'Iris-virginica':
                sample.append(2)
                wrong.append(sample)
            else:
                correct.append(sample)

    return wrong, correct;
```

Listing 2: Python-code to train a linear classifier

```
def train():

    W = np.zeros((n_classes, n_features))
    alpha = 0.01
    iterations = 0
    while True:
```

```
        wrong, correct = classify(W, training_data)
        W = update(W, alpha, wrong)
        iterations = iterations + 1
        if not wrong or iterations > 10000 or np.linalg.norm(W) > 10:
            break

    return W, wrong, correct
```

Listing 3: Python-code for updating the weight-matrix

```
def update(Weight, alpha, wrong):

    for sample in wrong:

        sample_features = np.zeros((n_features, 1))
        for i in range(n_features - 1):
            sample_features[i][0] = sample[i]

        z = np.matmul(Weight, sample_features)
        g = sigmoid(z)
        t = getTrueLabel(sample[n_features])

        grad_W_MSE_k(g, t, sample_features)

        Weight = Weight -  alpha * grad_W_MSE_k(g, t, sample_features)
    return Weight
```

B KNN Classifier in Python

Algorithm for clustering:

Algorithm 1 k-means algorithm

1: Specify the number of k of clusters to assign.
2: Randomly initialize k centroids.
3: **repeat**
4: Assign each point to its closest centroid.
5: Compute the new mean of each cluster.
6: **until** The centroid positions do not change

Libraries included in listing 4 and imported data from the file `data_all.mat`[2] in listing 5. The functions for the nearest neighbor are given in listing 6, functions for K-nearest given in listing 7, clustering in listing 8 and finding confusion matrix in listing 9.

[2] resource from the course material

Listing 4: Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import time
from numba import njit
from sklearn.cluster import KMeans
```

Listing 5: Import data

```
data = scipy.io.loadmat('data_all.mat')

size_img =  int(data['vec_size'][0][0])      # Vector (img) size, 784
num_ref  =  int(data['num_train'][0][0])     # Size ref. data, 60000
num_test =  int(data['num_test'][0][0])      # Size test data, 10000

ref_imgs =  data['trainv'].astype(np.float) # Ref. data, 60000x784
ref_lbls =  data['trainlab'].astype(np.int) # Lables ref. data, [0..9]

test_imgs = data['testv'].astype(np.float)  # Input data, 10000x784
test_lbls = data['testlab'].astype(np.int)  # Lables input data, [0..9]

num_lbl =   10                              # Number of labels, 10
```

Listing 6: Definitions of nearest neighbor

```
@njit
def euclidean_dist_square(a, b):
    diff = np.subtract(a, b)
    return np.dot(diff, diff)

@njit
def get_nearest(img):
    img_idx, img_dist = 0, euclidean_dist_square(img, ref_imgs[0])
    for idx in range(len(ref_imgs)):
        this_dist = euclidean_dist_square(img, ref_imgs[idx])
        if this_dist < img_dist:
            img_idx, img_dist = idx, this_dist
    return ref_lbls[img_idx]
```

Listing 7: Definitions of K-nearest neighbors and mode of class (vote)

```
def get_nearest(img, K):
    ''' List of K nearest labels to img'''
    list_nearest = []
    for idx in range(num_ref):
        dist = euclidean_dist_square(img, ref_imgs[idx])
        list_nearest.append((ref_lbls[idx], dist))
    list_nearest.sort(key=lambda x: x[1])
    return [int(i[0]) for i in list_nearest[:K]]

def vote(lbls):
    ''' Label with the most votes in neighbors '''
```

```
    nearest_counter = [0]*num_lbl
    for lbl in lbls:
        nearest_counter[lbl] += 1
    return max(range(len(nearest_counter)), key=nearest_counter.__getitem__)
```

Listing 8: Clustering of data points

```
M = 64      # number of cluster points

def cluster(data, lbls, classes, M):
  ''' Cluster of ref_data and ref_lbl in M clusters for each class '''

  ref_data_cluster = []
  ref_lbl_cluster = []

  for cl in classes:
      cl_indices = np.where(lbls == cl)[0]
      cl_data = data[cl_indices]

      data_cluster = KMeans(n_clusters = M). fit(cl_data). cluster_centers_
      lbl_cluster = [cl]*M

      ref_data_cluster.extend(data_cluster)
      ref_lbl_cluster.extend(lbl_cluster)

  return ref_data_cluster, ref_lbl_cluster

started_at = time.time()
ref_imgs, ref_lbls = cluster(ref_imgs, ref_lbls, range(num_lbl), M)
num_ref = M*num_lbl
ended_at = time.time()
```

Listing 9: Confusion matrix

```
def get_confusion_matrix(lbls_predicted, lbls_true, lbls):
    ''' Confusion matrix, true labels along first dimension,
        predicted labels along second dimension '''
    tensor = []
    for predicted_lbl in lbls:
        lbl_row = []
        for true_lbl in lbls:
            predicted_indices = np.where(
                lbls_predicted == np.uint8(predicted_lbl))[0]

            true_indices = np.where(
                lbls_true == np.uint8(true_lbl))[0]

            num_occurences = len(
                np.intersect1d(true_indices, predicted_indices))

            lbl_row.append(num_occurences)

        tensor.append(lbl_row)
```

```
    return np.array(tensor)
```