Assignment 1

14th April 2018

1 Introduction

In this assignment we evaluate 4 different classification algorithms using the MNIST hand written digits as our data. The algorithms are:

1. Distance analysis between the images
2. Naive Bayes
3. Multi-class perceptron
4. Gradient descent

2 MNIST data set

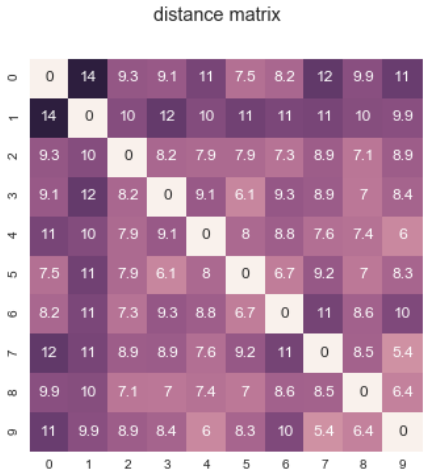
MNIST digit database is a classic in machine learning. The aim is to classify a hand written digit based on a black and white image. It's widely used to study classification algorithms.

3 Distance classifier

One of the simplest ways to classify data is based on distance calculation in some relevant parameter space. We start by creating a cloud for each digit by grouping all training images for each digit C_d. After that we calculate the center for each cloud by calculating the mean value for each position for every cloud C_d. Basically we're looking at every image as 256 dimensional point (16 * 16) then taking the mean of all points in each dimension. Since the images are of hand written digits it's natural that there will be some images outside the norm, and to take this in account we compute the radius r_d which is largest distance (Euclidean distance) within a cloud C_d and it's center see Table 1, that is between the center of C_d and the points of C_d.

0	1	2	3	4	5	6	7	8	9
15.9	9.5	14.2	14.7	14.5	14.5	14.0	14.9	13.7	16.1

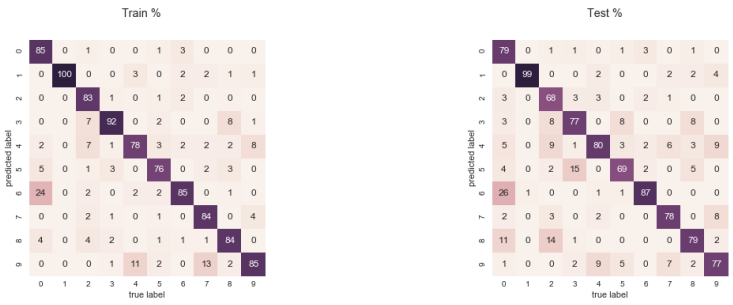
Then we calculate the distances between the clouds centers creating the distance matrix. Looking at the distance matrix in the figure below can give us an idea how easily separable different digits will be. For instance, we can see



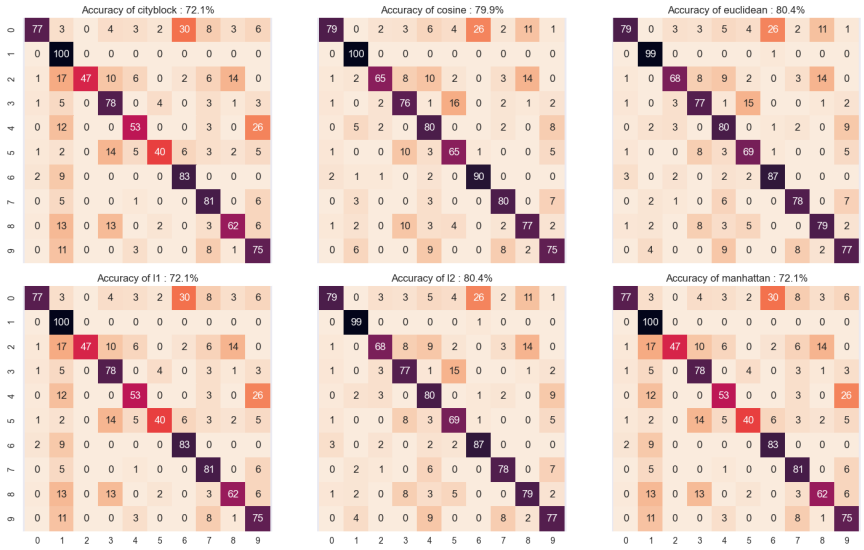
that 7 & 9 have the smallest distance (5.4) rendering them to be very hard to distinguish.

Now that every thing is in place, the way to classify a digit image is by calculating the distance between this image and the centers of the clouds, and the minimum distance will be the class(digit) assigned to the image.

We apply the classifier on both the training and the test data. To measure the effectiveness of the classifier we calculate the percentage of the correctly classified images and create a confusion matrix for both data sets.



So far we have used the euclidean distance, we then used the different distance metrics to evaluate which is the best suited for this classifier and the best results is obtained using the euclidean and l2 distance. The results are shown in the confusion matrices below.



4 Naive Bayes

Instead of looking at 256 dimensional space and trying to classify based on all the values of the individual pixel why not look at specific feature(s) that separates the digits, using the Naive Bayes classifier.

Naive Bayes classification is a probabilistic classification scheme based on Bayes theorem, which is based on conditional probabilities. Bayes theorem is given by the equation:

$$P(C_k|x) = \frac{P(x|C_k)(P(C_k)}{P(x)}$$

We want to find posterior probability $P(C_k|x)$: the probability of a class C_k given a feature(s) x. From the equation above this can be easily calculated by computing the following:

- $P(C_k)$ prior probability of class
- $P(x)$ prior probability of a feature
- $P(x|C_k)$ probability of feature given class (likelihood)

More specifically, we will say that a data point will belong to a class j and not to class i if:

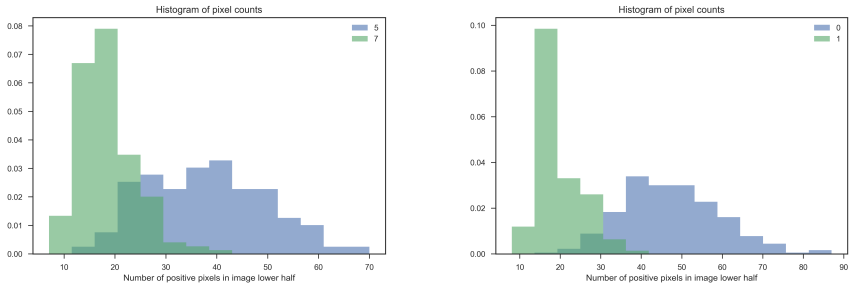
$$P(C_j|x) > P(C_i|x)$$

This implies results will depend on the selected features(s), as will be discussed later after describing the implementation of Bayes classifier on 2 dig-

its(classes) of the MNIST data set. We choose to differentiate between the digits 5 and 7, calculating the $P(C_k)$ prior probabilities for the classes is straight froward. The feature we selected is the amount pixels > 0 in the bottom half of the image, we make use of the histogram of the features of the 2 classes, returning the feature frequency, from there we are able to calculate the $P(x|C_k)$.

We now have what we need to calculate the posterior probability $P(C_k|x)$ using Bayes theorem. We use the classifier on the the test data of the 2 classes. Test cases of the digits are placed in the bins predetermined by the histogram, and consequently are assigned a class using the posterior probabilities. In the case of 5 and 7, the accuracy on the test set was 81%. The conclusion here is that with the proper feature(s) Bayes classifier performs with high accuracy, and it's fast.

We tested the classifier on 2 sets of numbers (5 & 7) giving 80.7% accuracy and to see how well it generalized (0 & 1) giving 93.3% accuracy the histograms of the feature are shown below



While implementing Bayes algorithm we first selected to look at the pixels on the left half of the image bit that was a bad choice of a feature and gave us 63% accuracy

5 Multi-class perceptron - single layer Neural Network

The idea of neural networks is to mimic how the human brain learns to solve a problem, that is the algorithm learns by correcting it self. Here we will work with a single layer neural network that is an input layer (256 pixels + 1 bias node) and an output layer (10 digits), and there is a link from each input node to each output node, these links have weights w_i (257 * 10), and according to the weights the output nodes are activated by the sum of the input times the weight, the input is then classified to be the output node with the maximum activation value.

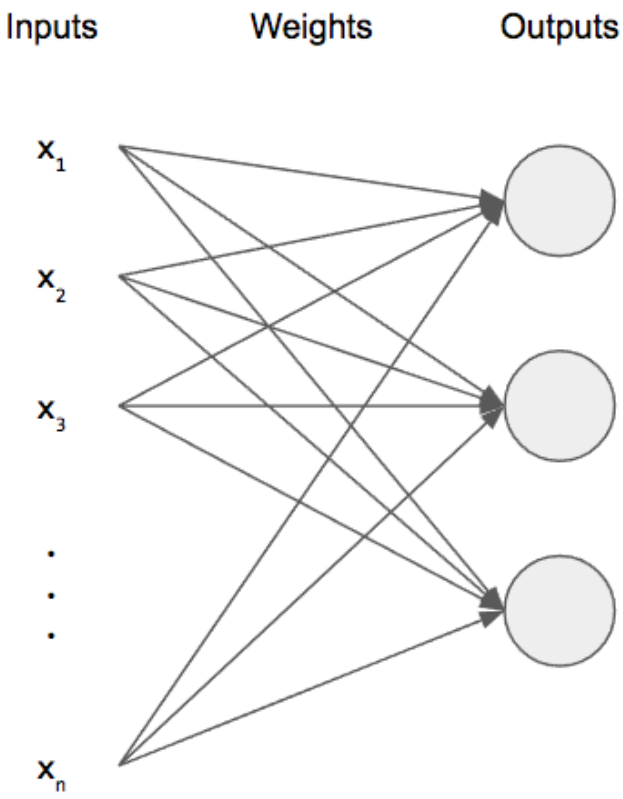


Figure 1: single layer neural network

Looking at the figure above we can represent the input as vector X with size 256, the weights as a matrix W of size 257 * 10, the output as vector Y with size 10 so to calculate the value of the output nodes we use matrix multiplication as follows: $W * X^T = Y$. We also use a learning rate η that is used to control

by how much weights are updated every iteration. Training the perceptron the proceeds as follows:

1. Initialize W randomly
2. The output nodes values is calculated as the sum of the input * weights $\sum_{i=1}^{257} w_{ij} * x_i$ using matrix multiplication
3. Class of input = the max of the output nodes
4. Iterate over miss-classified training samples and update the weights corresponding to the miss-classified digit with $\eta * d$, d being the desired output
5. Repeat n times or until there is no miss-classified input

After running the algorithm and testing the data, as expected the accuracy on the test set and number of epochs(iterations) needed to learn, depend on two factors: the learning rate η and the random numbers we start with as seen in the table below:

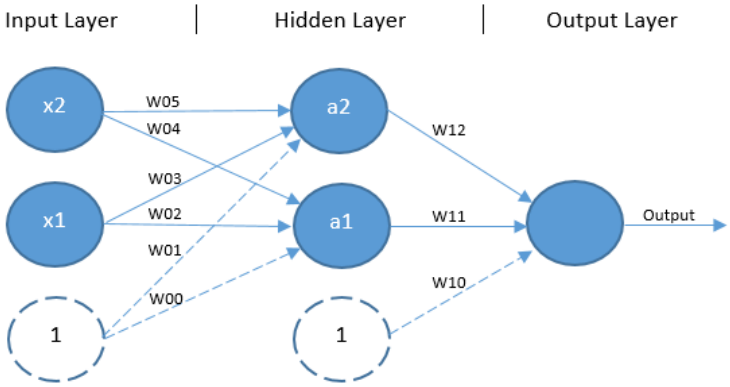
Learning rate	Random seed	Iteration to convergence	Test accuracy
.01	100	2123	82.1
	500	241	86.6
	11024	218	87.2
.5	100	280	86.6
	500	235	86.5
	11024	223	87.2
.1	100	278	86.8
	500	243	86.4
	11024	277	87

6 Gradient descent - Neural Network 2 layers

For this classifier we start with the classical XOR (exclusive OR problem, it's a simple logical function that returns true only if the inputs are not equal. The XOR is a classification problem that single layer network fails to learn, as was first shown by Minksy and Papert (1969)).

Here we solve the XOR problem with a 2 layer neural network, that is we have a hidden layer that is required to solve the XOR, again we have an input layer with 2 nodes and the bias, the hidden layer with 2 nodes and a bias, then an output layer with one node.

We train the network in similar way that was done in the previous section but this time it's done on two layers and we use the gradient descent method to update the weights. How fast the classifier converges depend on the initial random weights, the learning rate η and the activation function, the most used



activation functions being sigmoid, tanh or relu. This network is then trained as follows:

1. Initialize the weights with random values and calculate mean squared error (MSE) of the output : MSE $= (\hat{y} - y)^2$.

2. Calculate the gradient i.e. change in MSE when the weights are changed by a very small value $\varepsilon \sim 10^{-3}$ from their current value. For every weight, the gradient is calculated with

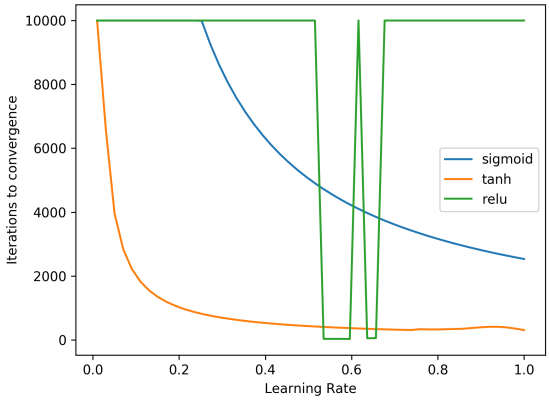
$$\frac{\partial MSE}{\partial w_i} = \frac{MSE(w_1, .., w_i + \varepsilon, .., w_n) - MSE(w)}{\varepsilon}$$

3. Update the weights using the gradient and learning rate η to reach the optimal values where the MSE is minimized.

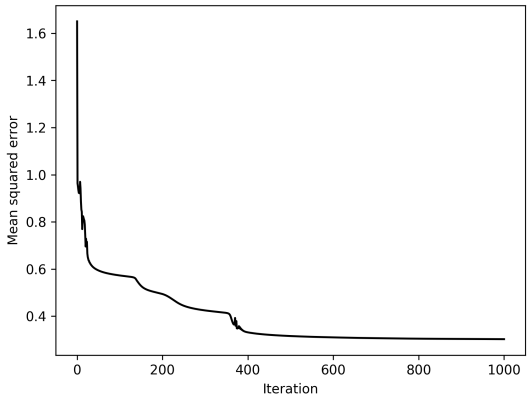
4. Repeat steps 2 and 3 until all inputs are correctly classified.

Because this network is fast to train, having only 9 weights in total, this provides an opportunity to test for different learning rates and activation function and see how the network performs. We test how many iterations it takes for the network to converge, that is until all inputs are correctly classified. To contain learning strategies that take too long to converge, we limit the number of iterations to 10,000. Of course, the weights are initialized using the same random seed for every configuration, to keep the results consistent. The results of this are shown in the figure below.

We see than the tanh activation function performs better than the sigmoid which was expected since tanh has values in the range (-1,1) while sigmoid has a range of (0,1), which means that the derivatives of the tanh will be bigger than that of sigmoid. It also seems that higher learning rate will cause the network to converge faster. Looking at the relu the results are a bit puzzling, as it fails to converge within 10,000 iterations pretty consistently. For some specific learning rates however, it suddenly performs very well and converges within 50 iterations.



Now that we have the basic working example of a neural network, we can use this scheme to test how well such a neural network would do on the MNIST data set. We simply increase the scope of the network, still being 2 layers, but with 257 nodes in the input layer, 30 nodes in the hidden layer and 10 nodes in the output layer. Because of the way the gradient is calculated, we can already expect this network to be very slow. The result of training this network for 1,000 iterations with a learning rate of $\eta = 5$ and a sigmoid activation function is shown below.



After 1,000 iterations, the mean squared error sits just below 0.4, and it looks like the network might be in a local minimum. The choice of activation function and learning rate has a lot of influence on the rate of convergence as we have seen before, but to properly test this we'd need either a lot of time or a more powerful computer, as training the network for 1,000 iterations currently takes more than 6 hours. When applied to the test set, the accuracy of the

network clocks in at 63.3%, which is of course not at all satisfactory given the time spent training this network.

The performance of this network gives us insight into why it has taken so long for neural networks to take off, as with insufficient computing power this implementation is very inefficient and doesn't yield the results to justify the computing power.