DIGIT CLASSIFICATION WITH THE KERNEL PERCEPTRON

Ozlu Dolma - 942512

March 20, 2022

1 Introduction

The purpose of this study to implement the polynomial kernel perceptron algorithm from scratch for multi-class classification of handwritten digits in the MNIST dataset. For different number of epochs (from 2 to 10) and the degree of the polynomial (from 1 to 6), the multi-class classification performance was examined for two different methods used in predictor selection: (1) the average of the predictors in the ensemble and (2) the predictor achieving the smallest training error among those in the ensemble. The structure of the study is as follows: First, the theoretical framework is presented briefly in order to describe the main mechanisms underlying the polynomial kernel perceptron. Next, an overview of the dataset is provided in order to have a further visual understanding of the multi-class classification problem to be solved in this study. Then, the methodology and the experimental results are presented and discussed in detail.

2 Theoretical Framework in a Nutshell[1]

2.1 Linear Prediction

Linear prediction is a parametric algorithm which requires that the predictor is specified using “d” parameters irrespective to the size of the training set, where d is the dimensionality or the number of attributes in the learning problem. Here, the predictors or the linear classifiers are functions from R^d to the label space Y where the elements of the label space can be real numbers in the case of regression or binary labels such as $\{1, -1\}$ in the case of binary classification. The predictors are of the following form:

$h(x) = f(w^T x)$ where w $\in R^d$ and $w^T x$ is the inner product of the two vectors w and x

Here, w is the vector of d coefficients representing the linear predictor. They are trainable parameters that are learnt when training the predictor on a training set.

[1]The theoretical information presented in this section is a brief summary of the information obtained during the lectures dedicated to these topics in the course of ”Machine Learning, Statistical Learning, Deep Learning and Artificial Intelligence”, taught by Professor Nicolò Cesa-Bianchi, at the University of Milan, in the Academic Year 2020-2021.

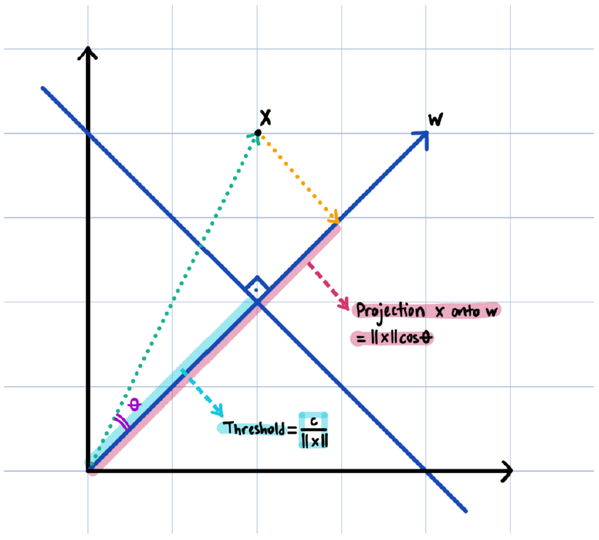


Figure 1: Projection of data point x onto w and the decision rule

In general, the function "f" is called the activation function. In the case of binary classification, the activation function is the "sgn" function and it defines the rule or the decision boundary for the classification of the data points in the following way:

$$sgn(z) = \begin{cases} 1 & if \quad z > 0 \\ -1 & if \quad z \leq 0 \end{cases}$$

and the predictors are then written as:

$$\text{h(x)} = \text{sgn}(\text{w}^T x - c)$$

That is, the sgn function takes $(w^T x - c)$ and returns a label (1 or -1) based on the rule described above.

The hyperplane separating the data points is defined by the linear classifier:

$$\text{H}=\{x \in R^d : w^T x = c\}$$

They are the points x that lie on this hyperplane.

Accordingly, two half-spaces can be defined:

$$H^{+}=\{x \in R^d : w^T x > c\} \quad \text{x}\in\text{H}^{+} \leftrightarrow h(x) = 1$$
$$H^{-}=\{x \in R^d : w^T x \leq c\} \quad \text{x}\in\text{H}^{-} \leftrightarrow h(x) = -1$$

The length of the threshold can be computed as follows:

$$w^T x = c$$
$$\|w\| \, \|x\| \, \cos\theta = c$$
$$\|x\| \, \cos\theta = \frac{c}{\|w\|}$$

This is depicted in Figure 1. The vector w and c define a hyperplane where the hyperplane is orthogonal to w and crosses w at a distance from the origin which is equal to c/‖w‖. When x is such that its projection onto w is smaller than the threshold, it will be classified as negative which means that it is on the negative side of the hyperplane defined by w and vice versa. In order to remove the additional term "c" and be able to work with homogeneous hyperplanes which go through the origin, an extra dimension of -c can be added to the set of dimensions w. Thus, it is possible to have homogeneous hyperplanes using a vector v=(w, -c) where $v \in R^{d+1}$ and adding an extra feature with value 1 to x where it becomes x'=(x, 1).

When learning binary linear classifiers, the length of the w is irrelevant to determine the $sgn(w^T x)$. The only thing that is important here is the angle between the two vectors w and x. If the angle θ between them is smaller than 90 degrees, the sign of the $\cos\theta$ will be positive which would mean that x will be on the positive half space. If the angle is greater than 90 degrees, it will be classified as negative. Given that the loss function is the zero-one loss, the predictor makes a mistake whenever signs of $(w^T x)$ and y disagree and accordingly zero-one loss can be described using the indicator function in the following way:

$$I\ \{yw^T x \leq 0\}$$

Given a training set of size m where $(x^t, y^t) \in R^d$ and the labels are $\{1, -1\}$, one may try to find then the Empirical Risk Minimizer (ERM) over the class of predictors H^d:

$$\mathrm{h}_s = \underset{h \in H_d}{\arg\min}\ \tfrac{1}{m} \textstyle\sum_{t=1}^{m} I\{y_t w^t x_t \leq 0\}$$

However, minimizing the zero-one loss is computationally hard. In other words, it is unlikely that there exists an algorithm that can answer the question of whether there exists or not a homogeneous hyperplane w such that it makes at most K mistakes on the training set, let alone it precisely finds a hyperplane whose training error is within a constant factor of the best possible training error.

An exception to this is when the training set is linearly separable. In this case, it is possible to find a separating hyperplane in polynomial time. There always exists a linear classifier whose training error is zero and the problem becomes a feasibility problem where one has a system of linear inequalities composed of m elements of the form $y_t w^T x_t > 0$ and one needs to find w which satisfies all of these inequalities in order to find a separating hyperplane for the dataset. However, since the feasibility problems are typically complicated, are not very useful for learning and it is difficult to extend them to solve other problems, one needs to use another algorithm called "Perceptron".

2.2 Perceptron Algorithm and Online Learning

Perceptron algorithm runs in cycles over the dataset which is called epochs. It starts with an empty vector w. In each epoch, the current vector w determining the hyperplane is tested on each example of the training set. If w is found to be incorrect on some data points, an update is performed. If an entire epoch is done without any updates, the current vector w defining the final hyperplane is output by the algorithm. How an update is performed is depicted in the Figure 2 below.

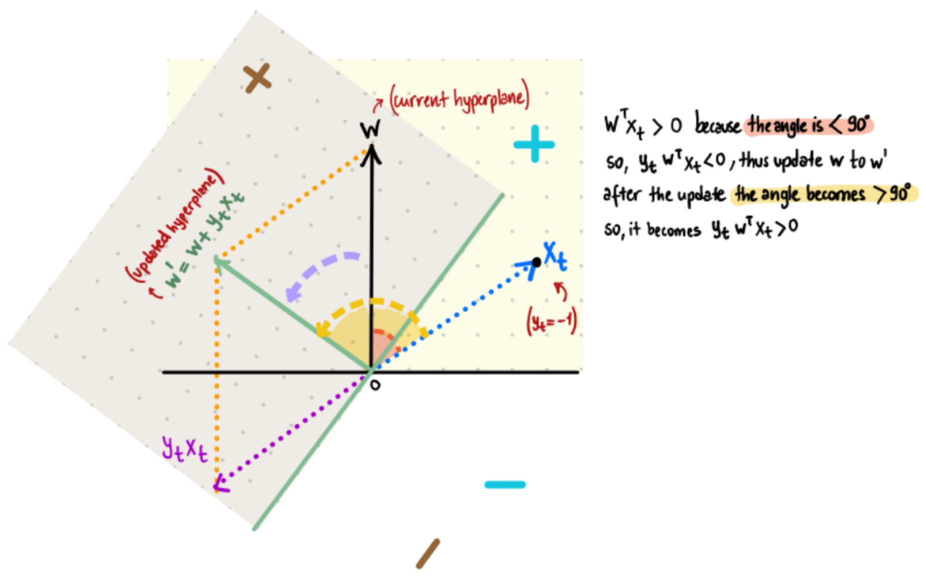

(current hyperplane)
w
(updated hyperplane)
w' = w + y_t x_t
x_t
(y_t=-1)
y_t x_t
o
+
−
w^T x_t > 0 because the angle is < 90°
so, y_t w^T x_t < 0, thus update w to w'
after the update the angle becomes > 90°
so, it becomes y_t w^T x_t > 0

Figure 2: Update w to w' when the data point x_t with label -1 was wrongly classified as +1 using w

Since every time the hyperplane is adjusted it is possible that the update made recently may destroy what was gained in the previous updates, one may be concerned about whether this process will converge to a separating hyperplane. In fact, on the linearly separable datasets, the algorithm converges and the number of updates it makes on any linearly separable dataset is upper-bounded by the following quantity:

$$min_{u:\gamma(u)\geq 1} \left\|u\right\|^2 max_{t=1...m} \left\|x_t\right\|^2$$

where the first term determines the separating hyperplane u with the smallest length that has a margin of at least 1 and computes the square of it and the second term determines the square of the radius of the smallest ball centered in the origin that includes all the training examples.

Basically, the first term indicates how separable the data points are and the second term indicates how spread out the points are in the space with respect to the origin. u defines a separating hyperplane because the margin of u is at least 1 and this can only be true for separating hyperplanes. This theorem is called the "Perceptron Convergence Theorem". In general, perceptron algorithm goes through the examples one after the other and thus it depends on the ordering of them in the training set. However, this theorem states that irrespective of the ordering of the examples in the dataset, the algorithm will always converge with this upper-bound and the perceptron will not make a number of updates more than this quantity.

Perceptron is an online algorithm where the points are accessed sequentially and the algorithm outputs a sequence of predictors. That is, whenever the perceptron receives a new data point, it updates the current predictor. This type of learning protocol is useful when the dataset is so large that processing the whole dataset all at once may become computationally cumbersome, as it is the case in the current study, and also when the examples are generated continuously (e.g. sensor data, financial data). Since each data point is processed in a constant time of the order of d, the adjustment of the predictor is done in an efficient way and each epoch takes a time that is linear in the number of data points.

When the online learning protocol is adopted, however, the performance of the algorithm is evaluated in a different way compared to when it is the case for the batch learning since there is no single predictor generated by the algorithm the risk of which can be compared against the risk of the best predictor in the class of predictors that the algorithm uses. However, a similar approach can still be applied. It is possible to compare the sequential risk of the algorithm with respect to the sequential risk of the best predictor in the class which could have been used if the entire portion of the dataset would have been provided to the algorithm at once. This difference is called "regret".

The advantage of online learning, on the other hand, is the fact that, each time a data point is processed by the algorithm the current predictor h_t condenses all the information regarding the first t-1 data points and the corresponding losses that the algorithm has seen so far. As further elements from the data stream are processed by the algorithm, the new information is incorporated further by making an update. This procedure is much more straightforward when the loss function is a convex and differentiable function (e.g. square loss), since it is possible to do convex optimization using gradient descent and the expectation is that the regret goes down as the algorithm processes more and more examples from the data stream and it predicts better and better. In other words, for any possible portion of the data stream up to some number of data points, it is desired that the sequential risk of the algorithm is always close to the sequential risk of the best predictor in the class of predictors.

Unfortunately, this theorem of convergence cannot be directly applied for the perceptron algorithm because perceptron algorithm is used for binary classification and zero-one loss at which the algorithm's performance is evaluated is not a convex function. As a solution to this problem, one can use the "hinge loss" function which looks like a convex function incorporating the fact that here the focus is on the number of mistakes made by the perceptron and puts a convex upper-bound on top of the indicator function which makes it possible to run an online gradient descent (OGD) analysis. When conducting OGD analysis on the perceptron, the focus is then only on the steps where the perceptron makes a mistake, because the algorithm does nothing when the signs of $(w^T x)$ and y agree with each other. On any data sequence for binary classification, then, the number of mistakes of the perceptron is compared to the hinge loss of the best predictor. Accordingly, the total number of mistakes (m_T) that the perceptron makes on any sequence of data points which is not necessarily linearly separable is upper-bounded as follows:

$$m_T \leq \sum_{t=1}^{T} h_t(u) + (\|u\| X)^2 + \|u\| X \sqrt{\sum_t h_t(u)}$$

When the data points are linearly separable, the hinge loss $h_t(u)$ is always zero because there always exists some u in R^d such that the margin of u is at least 1 for each example in the dataset. Thus, the terms including $h_t(u)$ cancel out and the number of mistakes is upper-bounded by the constant quantity which corresponds to the upper-bound defined before in the Perceptron Convergence Theorem. As the data points become linearly non-separable, however, the hinge loss terms become positive and they will contribute to the number of mistakes.

When conducting OGD analysis on the perceptron, the effect of the learning rate which is generally used in the analysis for the convex losses, becomes neutral in the sense that the algorithm will have the same prediction performance regardless of the value of the learning rate. This is so, because the learning rate only affects the norm of w as a scaling factor and in perceptron predictions depend only on the $\text{sgn}(w_t^T x_t)$ and this sign does not depend on the norm of w.

2.3 Polynomial Kernel Perceptron

Although linear predictors can easily be used for online learning, they usually have a high bias. That is, the approximation error incurred is large, because in general the Bayes Optimal Predictor for a given learning problem is not a linear function in the given feature space. On the other hand, since the estimation error will be small, in order to find a balance between bias and variance errors and to have a small risk, one can use "feature expansion". In feature expansion, new features are created using the features that already exist. That is, for any point x in the original feature space R^d, a function takes x and maps it to a high dimensional space called V. An example for such a feature expansion from a two dimensional space to a six dimensional space can be described as the follows:

$$\Phi : R^d \to V \quad x \epsilon R^d, \quad \Phi(x) \epsilon V$$
$$d = 2 \quad \Phi : R^2 \to \Phi : R^6 \quad \Phi(x_1, x_2) = (1, x_1, x_2, x_1^2, x_2^2, x_1 x_2)$$

where the features in the expanded feature space are called monomials (here, monomials of degree ≤ 2). Then, for this example, the linear prediction on a feature expanded point will have the following form:

$$w_T \Phi(x_1, x_2) = (w_1, w_2 x_1^2, w_3 x_2^2, w_4 x_1, w_5 x_2, w_6 x_1 x_2)$$

Doing feature expansion has the advantage of being able to learn a complex polynomial classifier in the original feature space. Learning a polynomial classifier, then, makes it possible to separate data points that are not linearly separable, because all higher degree curves have a parametric form and they include the set of all homogeneous parabola, hyperbola, and ellipses.

A generic polynomial expansion is then composed of features of the form $\prod_{s=1}^{K} x_{v_s}$ where v_s are the indices in the original feature space and monomials are constructed up to K degrees. The cardinality of the feature map N is the following geometric sum:$\sum_{K=0}^{n} d_K$.This is the total number of monomial features x_{v_s} of degree up to n on R^d that can be constructed in order to have a feature expansion. Although the feature expansion can be done in this way, the number of components that are constructed are exponential in the degree of the polynomial used and if d is already large, it may become a computational barrier of $\theta(d^n)$.

Going back to perceptron, it starts with a vector of zero coefficients $w_1 = (0, \dots, 0)$ and it updates w only when it makes a mistake and the update rule can be written as $w_{t+1} = w_t + y_t x_t I\{y_t w^t x_t \leq 0\}$. If a set S comprises all the indices s of the data points where perceptron made a mistake while running over the data set such that $y_s w_s^T x_s \leq 0$, then w can be expressed as a sum of over $y_s x_s$ and thus the linear classifier that the perceptron uses can simply be described in the following form:

$$h(x) = sgn(\textstyle\sum_{s \epsilon S} y_s x_s^T x_t)$$

Accordingly, when the perceptron is run in the expanded feature space, the classifier has the following form $h_\phi(x) = h(\phi) = sgn(\sum_{s \epsilon S} y_s \phi(x_s)^T \phi(x))$. Computing $\phi(x)^T \phi(x')$ however can be problematic in the sense that there are N components when computing the inner product between the vectors and N can increase exponentially with the degree of the polynomial. In particular, $\phi(x)^T \phi(x')$ is called the Kernel Function $K(x, x')$ and and it can be computed in a more efficient way using the "kernel trick" which addresses the computational barrier problem described above. Using the kernel trick, the inner product between the vectors $\phi(x)^T \phi(x')$ can be computed faster in time $\Theta(d)$ by performing $(1 + x^T x')^n$.That is, since the inner product computed is $x^T x'$ over the space of features, it is order of d rather than order of N.

The generic polynomial kernel can be expressed as follows:

$$K(x, x') = (1 + x^T x')^n \text{ where } n \in N$$

This polynomial kernel corresponds to feature expansions of:

$$\phi : R^d \to R^N \quad N = \Theta(d^n) \quad K(x, x') = \phi(x)^T \phi(x') \quad \forall x, x' \in R^d$$

In sum;

$$\phi(x)^T \phi(x') = (1 + x^T x')^n$$

Consequently, perceptron can be performed in a polynomial expanded feature space efficiently by running the following pseudo-code of the Kernel Perceptron Algorithm:

```
$S = \emptyset$                                         : initially, S will be the empty set
For $t = 1, 2, \dots$                                   : over the stream of the data set
  1. Get $(x_t y_t)$                                    : get the example
  2. Compute $\hat{y} = sgn(\sum_{s \epsilon S} y_s K(x_s, x_t))$   : do the prediction
  3. If $\hat{y}_t \neq y_t$ then $S \leftarrow S \cup t$          : if the prediction is different from the
                                                          true label, add the current index
                                                          of the training set to S
```

So, basically, the algorithm memorizes the subset of points where it made a mistake, because they are the points which define the linear predictor output by the algorithm.

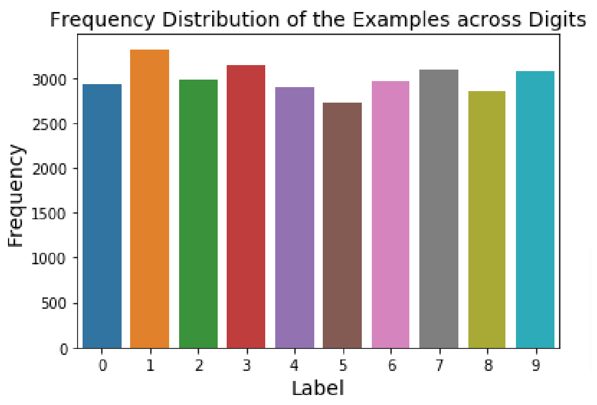


Figure 3: Training Dataset (30,000 Examples)

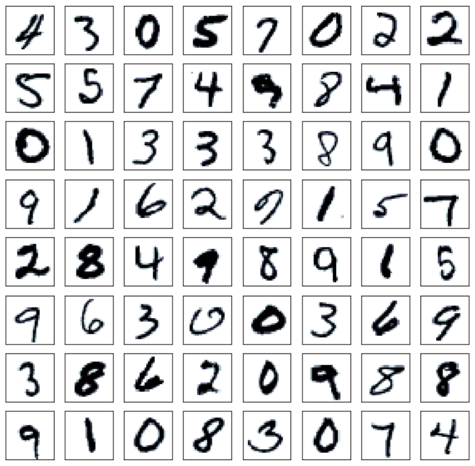


Figure 4: A Sample of Handwritten Digits

3 Methodology and Experimental Results

3.1 An Overview of the Dataset

Originally, the MNIST dataset includes 28 x 28 pixel images of handwritten digits and is composed of a training set and a test set which include 60,000 and 10,000 examples, respectively. Each digit has 784 features that are integer pixel values ranging from 0 to 255. The corresponding labels, on the other hand, are integer digit values from 0 to 9. Because of the technical problems encountered in terms of the memory space and in order to have a reasonable running time, a random subset of the training set consisting of 30,000 examples was used as the training set in this study. The test set was used as it is. The frequency distribution of the examples in the training set across the 10 digits are presented in Figure 3. As seen, the distribution is quite balanced across different digits.

In order to see how the images in MNIST look like, in Figure 4, a visualization of a sample of handwritten digit images is provided.[2] Moreover, in order to have a further understanding of the classification problem to be solved, via principal component

[2]Python code for this figure was adopted from the code presented in the following website: https://www.kaggle.com/schateau/hands-on-kaggle-s-mnist-dataset-99-07.

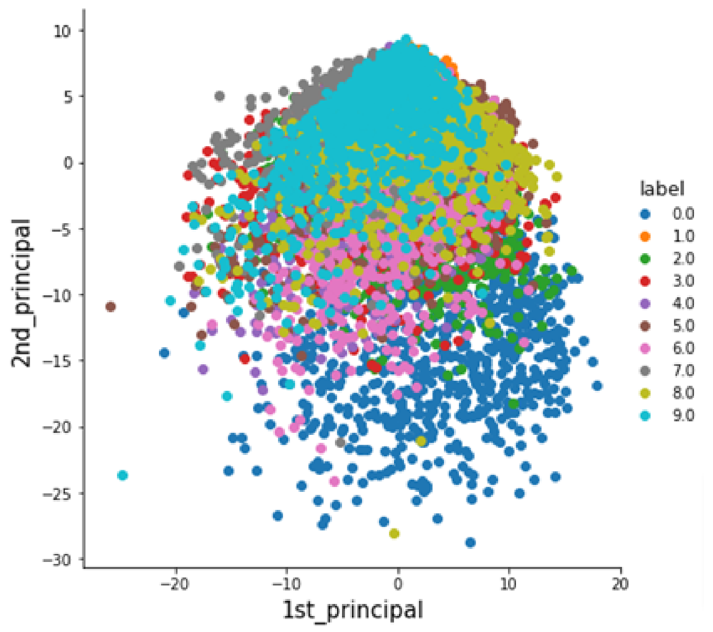


Figure 5: MNIST Dataset in 2 Dimensions

analysis, the 784 dimensional MNIST data was reduced to 2 dimensional data and the dispersion of the data points were displayed as in Figure 5.[3]

3.2 Methods

In this study, kernel perceptron was implemented by performing practically the following pseudo code:

```
$w \leftarrow 0$, $e \leftarrow 0$                          :initialize coefficients and the number of errors
for epochs = 1...n do                                      :for the specified number of epochs compute
    for all $(x_t y_t) \in D$ do                           :over the stream of the dataset
        $a \leftarrow \sum_m w_m \phi(x_m)\phi(x_n) + b$   :compute the activation for the example
        if $y_n * a \leq 0$ then                           :if the signs do not agree then
            $w_n \leftarrow w_n + y_n$                     :update w
            $e \leftarrow e + 1$                           :update the number of mistakes made
        end if                                             :move to the next example
    end for                                                :after all examples have been examined, end
end for                                                    :after it is repeated for the specified
                                                            number of epochs
return w                                                   :return the final w vector
```

In order to reduce the computation time, the product between the vectors $\phi(x_m)\phi(x_n)$ were computed before the algorithm was run. That is, for each degree of the polynomial (from 1 to 6) a kernel matrix was computed. Then, depending on the degree of the polynomial employed, that appropriate kernel matrix was used for training and testing. Thus, instead of the feature values of the data points, both the

[3]Python code for this figure was adopted from the code presented in the following website: https://www.kaggle.com/mkashif/dimensionality-reduction-and-viz-of-mnist-dataset/notebook.

"Perceptron" function used to obtain a classifier and the "Predict" function used to classify the data points received the appropriate kernel matrix as one of its arguments. In order to compute these kernel matrices, from the sklearn.metrics.pairwise sub-module, polynomial function was used. Using this function, the polynomial kernel is computed as follows:

$$k(x, y) = (\gamma x^T y + c)^d$$

where x and y are the two Numpy arrays with the shapes: number of examples (in X or Y), number of features. The term c is used to specify if the kernel is homogeneous or not. The degree of the polynomial is defined with d. γ is a coefficient which has a default value of 1/number of features if specified as "None". The kernel matrix that is returned is also called the "Gram matrix". In this study, both γ and c values were specified as 1. The X matrix for the MNIST dataset is a huge and sparse matrix and this polynomial function can compute the kernel matrices in a more compact and faster way since it utilizes the "safe sparse dot" which can calculate the dot product of sparse matrices correctly.

For each class (digit), the labels of the training set were encoded using one-versus-all encoding. That is, for each class c_i, a synthetic label set y' is created such that:

$$y' \begin{cases} 1 & y = c_i \\ -1 & y \neq c_i \end{cases}$$

Ten binary classifiers, one for each of the 10 digits, were trained to select one class (digit) against the rest of the classes (other digits) using the appropriate one-versus-all encoded synthetic label sets.

For each binary classifier, separately, for each degree of the polynomial kernel used (from 1 to 6), the algorithm was run for some number of epochs (from 2 to 10) and an ensemble of predictors was collected. For each digit, (1) the predictor achieving the smallest training error among those in the ensemble and (2) the average of the predictors in the ensemble were determined which were then used as the ultimate predictors for multi-class classification.

Two different functions were created to perform these tasks. In particular, both functions take as arguments the relevant kernel matrix (depending on the polynomial degree), the one-versus-all encoded synthetic label set for the digit under consideration, and the number of epochs to be performed. The first function collects for each epoch the weight vector w and the associated number of errors output by the perceptron, then returns the weight vector w, the error of which is the lowest with respect to the others. In a similar manner, the second function collects for each epoch the weight vector w and then computes the average of these weight vectors across the number of epochs and returns that computed vector. These two functions were applied for all the ten digits and ultimately two lists composed of 10 binary classifiers for each digit were created, which were then used for multi-class classification. This was repeated for each degree of the polynomial.

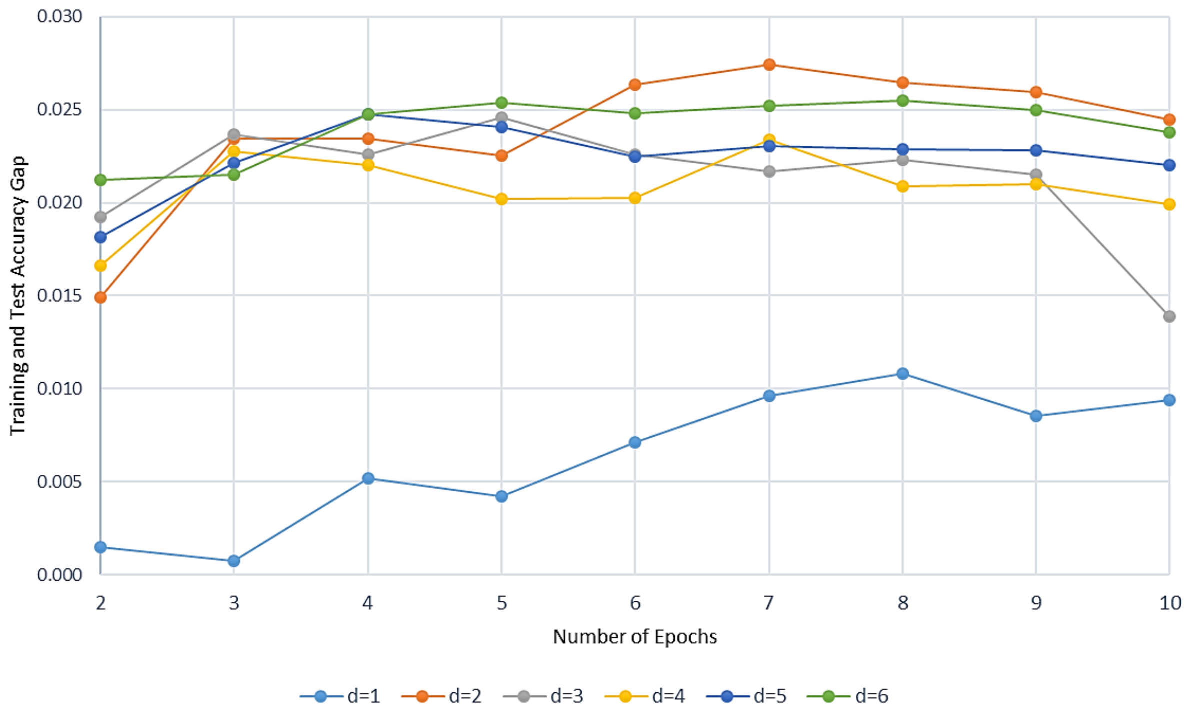


Figure 6: Training and Test Accuracy Discrepancy - Method: Minimum Error

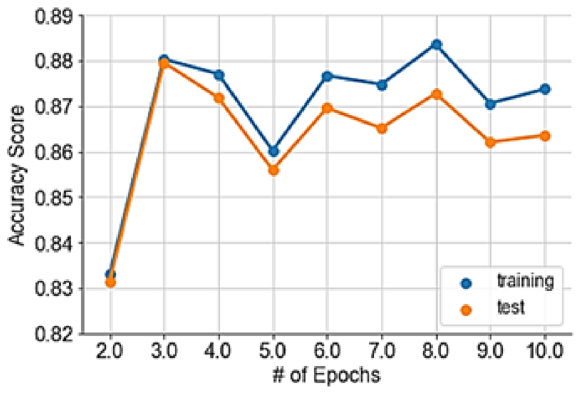
In the multi-class prediction, then, these 10 trained binary classifiers were used in such a way that the class y is predicted using $argmax_i g_i(x)$ where g_i corresponds to the binary classifier for class i. That is, the digit, the binary classifier of which has the maximum score (w.r.t the scores of the other 9 binary classifiers) was predicted as the label of the example that was examined. For both methods described above, using zero-one loss, the multi-class classification performances were evaluated on the test set, for the different values of the number of epochs and the degree of the polynomial.

3.3 Results

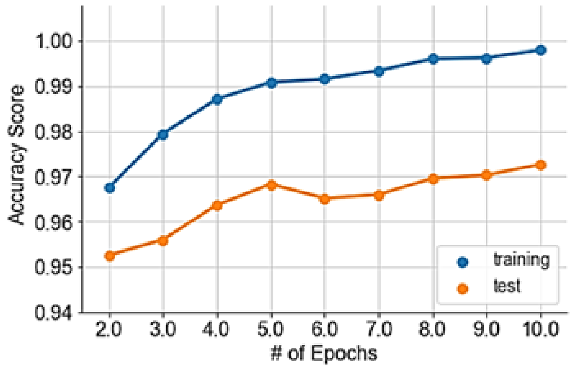
When the discrepancy between the training and the test accuracy is examined, for different polynomial kernel degrees across different numbers epochs, as depicted in Figure 6[4], it can be observed that, if predictors achieving the smallest training error among those in the ensemble are used, the difference is smaller for degree 1, but it tends to increase as the number of epochs increases. Compared to other degrees of the polynomial (2,3,5,6), 4^{th} degree polynomial has a smaller discrepancy which remains around 0.02 for different number of epochs. For the 3^{rd} and 5^{th} degree polynomials, after 5 epochs, the discrepancy values are comparable to the ones obtained with the 4^{th} degree polynomial. In Figure 7, separately for each polynomial degree, the trends of accuracy scores across different number of epochs can be seen.

As presented in Figure 8[5], if the average of the predictors in the ensemble is used, the examination of the discrepancy between the training and the test accuracy for

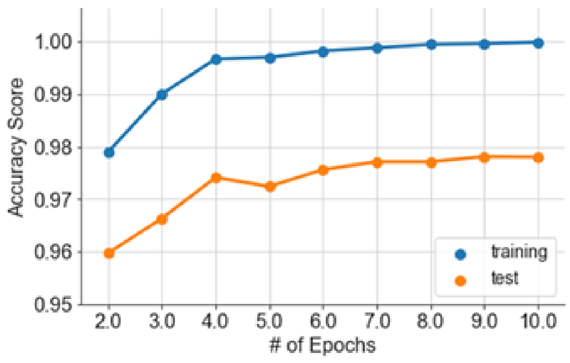
[4]This figure was created using Microsoft Excel.
[5]This figure was created using Microsoft Excel.



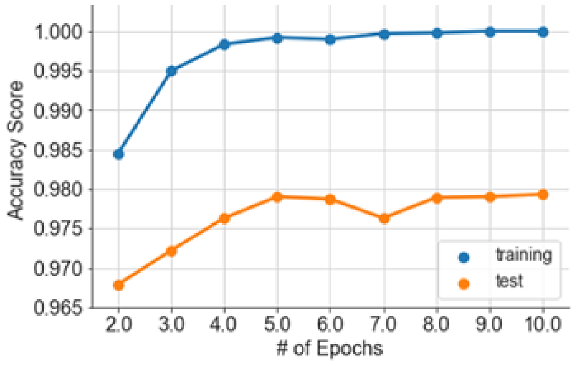
(a) Polynomial Degree ”1”



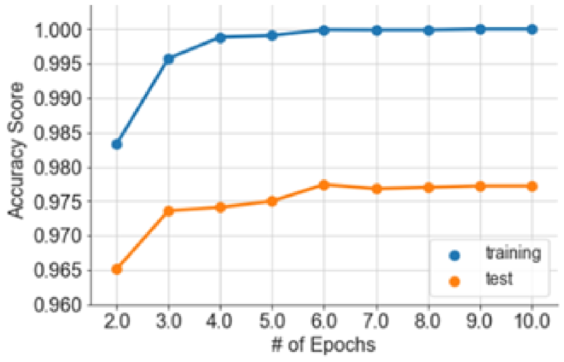
(b) Polynomial Degree ”2”



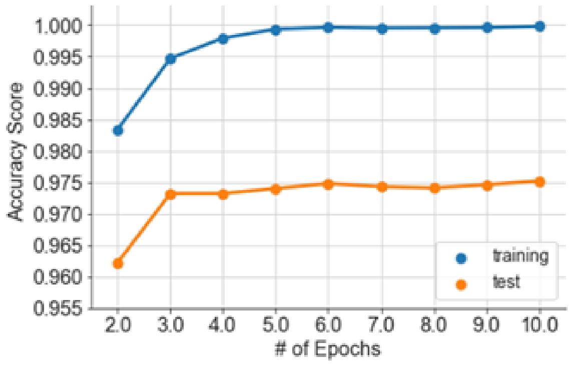
(c) Polynomial Degree ”3”



(d) Polynomial Degree ”4”



(e) Polynomial Degree ”5”



(f) Polynomial Degree ”6”

Figure 7: Training vs Test Accuracy - Method: Minimum Error

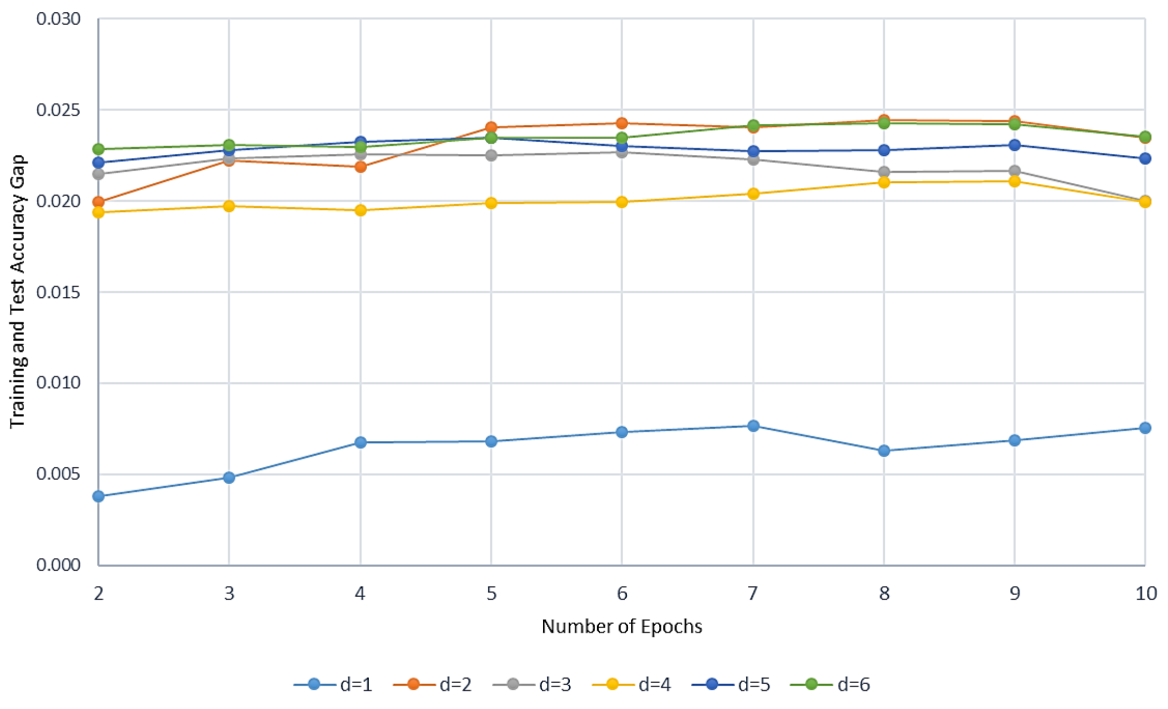
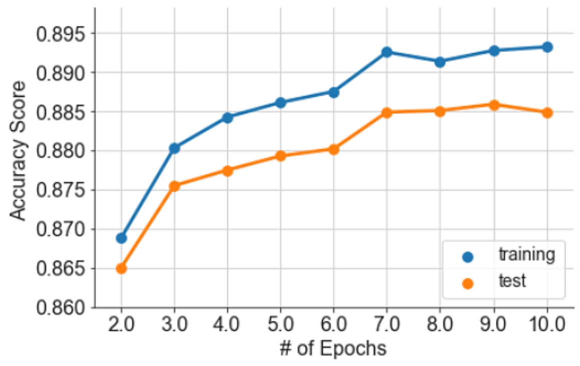


Figure 8: Training and Test Accuracy Discrepancy - Method: Average

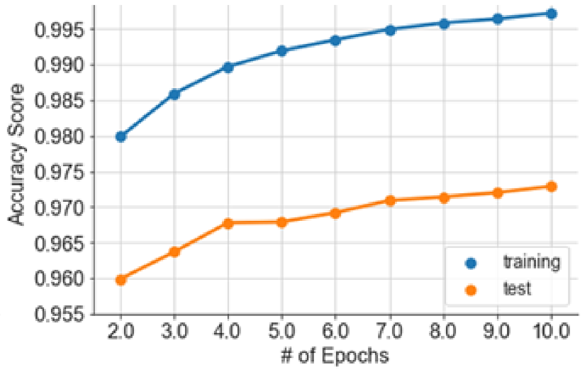
different polynomial kernel degrees across different numbers epochs reveals that, as before, the difference is much smaller for degree 1 but it tends to increase as the number of epochs increases, however, this time with a smaller slope. When the average of the predictors is used, it can be more clearly observed that the 4^{th} degree polynomial has a smaller discrepancy compared to other polynomial degrees and is almost always stable around 0.02 for different number of epochs. For other polynomial degrees on the other hand, the values range between 0.022 and 0.024. For the 3^{rd} degree polynomial, after 7 epochs, the discrepancy values are comparable to the ones obtained with the 4^{th} degree polynomial. As before, separately for each polynomial degree, the trends of accuracy scores across different number of epochs can be seen in Figure 9.

Figure 10 enables a comparison of the multi-class classification performance for different values of the number of epochs and the degree of the polynomial when predictors achieving the smallest training error among those in the ensemble are used on the test set. For almost all different number of epochs, 4^{th} degree polynomial kernel outperforms the others, achieving a 0.98 accuracy score. It can be also observed that the prediction performance for polynomial degrees above 2 are comparable and far better than the degree 1.

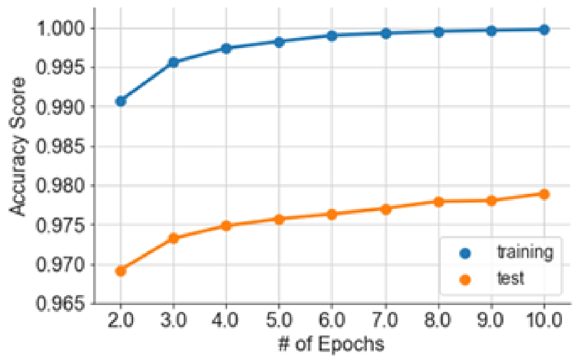
Figure 11, on the other hand, provides a comparison of the multi-class classification performance for different values of the number of epochs and the degree of the polynomial when the average of the predictors in the ensemble is used on the test set. In this case, it is more clear that for all different number of epochs, 4^{th} degree polynomial kernel outperforms the others, achieving again an accuracy score of 0.98. As in the previous case, the prediction performance for polynomial degrees above 2



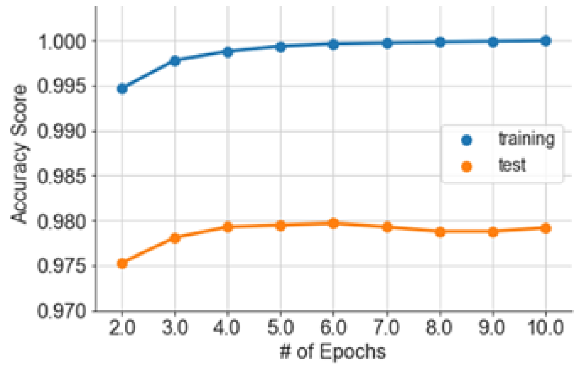
(a) Polynomial Degree "1"



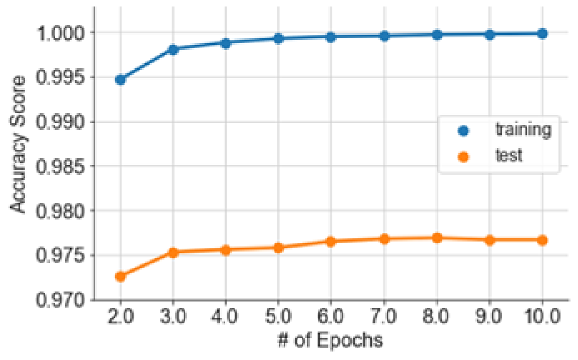
(b) Polynomial Degree "2"



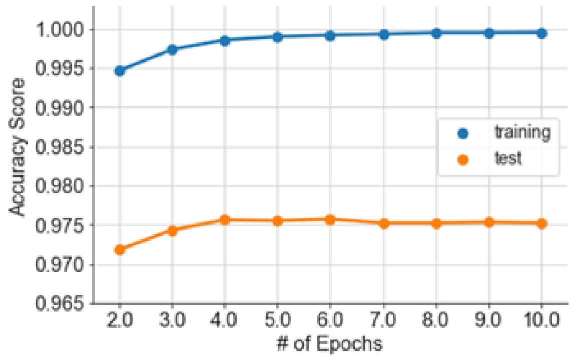
(c) Polynomial Degree "3"



(d) Polynomial Degree "4"



(e) Polynomial Degree "5"



(f) Polynomial Degree "6"

Figure 9: Training vs Test Accuracy - Method: Average

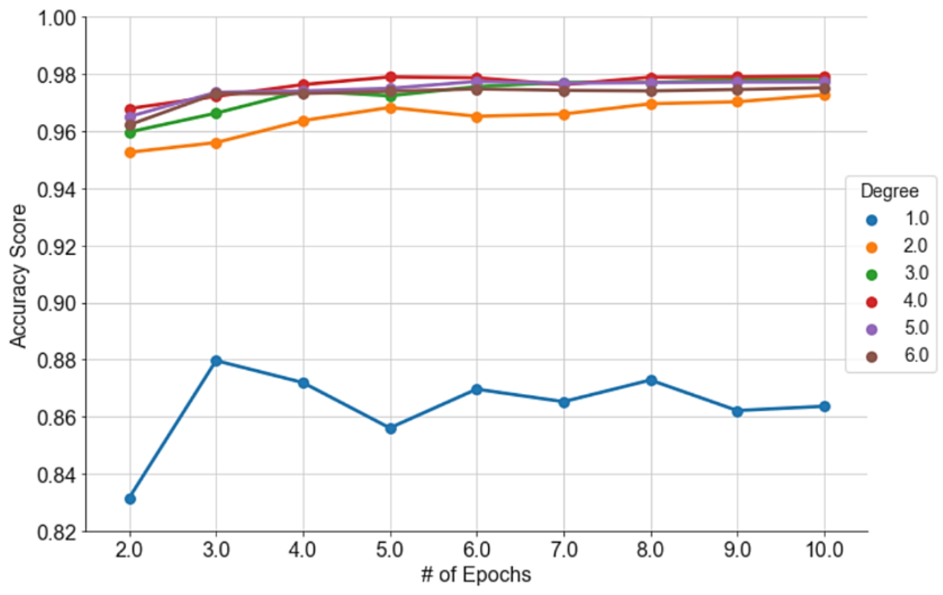


Figure 10: Method - Predictor with the Smallest Training Error

are comparable and far better than the degree 1. The prediction performance of the 2nd degree polynomial kernel is poorer when compared to the degrees of 3, 5, and 6.

Figure 12 presents a comparison of the results for the two different methods employed, (1) using the predictors achieving the minimum error among those in the ensemble and (2) using of the average of the predictors in the ensemble, for the different polynomial kernel degrees and the different number of epochs. It can be observed that using the average of the predictors in the ensemble almost always results in a better multi-class prediction performance than using the predictors achieving the minimum error. Best prediction performance is obtained with the 4th degree polynomial kernel running 6 epochs using the average of the predictors in the ensemble. The accuracy score achieved is approximately 0.98. A comparable performance was also be achieved with 4th degree polynomial running 5 epochs for both predictor selection methods. Similar performance was also observed when the 3rd degree polynomial is used, however, after 10 epochs are completed.

In order to better understand for which of the handwritten digits in the test set predictors made higher number of mistakes, the confusion matrices were computed for both methods. Since for both methods, the 4th degree polynomial kernel outperformed the polynomial kernels of other degrees, here, for illustrative purposes, only the results for that degree were further scrutinized for the number of epochs equal to 4. As it can be seen in Figure 13, most of the misclassifications were incurred for the digits 5, 7, and 9. Digit 5 was mostly misclassified as 3, 6, or 8. Digit 7 was wrongly predicted as 1, 2, or 9 for most of the time. Digit 9 was mostly mislabeled as 4 or 8.

When the two methods are compared, it can be seen that if the "average" method is adopted, the number of misclassified examples decreases for some digits such as 9. For these digits, the number of misclassifications are circled in green in the figure. However, for the digits 4 and 5, the number of errors even increased, which are circled

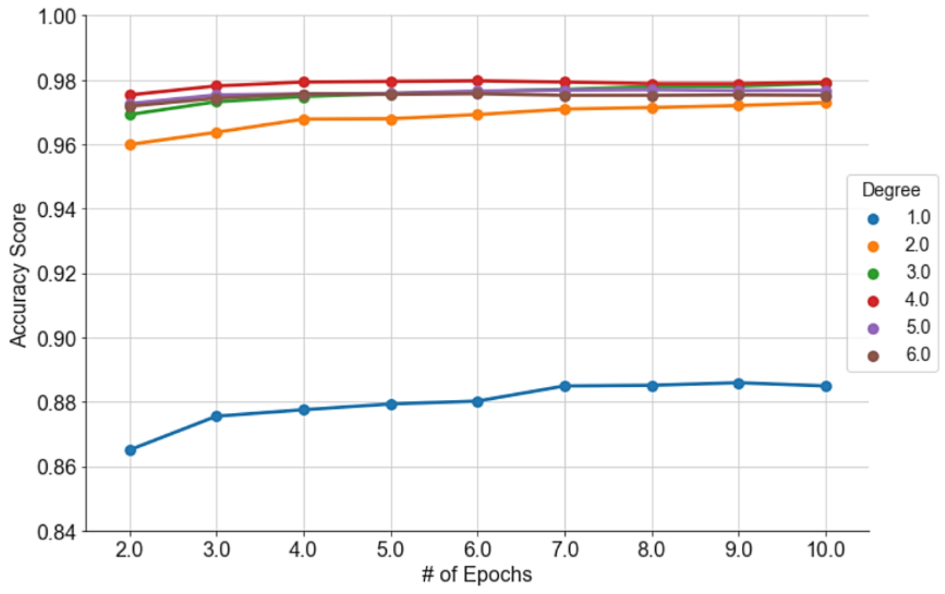


Figure 11: Method - The Average of the Predictors in the Ensemble

in red in the figure. For some of the digits there was a minimal or no improvement in the classification performance. For such cases, the number of misclassifications are circled in yellow. The dashed circular lines emphasize the number of misclassifications that are higher or equal to 10. This kind of comparison can also be conducted for other number of epochs and degrees of the polynomial kernel, in order to have further feedback on the multi-class classification performance.

3.4 Concluding Remarks

Overall, the findings of this study revealed that for multi-class classification of handwritten digits, the use of the average of the predictors in the ensemble was at least as effective as the use of the predictor achieving the smallest training error among those in the ensemble. For both methods, the multi-class classification performance of the perceptron with 4^{th} degree polynomial kernel involved less number of errors than the performance of the polynomial kernels of other degrees. Nevertheless, except the polynomial degrees 1 and 2, for other degrees, the performances were comparable to the 4^{th} degree polynomial kernel.

When the handwritten digits which were misclassified were examined, the results indicated that the classifiers were mostly confused by the digits that in general are drawn in the same way such as 4 and 9 or 5 and 6, which can be evaluated as an admissible error.

In this study, 30000 examples from the MNIST training data were used to train the algorithm. This limitation was due to the memory space related problems encountered when computing the kernel matrices. This problem could be solved by writing more efficient Python codes or using more appropriate computation methods instead of the ones applied in the current study. If the whole dataset could be used, the algorithm would be trained with a more diverse information, which would most probably result in classifiers with better performance.

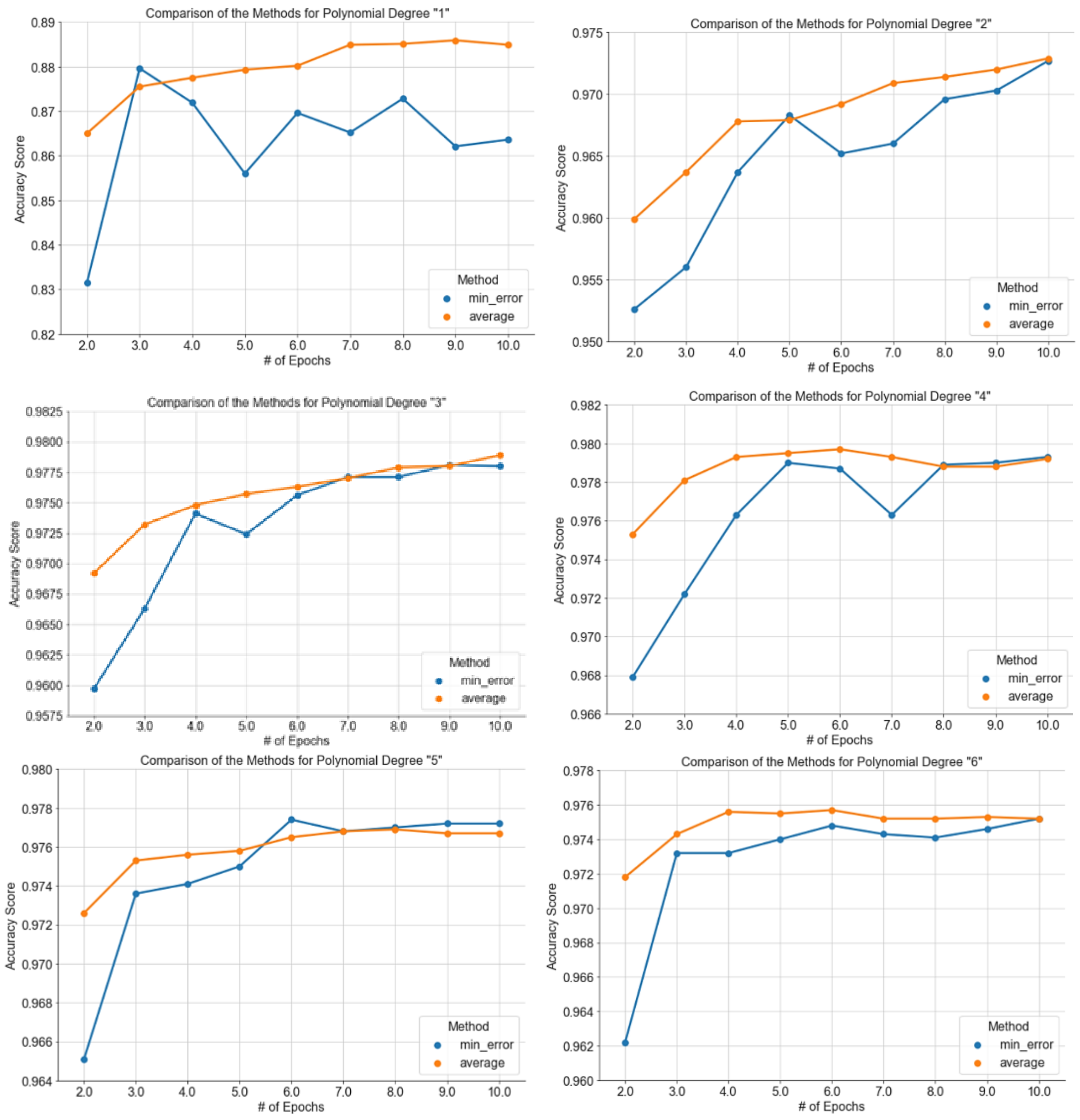
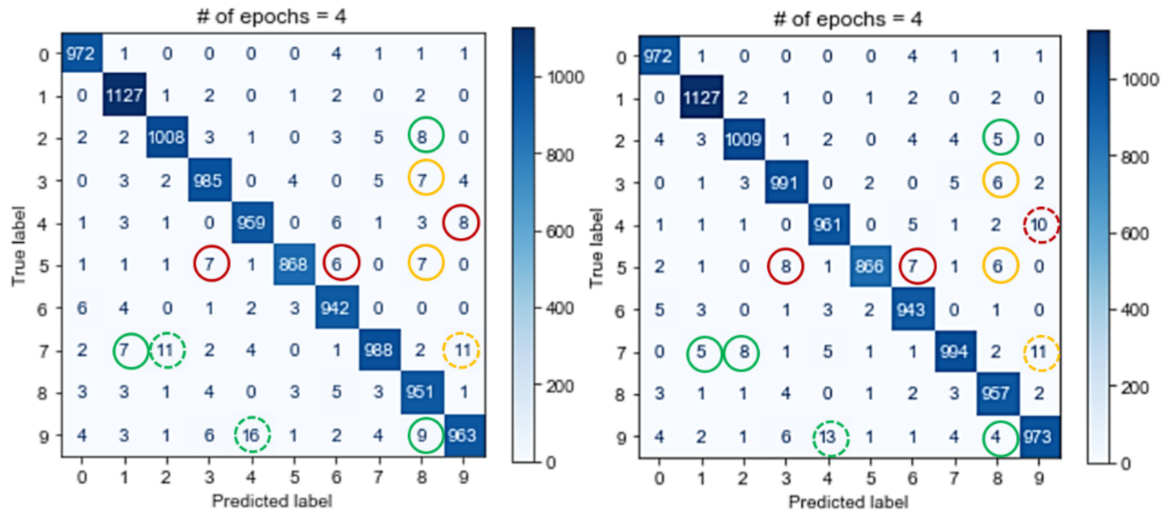


Figure 12: Comparison of the Results for the Two Different Methods



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	972	1	0	0	0	0	4	1	1	1
1	0	1127	1	2	0	1	2	0	2	0
2	2	2	1008	3	1	0	3	5	8	0
3	0	3	2	985	0	4	0	5	7	4
4	1	3	1	0	959	0	6	1	3	8
5	1	1	1	7	1	868	6	0	7	0
6	6	4	0	1	2	3	942	0	0	0
7	2	7	11	2	4	0	1	988	2	11
8	3	3	1	4	0	3	5	3	951	1
9	4	3	1	6	16	1	2	4	9	963

(a) Method: Min. Error

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	972	1	0	0	0	0	4	1	1	1
1	0	1127	2	1	0	1	2	0	2	0
2	4	3	1009	1	2	0	4	4	5	0
3	0	1	3	991	0	2	0	5	6	2
4	1	1	1	0	961	0	5	1	2	10
5	2	1	0	8	1	866	7	1	6	0
6	5	3	0	1	3	2	943	0	1	0
7	0	5	8	1	5	1	1	994	2	11
8	3	1	1	4	0	1	2	3	957	2
9	4	2	1	6	13	1	1	4	4	973

(b) Method: Average

Figure 13: Confusion Matrices for 4th Degree Polynomial Kernel for 4 Epochs

In this study, only polynomial kernel perceptron was implemented. Besides the polynomial kernel, one could also implement Gaussian kernel, which can be conceived as a linear combination of polynomial kernels. Gaussian kernel is more powerful in classification by allowing learning hyperplanes in infinitely many dimensional space. Polynomial kernel has a limitation, in the sense that, when the degree of the kernel is fixed, the curves used are then bounded by that degree. However, for Gaussian kernels, this is not the case and it is possible to approximate any separating function in the original feature space.

4 My Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.