Statistical Methods for Machine Learning ProjectDigit classification with the Kernel Perceptron

Rishav Mondal
Matriculation: 963810

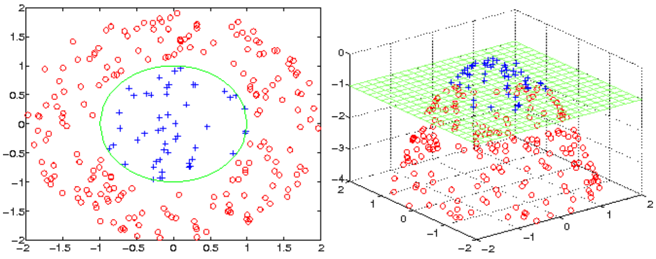
I declare that this material, that I am submitting, is entirely my own work and has not been taken from the work of others, save to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences 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 submitted by me or any other person for assessment on this or any other course of study.

Abstract**—** In this project we implement the Kernel Perceptron method built from scratch for performing the Multiclass-Classification one a dataset using the One-vs-All approach. The Dataset used in this case is the *MNIST* Dataset. The prediction is performed using a varying number of *epochs* and *degree of polynomial*. Here two different predictors are used for the predictions, they are: the average of all the predictors and the predictor with the smallest training error. It is found that both the predictors have similar results on both the varying *epochs* and *degree of polynomial*. With the best result being obtained for *Degree = 2* and *epoch = 6 or 8*. The worst performance is received at *Degree = 3* and *epoch = 1*.

I. Introduction

Classification tasks make a very good use of linear predictors, since it is so convenient to use them, but in reality, the predictor for a given learning problem is far from being a linear function, as it would otherwise lead to high bias. So, to reduce bias, we commonly use techniques such as feature expansion in which higher-level features can be obtained from already existing features coupled with it the feature vector.

Practically, the aim is to identify the hyperplane which divides between different classes in the expanded dimensional space, as shown in Figure 1. In this way, it is possible to learn more complex predictors in the original space like circles and parabolas. However, with the increase in dimensionality the risk of overfitting also increases.



(a) Input Space (b) Projected Space

Fig. 1: Image representation of the kernels, source: semanticscholar.org

This method, however, has some major drawbacks, that is, the computation cost increases with the increase in number of dimensions, since we would have to compute the coordinates of each point in the augmented dimensional space. Using *Kernels*, however, we can overcome this problem of performing these operations, thereby reducing the complexity while obtaining the same results.

II. Dataset

The dataset that was used is the *MNIST Dataset*, which is a large dataset of handwritten digits from **0** to **9**. It is provided by *Kaggle.com***.** The images here have 785 features for each observation: the first feature is the target label, the remaining 784 represent a 28 X 28-pixel image represented as a gray scaled value 0-255 for every single handwritten digit. It is available in image format which must be pre-processed and separated in training and test dataset for further usage, it is also available to be used in a pre-processed **CSV** format, where the training and the test dataset has already been adequately separated.

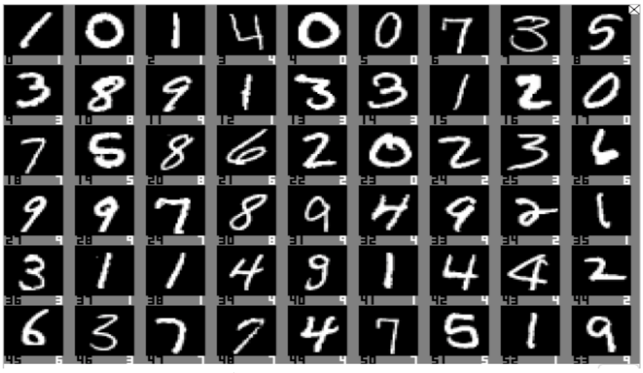
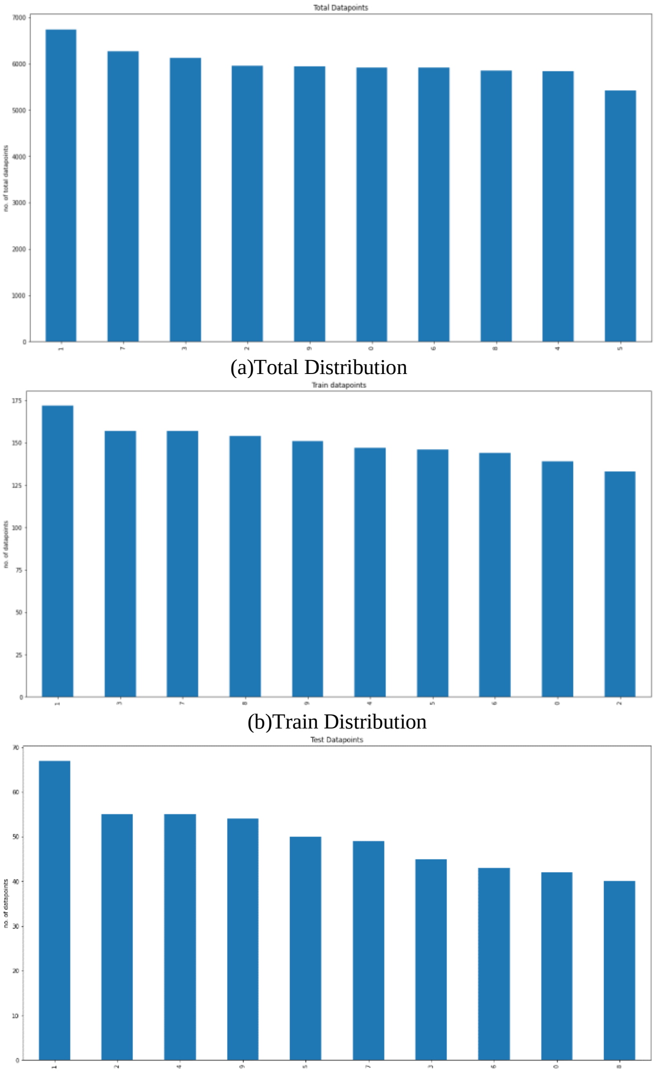


Figure 2: Visualization of the handwritten images, Source: Kaggle.com

For this project analysis, the preprocessed dataset has been used, also available on *Kaggle.com*. Since the dataset is large, for convenience only a fraction of the dataset has been used, with 1500 datapoints used in training and 500 datapoints used in testing.



(a)Total Distribution

(b)Train Distribution

(c)Test Distribution

Figure 3: Distribution of datapoints in the datasets used for analysis

The above graph shows the distribution of the datapoints in the **MNIST Dataset.**

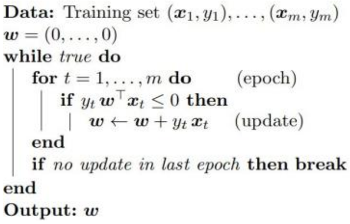
III. THEORITICAL BACKGROUND

The analysis exploits the potential of kernels to perform feature expansion to obtain possibly better and more complex predictors, with respect to the linear ones, which might work better with the dataset in use.

When performing features expansion, the aim is to create new features starting from the existing ones to better identify possible important relationships between the features themselves and the target variable. In formulas, this consists in defining a function to expand feature φ : R$^d \rightarrow R^N$ with $N >> d$. For example, considering a binary classifier h : R$^d \rightarrow$ {1, 1}, with $h(x) = sgn(w^T \ \varphi(x))$ as linear in the expanded feature space, we can define the product as

$$w^T \phi(x) = \sum_{i=1}^{N} (w_i \phi(x)_i) \qquad (1)$$

Focusing on the Perceptron Learning Algorithm, the most common application of this algorithm allows to linearly separate data points in feature space and perform binary classification tasks. The Perceptron algorithm aims at finding a homogenous separating hyperplane by iterating through the training examples one after the other and computing the best values for each weight to classify correctly all inputs. The current classifier is tested on each training example and, in case of miscalculation, the associated hyperplane is adjusted.



```
Data: Training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)$
$\boldsymbol{w} = (0, \ldots, 0)$
while true do
    for $t = 1, \ldots, m$ do        (epoch)
        if $y_t \boldsymbol{w}^\top \boldsymbol{x}_t \leq 0$ then
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_t \boldsymbol{x}_t$    (update)
    end
    if no update in last epoch then break
end
Output: $\boldsymbol{w}$
```

Figure 4: Perceptron Algorithm for linearly separable cases

The linear classifier obtained from Perceptron are in the form

$$h(x) = sgn(w^T x) = sgn\left(\sum_{s \in S} y_s x_s^T x\right) \qquad (2)$$

where S is the set of training data point on which the Perceptron algorithm made an update.

Applying feature expansion to the Perceptron algorithm we get that the classifier of Perceptron in R^N stores a subset of its training examples x_i, associates with each a weight α_i, and makes decisions for new samples x^{i} by evaluating

$$h_\phi(x) = sgn\left(\sum_{i}^{n} \alpha_i y_i \phi(x_i)^T \phi(x')\right) \qquad (3)$$

Computing $\varphi(x_i)^T \ \varphi(x^{\text{i}})$ would require firstly to convert all data points to the new expanded dimensional space and then perform the dot product between each two vector leading to a time complexity of $O(n^2)$. Using kernels , we can perform this operation reducing the complexity while obtaining the same result.

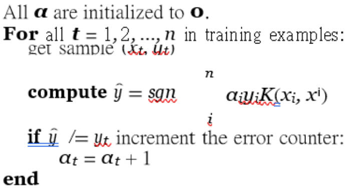
The ***Kernel Trick*** allows to compute this product in a more efficient manner: the dot product can be computed without even transforming the observations into the expanded dimensions and so the required time is just $O(n)$.

The kernel function used in this analysis, which is the polynomial kernel, is defined as $K_n(x, x^{\text{i}}) = (1 + x^T x^{\text{i}})^n$ with $K_n(x, x^{\text{i}}) = \varphi(x) \ \varphi(x^{\text{i}})$, where $\boldsymbol{n}$ represents the degree of the polynomial.

Replacing the dot product of equation (3) with the kernel function, the Kernel Perceptron classifier becomes:

$$h_K(x) = sgn\left(\sum_{i}^{n} \alpha_i y_i K(x_i, x')\right) \qquad (4)$$

The modified pseudo-code for the Kernel Perceptron eventually can be written as follows:



```
All $\boldsymbol{\alpha}$ are initialized to 0.
For all $\boldsymbol{t} = 1, 2, \ldots, n$ in training examples:
    get sample $(x_t, y_t)$
    compute $\hat{y} = sgn \sum_{i}^{n} \alpha_i y_i K(x_i, x^{\text{i}})$
    if $\hat{y}$ /= $y_t$ increment the error counter:
        $\alpha_t = \alpha_t + 1$
end
```

The Perceptron Algorithm was designed as a binary classifier and therefore , as it is in the pseudo-code above, it does not “natively” support classification for more than two classes. However , it can be slightly modified to support it.

One approach to achieve multi-class classification, is the one used in this analysis, is to split the multi-class classification datasets and fit a binary classification model for each label. Then , merge this classifier to make multi-class predictions.

There are different techniques to perform this merge. In this project, the met hod used is the One vs All: once all binary classifiers are trained, predictions are made using the most confident model exploiting the fact that each Perceptron binary classifier has the form $h(x) = sgn(g(x))$ and therefore multi-class predictions can be obtained using:

$$\hat{y} = \underset{i \in 1, \ldots, I}{\operatorname{argmin}} \, g_i(x) \qquad (5)$$

meaning that we apply all classifiers g_i to an unseen sample x and predict the label i for which the corresponding classifier reports the highest confidence score.

But this approach however has some issues. Firstly, it requires training a classifier for each class and it can be very slow when the dataset has many classes like *MNIST.* Second, when transforming the multi-class dataset into different binary classification datasets, the label distribution becomes highly unbalanced since the number of negatives is usually much higher than the number of positive labels. Moreover, the scale of confidence can vary between each classifier leading issues when merging them.

Finally, the Zero-One-Loss is used to evaluate the performance of the Kernel Perceptron in multi-class classification tasks. It is defined as:

$$L(\hat{y}, y) = \begin{cases} 1, & \text{if } \hat{y} = y \\ 0, & \text{if } \hat{y} \neq y \end{cases} \tag{6}$$

This loss function is a very common metric in classification tasks. For each observation, it assigns 0 for correct classification and 1 for an incorrect classification. The total loss is then computed by summing the assigned values and dividing this sum by the total number of observations.

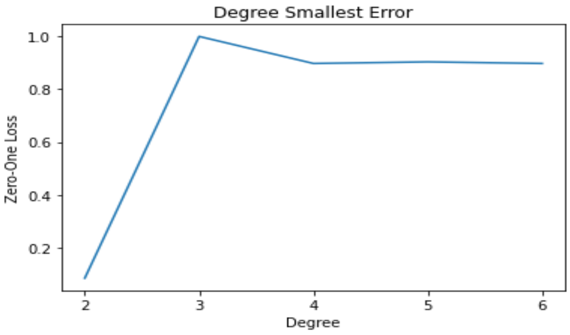
The Accuracy metric is also reported in reported in the report, and it corresponds to the inverse of the Zero-One-Loss as it is computed as the mean of Zero-One-Loss function.

IV. Analysis

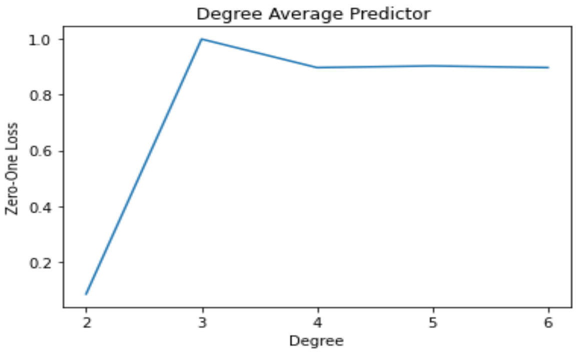
After the already processed data, that is the dataset was already separated into *training* and *test* sets, was imported, the binary classifiers are created. To each binary classifier is passed a random permutation of the training set and the corresponding kernel matrix which is computed by the *kernelMatrix()* method for a given exponent, and the polynomial kernel is calculated using the *poly_ker()* method: this method allows to compute -for training part of the dataset- just the top half of the kernel matrix and then reflect it in order to increase the efficiency. This matrix is computed once for all the 10 binary classifiers for each exponent.

Then, each of the 10 binary classifiers used in this analysis, one for each label 0 to 9, is trained cycling through the training set following the Kernel Perceptron pseudo-code. The algorithm is run for several epochs over the training dataset and the ensemble of predictors determined by the algorithm for each data point is collected.

Two different methods are defined: first, ***train_smallest_error()*** uses the predictor which achieves the smallest training during the training phase, among the ones in the ensemble, and second, ***train_average_predictor()*** which uses the average of the predictors in the ensemble.
The performances for each of these two predictors are then analyzed in relation with two predictors are then analyzed in relation with two parameters. The first one is the relation between the *Zero-One-Loss* of the predictions against the *Polynomial Degree*. The second one is the relation between *Zero-One-Loss* of the predictions against the number of *Epochs*



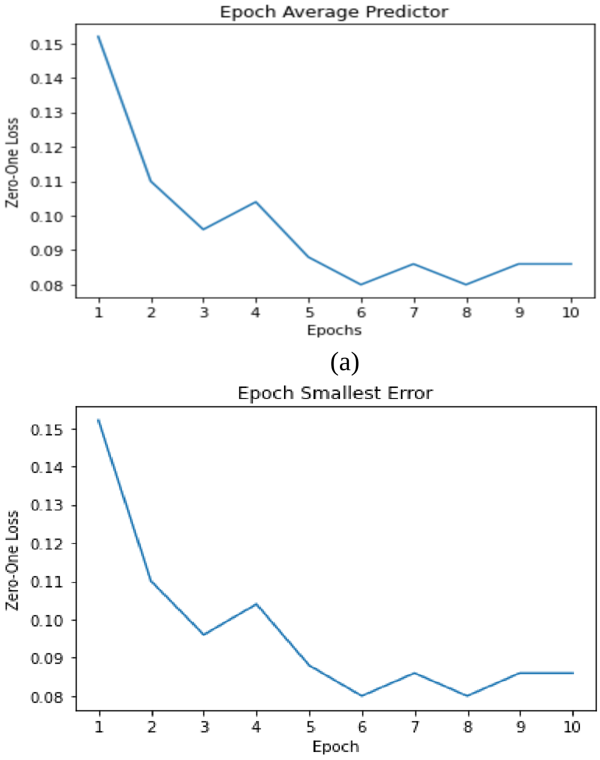
(a)



. (b)

Fig. 4: Zero-One-Loss against Degree

As we can see from the above graphs the best performances of the predictors, over a rannge of 10 epochs, has been achieved at ***Degree=2,*** and the worst performance of the of the predictors are achieved at ***Degree=3,*** we can also see that the performances of both the predictors are more or less the same.



(a)

(b)

Fig.5: Zero-One-Loss against Epochs , given Degree = 2

After finding out the degree with the best result, I used ***Degree=2,*** in my next set of experiments, where I fixed a degree of polynomial, and found the zero-one-loss against the epochs. After running the two predictors we found in this case as well both predictors returned surprisingly similar results. With the worst performance coming at ***Epoch=1*** and the best coming at ***Epochs= 6,8***.

All this proves that we would receive the best result of training when the ***Degree=2*** and the num of epochs are limited to ***Epochs=8***.

CONCLUSION

On the basis of this analysis we can now assess performance of the Kernel Perceptron classification algorithm with both types of predictors: the average of all the predictors, and the predictor with the smallest training error. As reported in the ***Analysis,*** section , the two predictors perform, surprisingly, in the same way apart from very minimal differences in the outcomes for what concerns the error-degrees relation. In both cases it is evident that the most suitable polynomial degree is 2, for which the algorithm presents very good predictive capabilities, and it is able to separate in an effective manner the labels in each binary classifier.

For this degree, thanks to the kernel trick, it is possible to successfully learn a classifier which is not linear and exploit this capability to learn, in this case, classifier up to second degree is more reliable.

However, the predictive abilities, with the other exponents are very poor and, for what concerns the relation between prediction errors and epochs, it is important to highlight an increase in the training error as the epochs increases.

APPENDIX

Link to the GitHub repository:

https://github.com/RishavMondal/Multiclass_Classification_Kernel_perceptron