Statistical Methods for Machine Learning Project
Digit classification with the Kernel Perceptron

Luca Romano, mat.980068

October 2021

Abstract: This project presents an implementation of the Kernel Perceptron built from scratch to perform multiclass classification with a one vs all approach. Its performances are evaluated on the *Mnist Dataset* by varying the number of epochs and the degree of the polynomial using two different predictors for each binary classifier: the average of the predictors of all epochs and the predictor which, among all epochs, minimizes the training error.
With both predictors, results are -surprisingly- almost the same: the relation between errors and the degree of the polynomial shows very good results for *degree* $= 2$ and very poor ones for the other degrees.
The relation between errors and epochs reaches, in both predictors, its best performance at the 6th cycle over the training set keeping almost the same accuracy when increasing the numbers of epochs.

1 Introduction

Linear predictors are very used in classification tasks since it is convenient to use them but, in general, the predictor for a given learning problem is far from being linear and therefore applying a linear classifier would lead to high bias. To reduce the bias, a commonly used technique is feature expansion in which higher-level features can be obtained from already available features and added to the feature vector.

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

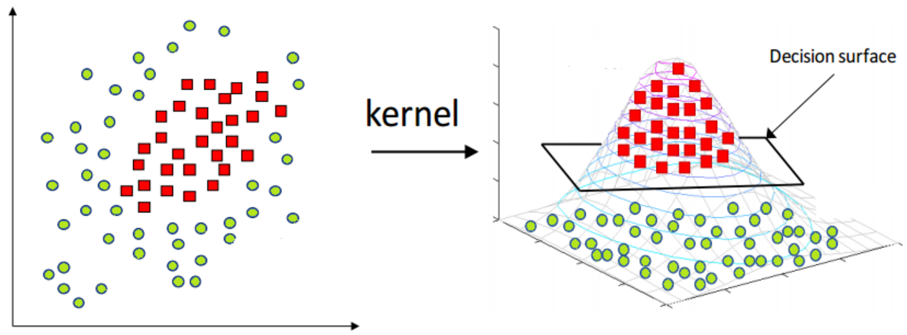


Figure 1: Image Representation of Kernels, source: Medium.com

The drawback of this technique is that by increasing the dimensions, the computations become more and more expensive: we would have to compute the coordinates of each point in the augmented dimensional space. Using kernels, we can overcome this issue and perform these operations reducing the complexity while obtaining the same results.

2 Dataset

The dataset comes from the MNIST Dataset from *Kaggle.com*, a large database of handwritten digits from 0 to 9. It has 785 features for each observation: the first feature is the target label, the remaining 784 represent a 28 X 28 pixel image represented as grayscaled value 0-255 for every single handwritten digit.

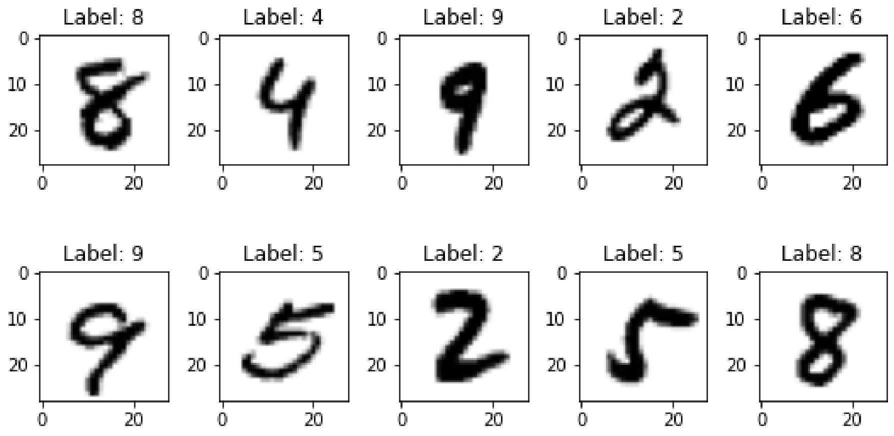


Figure 2: Plot of the first 10 training observations

For this analysis it was used just a fraction of the total number of data point contained in the dataset: among the total 10000 observations of the *test_dataset.csv* provided, were used 20% of them leading to a total of 2000 observations. The 70% of these 2000 observations was dedicated to the training of the Kernel Perceptron algorithm, the remaining 30% to the test.

3 Theoretical Background

This analysis exploits the potential of kernels To perform *features 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 $\phi : \mathbb{R}^d \to R^N$ with $N >> d$. For example, considering a binary classifier $h : \mathbb{R}^d \to \{-1, 1\}$ with $h(x) = sgn(w^T \phi(x))$ as linear in the expanded feature space, we can define the product as

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

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 homogeneous 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 linear classifier is tested on each training example and, in case of misclassification, 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 3: Perceptron Algorithm for linearly separable cases

The linear classifier obtained from Perceptron are in the form

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

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 $\mathbb{R}^N$ stores a subset of its training examples x_i, associates with each a weight α_i, and makes decisions for new samples x' by evaluating

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

Computing $\phi(x_i)^T \phi(x')$ 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 $K_n(x, x') = (1 + x^T x')^n$ with $K_n(x, x') = \phi(x) \cdot \phi(x')$, where n represents the degree of the polynomial.
Replacing the dot product of equation (3) with the kernel function, the Kernel Perceptron classifier becomes:

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

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

```
All $\boldsymbol{\alpha}$ are initialized to $\mathbf{0}$.
**For** all $\boldsymbol{t} = 1, 2, ..., n$ in training examples:
    get sample $(x_t, y_t)$
    **compute** $\hat{y} = sgn\left(\sum_i^n \alpha_i y_i K(x_i, x')\right)$
    **if** $\hat{y} \neq y_t$ 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 -the one used in this analysis- is to split the multi-class classification dataset into multiple binary 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 method 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:

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

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.

This approach however presents some issues. Firstly, it requires training a classifier for each class and it can be very slow when the dataset has many classes like the 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 to 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 the report, and it corresponds to the inverse of the zero-one Loss as it is computed as the mean of the one-zero loss function.

4 Analysis

4.1 Implementation

After the pre-processing phase in which the dataset is split into training and test set, 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 *symm_kernelMatrix()* method for a given exponent: this method allows to compute -for the training part of the dataset- just the top half of the kernel matrix and then reflect it in order increase the efficiency. This matrix is computed once for all the 10 binary classifiers for each exponent.

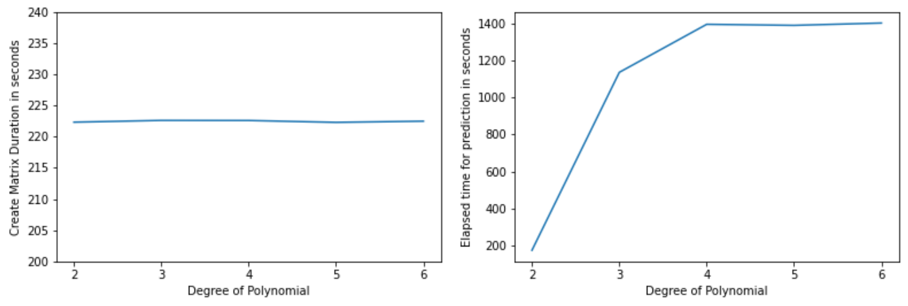
Then, each of the 10 binary classifiers used in this analysis -one for each label from 0 to 9- is trained cycling through the training set following the Kernel Perceptron pseudo-code. The algorithm is run for several epochs over the training dataset and the ensemble of predictors determined by the algorithm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

Two different training methods are defined: the first one, *fit_average()* uses the average of all predictors in the ensemble, the second one, *fit_smallest()* uses the predictor which minimizes the errors during the training phase.

The performances for each of these two predictors are then analyzed in relation with two parameters. Firstly, in the relation between their predictions accuracy and the corresponding zero-one loss against the polynomial degree. Second, in the relation between predictions accuracy and the corresponding zero-one loss, against and the number of epochs.

4.2 Results

The training time of the Kernel Perceptron takes many seconds, in particular due to the creation of the kernel matrix. However, as shown in figure 4(a) below, it is important to notice that due to the kernel trick the amount of time required to build a matrix for the same permutation of the training set is almost constant despite the different degree of the polynomial.



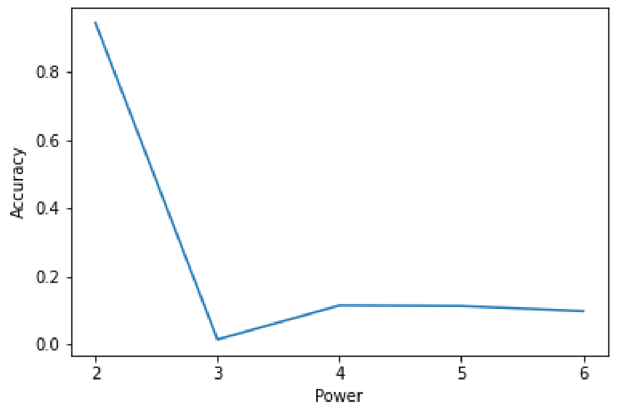
(a) To create the Kernel Matrix, by degree (b) For prediction task, by degree

Figure 4: Time elapsed

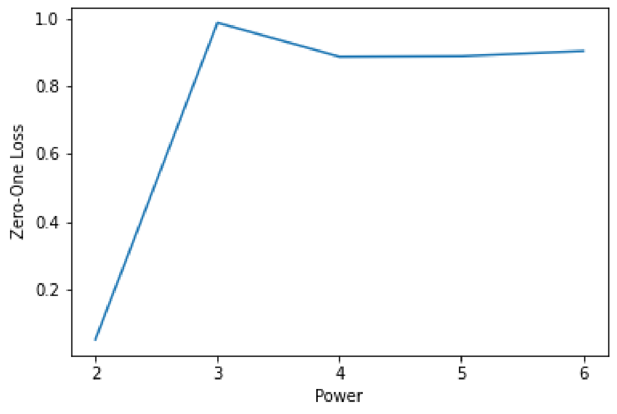
The time required to perform prediction however is far from constant and varies a lot when changing the degree. In particular, with $power = 2$ the time required is a lot less than with the other exponents as shown in Figure 4(b).

For what concerns the predictive power with respect to both the degree of the polynomial and the number of epochs, the two type of predictors produces, unexpectedly, the exact same results with just very few differences.

With respect to the relation between polynomial degree and errors it is evident that the best degree is 2. As reported in figure 5, in this case the errors are just 33 for 600 test observations with an accuracy of 0.945 and a zero-one loss of 0.055 for both predictors. All the other degrees from 3 to 6 provide bad results with respectively 592, 533, 532, 541 errors for the average of predictors and 592, 532, 533, 542 for the other. Therefore, their predictions have poor accuracy



(a) Accuracy - Degree, epochs = 10



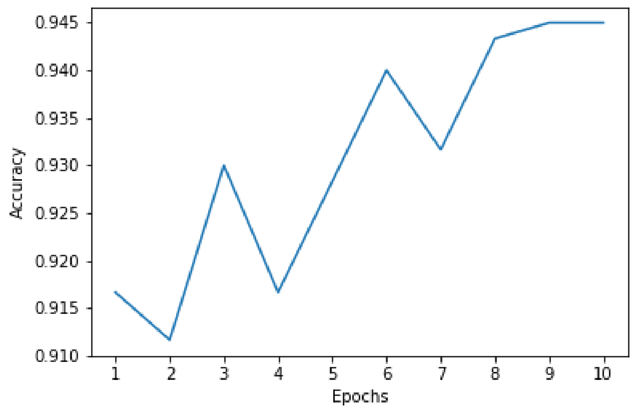
(b) Zero-One Loss - Degree, epochs = 10

Figure 5: Number of correct predictions against degree of the polynomial

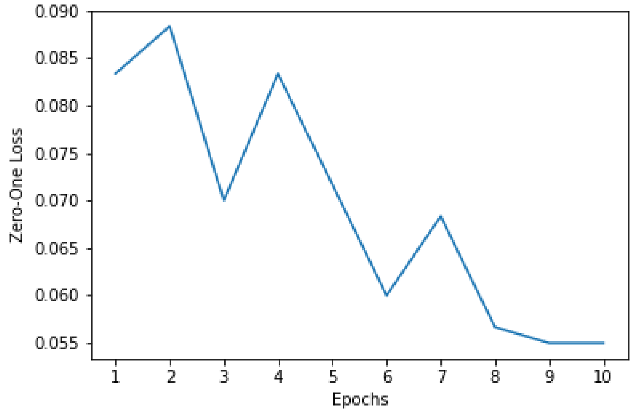
-below 10% in two cases- and a high zero-one loss, in particular with $degree = 3$ where it is over 98%.

Given these results, $degree = 2$ is the one used to analyze the relation between errors and number of epochs. Looking at the resulting predictions, the outcomes are identical for both predictors: for each epoch from 1 to 10 the results are respectively 50, 53, 42, 50, 43, 36, 41, 34, 33, 33.
Looking at the accuracy of the fitting for the training portion, it is possible to notice some differences in the two predictors, with the one averaging the predictors having an accuracy equal to one in all the ten binary classifiers at epoch

9 and 10, but nevertheless the results obtained for the multi-class prediction do not differ in any way.



(a) Accuracy - Degree, epochs = 10



(b) Zero-One Loss - Degree, epochs = 10

Figure 6: Number of correct predictions against number of epochs

As reported in Figure 6, in both predictors the least number of error is achieved at the 9th training cycle with an already good result after 6 cycle.
The tests conducted with other degrees analyzing the predictions for different numbers of epochs do not allow to achieve any good test-score. On the contrary, as reported in the output of the degree-errors relation for the predictor minimizing the training error (see output in the code linked in appendix), as the number of epochs increases the training accuracy seems to slightly decrease for all the binary classifiers. In the case case of $degree = 2$ this behaviour do not happen

and, as epoch increases, the training accuracy grows towards 1. Considering the relation between erroneous predictions and epochs for $degree = 6$, the predictor obtained as the average of predictors provides the following amount of errors as result: 544, 541, 545, 546, 543, 545, 545, 543, 543, 542. These numbers lead to an accuracy below 10% in all epochs and all the zero-one losses above 90%.

5 Conclusions

On the basis of this analysis we can now assess the performance of the Kernel Perceptron classification algorithm with both types of predictors: the average of predictors among all epochs and the one that minimizes the training error. As reported in the *Results* section, the two predictors perform, surprisingly, in the same way a part 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, classifiers up to second degree in a reasonable amount of time and without using unusual computing resources.

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.

6 Appendix

Link to Github repository:

https://github.com/rluca96/Kernel-Perceptron