Data Science and Economics
Department of Economics, Management and Quantitative Methods
Department of Computer Science "Giovanni degli Antoni"
Università degli Studi di Milano

MNIST digit classification with Kernel Perceptron

Machine learning

Shlyk Darya - 965162

a.y. 2020/2021

Contents

Abstract

The purpose of the project is to implement the kernel perceptron algorithm to address a multi-class classification problem. We are provided with the MNIST dataset consisting of 10000 images of handwritten digits, ranging from 0 to 9. Our task is to design a predictor able to classify a given image into a correct digit class. With 10 integer digits, there are 10 possible classes for each image. The strategy we will use to solve the problem comprises two steps. First, we learn a binary classifier for each digit, using a kernalized version of the Perceptron algorithm. In order to make multi-class predictions, we define a model that predicts with the class of the binary classifier obtaining the highest activation on a current example.
With the Perceptron, the learning is performed in online fashion. The learner accesses training examples one at a time, and updates the parameters of the classifier on the basis of mistakes made on the observed data. As a result of the training process, the algorithm generates a sequence of binary classifiers. Depending on the method followed to extract the final predictor for binary classification, several variations of the Perceptron algorithm can be adopted.

In our analysis we consider three implementations of the Kernel Perceptron. First, we study the influence of the polynomial degree on the training score and the validation score for the baseline model, the Vanilla Kernel Perceptron. Next, we analyse two variants of the algorithm. Namely, a model minimizing the training error over the sequence of binary predictors generated in the training phase, and an algorithm predicting binary labels with an average classifier. All the models are validated for different values of the number of epochs and polynomial degree, using the zero-one loss.

Introduction

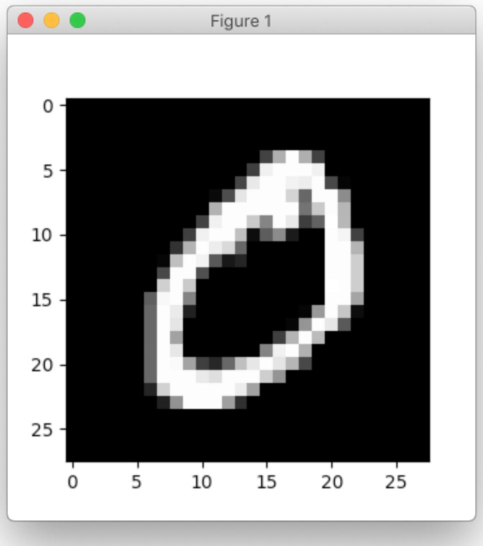

Figure 1: Example of a handwritten digit.

Preprocessing

In the dataset we retrieved from Kaggle, each image (28×28 pixel grayscale square) is already linearized into a 785 component vector, with the first value corresponding to the digit label (a number from 0 to 9) and the remaining 784 values are pixels (a number from 0 to 255). We store features and labels in two arrays and preprocess each of them separately.
The feature matrix is rescaled into 0-1 range, by dividing every pixel by 255, which is its maximum value. The bias is treated as a constant feature and folded into the feature matrix as a vector of ones. We apply a one-versus-all encoding to represent 10 digits in the label vector. In this way each digit has it own binary representation in a separate vector, in which we use 1 to denote one specific digit class and -1 for all the other classes. Before starting the learning we shuffle the examples to introduce randomization into the training process. Finally, the train-test split is performed, allocating to the test set 30% of the data.

Theoretical background

In all the implementations of the Perceptron algorithm, we will use a kernel-based approach. The method implicitly relies on some feature expansion function ϕ, mapping each data point from the original feature space to a higher-dimensional one. Applying a polynomial function of degree d on the input feature vector will produce an expanded vector, containing all the monomial terms up to degree d. As a result, the algorithm will train a linear classifier on a feature-expanded training set.
To derive the kernel version of the Perceptron algorithm, we rewrite the perceptron weight vector as a linear combination of training examples in the expanded feature space. The resulting equation of the weight vector learned by the algorithm has the following representation:

$$\mathbf{w} = \sum_{i}^{n} \alpha_i y_i \phi(\mathbf{x}_i)$$

- α_i is the number of times $\mathbf{x}_i$ was misclassified;
- $\phi(\mathbf{x}_i)$ is the transformed feature vector for the i^{th} example;
- y_i is the binary label in $\{-1, 1\}$ for the i^{th} example.

Contrary to the standard Perceptron, the kernel version of the algorithm stores all the training examples and, iteratively updates a "mistake counter" vector $\vec{\alpha}$. The above representation for the parameter vector $\mathbf{w}$, allow us to express the algorithm in terms of the inner products of feature vectors. Consequently, the kernel trick can be applied to directly compute the dot products between feature vectors, in this way, avoiding the explicit evaluation of the feature mapping function itself. To practically implement the kernel trick, we define a polynomial kernel function of the form $K(\vec{x}, \vec{x'}) = (c + \vec{x}^T \vec{x'})^d$, which contains all the monomial terms up to degree d. One of the interest point of the current analysis, will be to assess the impact of the specific degree choice on the predictive accuracy of the algorithm.

Perceptron algorithms

Vanilla Kernel Perceptron

The starting point for our analysis, is to consider the simplest version of the kernel perceptron algorithm, which we refer to as Vanilla Perceptron. In this implementation , the learner simply predicts with the sign of the binary classifier resulting from the training process. Before moving to analyze the variants of the Perceptron, it might be useful to define a baseline model and to consider its behaviour on our problem. This way it will be possible to evaluate how the predictive accuracy changes as we modify the initial algorithm. Below we report the algorithm for implementing vanilla kernel perceptron.

Algorithm 1 Vanilla Kernel Perceptron

```
1: Initialize $\vec{\alpha} \leftarrow \vec{0}$
2: Repeat for a given number of epochs:
3:     get next example $(\mathbf{x}_t, y_t)$
4:     if $y_t \sum_i^n \alpha_i y_i K(\mathbf{x}_t, \mathbf{x}_i) <= 0$ then        ▷ Check if misclassified
5:         $\alpha_t \leftarrow \alpha_t + 1$                                  ▷ Increase mistake counter
6: Return $\vec{\alpha}$
```

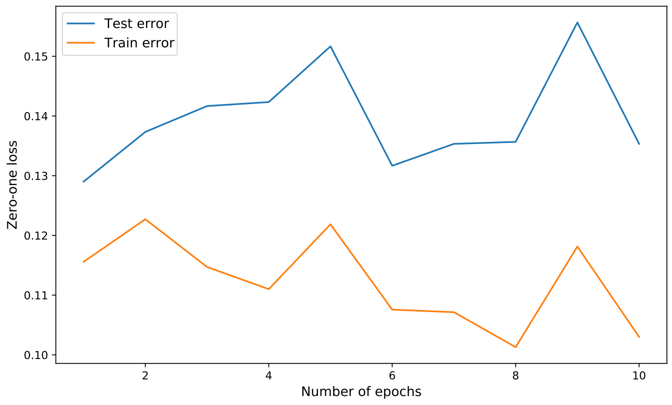

Figure 2: Multi-class Vanilla Perceptron vs number of epochs.

On this first model, we try to motivate the use of the polynomial kernels. If we set the degree of polynomial kernel to one, we are considering the case of an algorithm, which makes no use of polynomial feature expansion. Above we plot the performance of this algorithm as a function of the number of epochs run over the entire training data. It is immediate to notice that both test and training error are relatively high and considerably wiggly across the different number of epochs. To tackle this problem and to reduce the bias, we resort to feature expansion technique. By increasing the degree of the polynomial kernel function, we are constructing new features through a non linear combination of the base features. This way we can learn a hyperplane in a high dimensional space, where the data becomes linearly separable.

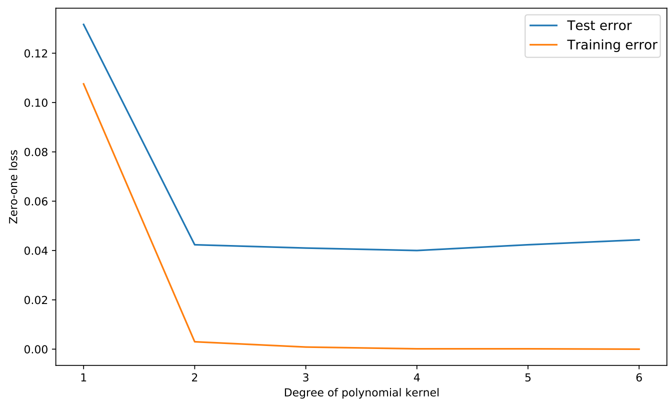

Figure 3: Multi-class Vanilla Perceptron vs degree of polynomial.

We can visualize the impact of the feature expansion by plotting the performance of Vanilla Perceptron as a function of the degree for polynomial kernel. The benefit of the feature expansion is quite evident from what we can see in Figure 3. There is a substantial drop in both, generalization and empirical error, already at the second degree of polynomial, and no considerable further improvement when a higher degree of polynomial is used. Assuming that this result also holds for the variants of the algorithm under analysis, we will use quadratic polynomial kernel to validate the Perceptron algorithms for different number of training epochs. The previous assumption will be tested later on.

Kernel Perceptron variants

In this section we consider two variants of the basic Perceptron algorithm. With respect to the binary classification of a single digit, different strategies can be used to derive the separating hyperplane.

Empirical Risk Minimization Kernel Perceptron

In the first approach we define the separating hyperplane in terms of the binary classifier achieving the smallest training error in the sequence of hypothesis generated at the training stage.
To practically implement this method we need to compare the relative performance on the training data for all the hypothesis learned by Perceptron and to pick the classifier with the lowest training error to perform the binary classification. Since the learning process is designed in a way to minimize the cumulative loss, at each consecutive iteration, the learner will pick a linear classifier so as to be more accurate in the later rounds. To speed up the computations, we derive the minimum training error separating hyperplane by minimizing the training error on the set of classifiers generated in the last epoch of the training process.

Algorithm 2 ERM Kernel Perceptron

```
1:  Initialize $\vec{\alpha} \leftarrow \vec{0}$, $min\ training\ error \leftarrow \infty$
2:  Repeat for a given number of epochs:
3:      get next example $(\mathbf{x}_t, y_t)$
4:      if $y_t \sum_i^n \alpha_i y_i K(\mathbf{x}_t, \mathbf{x}_i) <= 0$ then        ▷ Check if misclassified
5:          $\alpha_t \leftarrow \alpha_t + 1$                                  ▷ Increase mistake counter
6:          $m = \sum_i^n 1\{y_i \neq \hat{y}_i\}$                           ▷ Training error using $\vec{\alpha_t}$
7:          if $m < min\ training\ error$ then
8:              $min\ training\ error \leftarrow m$                        ▷ Update min train error
9:              $\vec{\alpha}_{erm} \leftarrow \vec{\alpha_t}$                               ▷ Remember new ERM classifier
10: Return $\vec{\alpha}_{erm}$
```

Averaged Kernel Perceptron

A very popular alternative to the standard Perceptron, is the averaged version of the algorithm. The learner collects a set of predictors from the training phase and keeps track of their survival time. At the testing stage the binary classifier computes the average of the sequence of predictors learned during training. The contribution of each predictor to the average is proportional to the number of iteration it made a correct prediction. We practically implement this idea by keeping a running sum of the parameter vectors learned by perceptron during the learning process. The algorithm is presented below.

Algorithm 3 Averaged Kernel Perceptron

```
1: Initialize $\vec{\alpha} \leftarrow \vec{0}$, $\vec{\alpha}_{ave} \leftarrow \vec{0}$
2: Repeat for a given number of epochs:
3:     get next example $(\mathbf{x}_t, y_t)$
4:     if $y_t \sum_i^n \alpha_i y_i K(\mathbf{x}_t, \mathbf{x}_i) <= 0$ then      ▷ Check if misclassified
5:         $\alpha_t \leftarrow \alpha_t + 1$                                  ▷ Increase mistake counter
6:     $\vec{\alpha}_{ave} \leftarrow \vec{\alpha}_{ave} + \vec{\alpha}_t$                     ▷ Update the running sum
7: Return $\vec{\alpha}_{ave}$
```

Experimental Results

Training Epoch hyperparameter

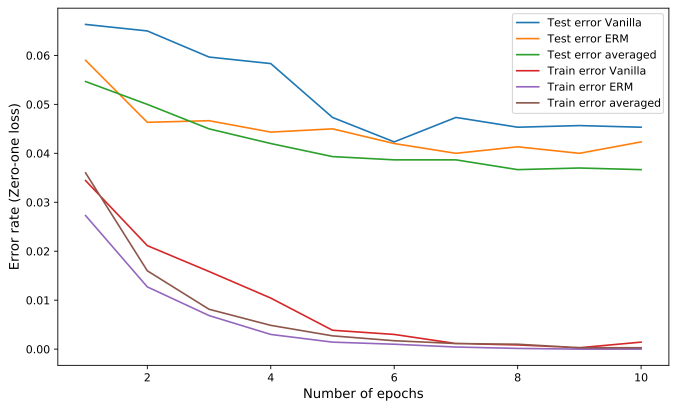

Figure 4: Multi-class kernel perceptron vs number of epochs.

For a given split of the data in training and test set, we are interested in the performance of the ERM and the Averaged Perceptron. In particular, we want to determine how effective are the estimators, output by these algorithm, on the training data, as well as how well they generalize on unseen examples. The plot above represents the evolution of the test and the training error for increasing number of epochs. To provide a complete framework, we also plot the error rate for Vanilla Kernel Perceptron, along with the two variants. To make the models comparable, quadratic kernel has been used. In all three cases, the training error is below the test error and reaches the zero-lower bound at around fifth epoch. For any number of epochs, the variants of the algorithm achieve better results, both in terms of test and training accuracy, with respect to Vanilla Perceptron. The test error rate of the latter stays above the error rates of the variants, achieving its minimum at 0.045 for the number of epochs equal to six.

The Averaged Perceptron generalizes the best. Its test error is the lowest almost for any number of epochs with respect to the other algorithms. It reaches its minimum test error of 0.04 with 8 training epochs and keeps its stable for higher number of epochs. As expected, the ERM Perceptron beats the other two models in terms of training accuracy. Its test performance is comparable with that of the Averaged Perceptron, and only slightly worse and less stable. The ERM Perceptron achieves a higher precision on the training data and converges to zero faster than the other two models, resulting in the lowest relative training error for any considered number of epochs.

Polynomial Degree

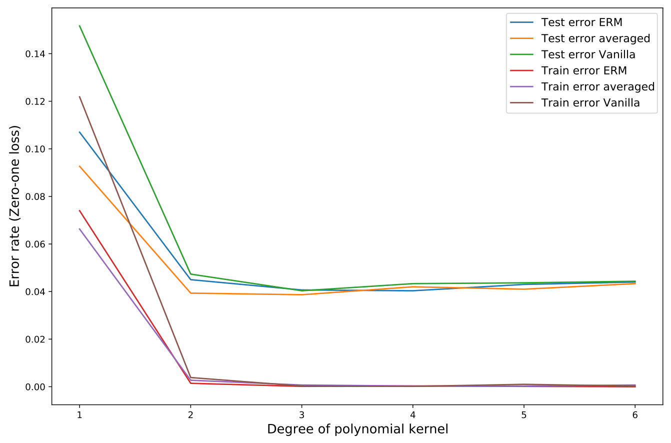

Figure 5: Multi-class kernel perceptron vs degree of polynomial.

On the same split of data, we now evaluate the multi-class classification performance for three algorithms for different values of the degree of the polynomial. In the above plot the training and test error rate are represented as a function of the polynomial degree ranging across integer values between 1 and 6. Differently to what we could have observed in the previous paragraph, the error rate behaves similarly for all the algorithms as we vary the value on this hyperparameter: substantial differences in the performance among the algorithms observable at the first degree of polynomial kernel quickly disappear as soon as we consider higher values of the degree. The error rate considerably plunges already at the second degree of the polynomial, and maintains a stable plateau for the degree values greater than two.

Conclusions

Among the three different versions of the Kernel Perceptron we have considered in this paper, the Averaged Perceptron algorithm proved to have better generalization properties. Its test error is the lowest and keeps stable for higher number of epochs. The algorithm exploits the benefits of the polynomial feature expansion already at the second degree of polynomial kernel. As a sanity check, we report a sample of 10 random handwritten digits classified using the Averaged Kernel Perceptron with the second degree polynomial Kernel, trained on 5 epochs and achieving 0.039 test error.

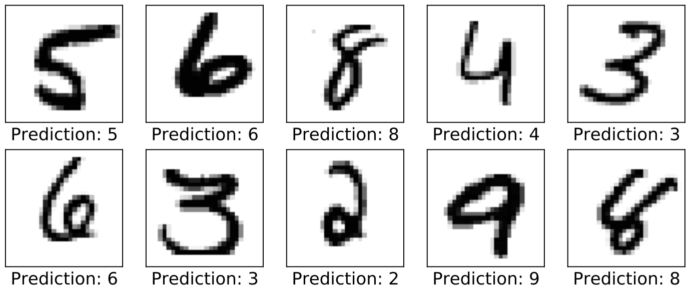

Figure 6: Random sample Multi-class classification

Link to the repository `https://github.com/dash-ka/ML_projects`.

Bibliography

Hal Daumé III, ”A course in Machine Learning” (2017), http://ciml.info/

Lecture notes of professor Cesa-Bianchi N.A.