MNIST evaluation with Kernel Perceptron

Matteo Ciarrocchi, e-mail: matteo.ciarrocchi2@studenti.unimi.it, student number: 973884

December 19, 2021

Abstract

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.

The analysis aims to develop a Kernel Perceptron learning algorithm for ten handwritten digits recognition, from 0 to 9, of the MNIST dataset. In the first place, kernel Perceptron classifiers have been trained for each digit, considering 5 epochs and a kernel expansion of degree 2 as well as collecting predictors at the end of each epoch in order to construct a starting framework. After that, three kind of classifiers have been computed to carry on the analysis: the empirical risk minimizer predictor, the average kernel perceptron and the weighted average one. Training and test accuracies have been computed, showing that the best performances for almost all the digits are given by the weighted average classifier. After that, "number of epochs" and "degree of expansion" hyperparameters have been tuned splitting the training set into training-validation subsets and also using cross-validation. The "degree of expansion" plays an important role since it sets the velocity of the learning process. Using the selected parameters, the models have been retrained and evaluated on the test set to show improvements. Furthermore, combining binary classifiers from different digits, sharing and not sharing parameters, into a single predictor, multiclassification analysis has been run. It has been demonstrated that binary classifiers individually tuned and then combined for the multiclassification have worse accuracies than the binary classifiers all tuned with the same parameters.

1 Why Perceptron with kernel?

Perceptron algorithm is one of the most simple linear classifiers used in machine learning. It takes training data points one by one and adjusting itself at each error until to obtain a final hyperplane. It is usually affected by bias and it ends up with bad performances when the dataset is not linearly separable. This happens because linear predictors are developed in a space as big as the number of features and the "Bayes optimal predictor" is not usually included in the set of the computable classifiers. In order to amend the problem, a popular technique is the feature expansion with which new features are computed through the linear combination of starting attributes. Since the size of the expansion is exponentially related to the number of the starting features, the *kernel-trick* is used to overcome the problem and to obtain the same results in a quicker way. Types of kernels are different, but in this paper only polynomial kernel expansion of different degrees is considered. The final accuracies are improved because data points with different labels become linearly separable in an higher dimensional space, fostering the work of the linear classifier as illustrated in the next sections.

2 Dataset presentation and an overview on the methodology

The analysis takes place from the MNIST dataset, considering 8000 observations of the training set and 2000 of the test set. It aims to develop an efficient algorithm for recognizing handwritten digits from 0 to 9. Using the *one-vs-all method* to encode the labels, ten different datasets, but differing only for the labels, have been created. Training and test datasets contain 785 columns, considering

784 of them for features and one for the labels encoded in -1 and 1 categories. Since the analysis is focused on images recognition, the features are codified as pixels and they share the same unit of measurement. This means that data can be encoded as vectors of real numbers and geometric relationships, as the Euclidean distance, among data points can reflect similar information about the labels' relationships.
Kernel Perceptron works pretty well with these datasets because they are *linearly separable* in an expanded space. In fact, there are several predictors for each digit which have training error equal to 0 and they are obtained simply by considering values of the hyperparamters sufficiently high. Since there is at least one predictor with zero training error for each digit, this is enough to state that the datasets are linearly separable. This is also the hypothesis of the "Perceptron Convergence Theorem", which states that the algorithm terminates or, in other words, that the zero training error is reached.
In order to extract a binary classifier for a whichever digit, first the null vector *alpha* has been considered. Then, the latter one has been trained taking points of the training set one by one, updating the components only when an error in the classification of the corresponding point occurred.
In the first part of the analysis, the process has been carried on for 5 epochs and always considering a kernel polynomial expansion of order 2, collecting the classifiers at the end of each corresponding epoch. Using them, different training and test accuracies have been computed with three different methodologies:

- Using the predictor of the epoch with the lowest training error.
- Computing a new predictor averaging the five ones collected during the training.
- Computing a new predictor through a weighted average of the five previous predictors, where the weight of each one has been determined taking into consideration the number of errors committed during the learning in the corresponding epoch.

In the second part of the analysis, the training set previously considered was split into training-validation sets for each digit in order to tune the hyperparameters *number of epochs* and *degree of the expansion*. For the first one, values going from 5 up to 15 with step 3 have been considered and for the second one, numbers in a range within 2 and 7. The parameters have been simultaneously evaluated, therefore each possible combination of these values has been reported. After having selected the best combination of parameters, which is the one corresponding to the smallest error in the validation set, the model has been trained considering it and the whole starting training set. The tuning process has been repeated also by using a k-fold cross-validation approach and the results have been compared.
Another possible technique is the *nested cross-validation*. As in the k-fold cross-validation, its main feature is to be dependent only on the size of training set and not on the peculiarity of the present data as in the validation set approach. Even if it's accurate, this method has not been used in this analysis because computationally too expensive to implement from scratch.
Next, for each digit, the multiclassification has been run by using the predictions of binary classifiers, before with hyperparameters tuned individually on each binary predictor and then with parameters shared across the digits.
The report is developed in this way: section 3 introduces a first approach to the kernel Perceptron algorithm, section 4 is related to the "tuning hyperparameters" processes, section 5 contains the multiclassification analysis and section 6 contains conclusions and it suggests possible insights.

3 First exercise: computing training errors and related predictors

Methodologies and code for the algorithm, accompanied by comments and explanations, are introduced in the first part of this section, followed by the results obtained on the MNIST dataset.

3.1 Methodologies

This section reports the kernel-Perceptron algorithm written from scratch used in the first exercise of this analysis and the auxiliaries functions. The second part of the analysis relies on the same structure, but with some add-ons to implement the tuning processes and the multiclassification.

```
def kern(x1,x2,d=2,c=1):
    pre = c + np.dot(x1.T,x2)
    ret = pre**d
    return ret
```

Figure 1: Kernel definition

```
def mapps_2(array,d):
    map = np.array([[None]*len(array)]*len(array))
    for row_index in range(len(array)):
        for k in range(row_index,len(array)):
                value = kern(array[k][:-1],array[row_index][:-1],d)
                map[row_index][k] = value
                map[k][row_index] = value
    return np.matrix(map)
```

Figure 2: Kernel evaluations

```
def train_kern_alpha(train, n_epoch, kernels): #train --> dataset to train
    alpha_list = []
    alpha = np.array([0.0 for i in range(len(train))])
    n_errors = []
    for epoch in range(n_epoch):
        sum_error = 0.0
        for i in range(len(train)):
            prediction = predict_2(train, alpha, i, kernels)
            error = train[i][-1] - prediction
            sum_error += error**2
            alpha[i] += error
        print('>epoch=%d, error=%.3f' % (epoch, sum_error))
        c = list(alpha).copy()
        n_errors.append(sum_error)
        alpha_list.append(c)
    print('average error = %2f' % (sum(n_errors)/len(n_errors)))
    return (alpha,alpha_list,n_errors)
```

Figure 3: Training process of the predictor

```
def kernel_perceptron(train, test, n_epoch, kernels,d):
    predictions_tr = list()
    predictions_test = list()
    alpha,alpha_list,errors = train_kern_alpha(train, n_epoch, kernels)
    print(alpha)
    for row in range(len(train)):
        prediction = predict_2(train,alpha,row,kernels)
        predictions_tr.append(prediction)
    for row in range(len(test)):
        prediction = predict_test(train,test,alpha,row,d)
        predictions_test.append(prediction)
    return(predictions_tr,predictions_test,alpha_list,errors)
```

Figure 4: Kernel Perceptron algorithm

```
def predict_test(train,test,alpha, row_index,d):
    activation = 0
    for k in range(len(train)):
        activation += alpha[k]*kern(train[k][:-1],test[row_index][:-1],d)
    return 1.0 if activation >= 0.0 else 0.0

def predict_2(array,alpha, row_index, kernels):
    activation = np.dot(kernels[row_index],alpha)
    return 1.0 if activation >= 0.0 else 0.0
```

Figure 5: Prediction's methods

```
def accuracy_metric(actual, predicted):
        correct = 0
        for i in range(len(actual)):
                if actual[i] == predicted[i]:
                        correct += 1
        return correct / float(len(actual)) * 100.0
```

Figure 6: Accuracy's definition

The definition of how the polynomial kernel evaluation is computed is reported in figure 1, where d is the value of the degree at which the kernel of two data points is computed. The kernels perform the same task of a feature expansion, but in a faster way and with the same outcome.
The function *mapps_2* of figure 2 is useful for the evaluation of those points of the training set. It has been used to collect the kernel evaluations of the training data point with themselves. It takes as input an array, like the training set, and the degree of the polynomial kernel. Therefore, the outcome is a symmetric squared matrix with side equal to length of the array and whose evaluations are used to compute the training accuracy. This map is useful for a computational reason in terms of time: being a symmetric matrix, it is possible to save half of the evaluations because symmetrical evaluations with respect to the diagonal are equal by definition. Therefore, kernels are computed all before that the algorithm starts and then each row is considered and processed for the classification of the point. *Mapps_2* is computed only one time for all the digits since only the labels of the datasets are different. The map is used only for the evaluation of the training data points.
The function of figure 3 trains the algorithm, starting from the null linear predictor composed by all zeros for a length equal to the size of the training set. This predictor is updated for a fixed number of epochs given as input, taking each point one by one and predicting its label using *predict_2*, which is based on the evaluation of *mapps_2*. After each evaluation, an error is computed and the *alpha* predictor is updated, considering in the update the label of the point that is misclassified. The predictors computed at the end of each epoch are reported. Additional information is also present, such as the number of errors committed in the epoch and the average error across the epochs.
Figure 4 reports the "root" of the used algorithm. It is the first function used in the analysis and its first call is to the training process mentioned above (figure 3). Then, it computes the training and test errors of the final predictor exploiting the functions of figure 5. The "kernels" input is computed using the function of figure 2.
Figure 5 depicts the activation functions. They work in a similar way, but the first one (*predict_2*) facilitates the learning process and the training error evaluations. Here, the kernel evaluations are computed in a previous step (*mapps_2*) and then the prediction is made through an inner product between the *alpha* passed by the training process and the row of the map which is currently evaluated. The function *predict_test* is used for the evaluation of the points of the test set and it cannot rely on the map because this map would not be symmetric, therefore time cannot be saved in the kernel evaluations. In this function, the classic implementation of the prediction in a Kernel Perceptron algorithm is shown.
After having computed the predictions, they are evaluated comparing them with the real labels of the training and test points, as described in figure 6.

```
training accuracy of kernel Perceptron for digit 0 = 99.80, average training error = 52.40, min training error with 04 epochs
training accuracy of kernel Perceptron for digit 1 = 99.89, average training error = 59.00, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 2 = 99.80, average training error = 92.80, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 3 = 99.39, average training error = 120.40, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 4 = 99.91, average training error = 81.00, min training error with 04 epochs
training accuracy of kernel Perceptron for digit 5 = 99.88, average training error = 98.40, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 6 = 99.66, average training error = 66.00, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 7 = 99.61, average training error = 88.20, min training error with 04 epochs
training accuracy of kernel Perceptron for digit 8 = 98.89, average training error = 154.00, min training error with 05 epochs
training accuracy of kernel Perceptron for digit 9 = 99.59, average training error = 169.40, min training error with 05 epochs
```

Figure 7: Training accuracies' sum up: 1st part

```
average training accuracy for digit 0 = 99.90, weighted average training accuracy = 99.94
average training accuracy for digit 1 = 99.69, weighted average training accuracy = 99.81
average training accuracy for digit 2 = 99.86, weighted average training accuracy = 99.88
average training accuracy for digit 3 = 99.49, weighted average training accuracy = 99.50
average training accuracy for digit 4 = 99.88, weighted average training accuracy = 99.95
average training accuracy for digit 5 = 99.83, weighted average training accuracy = 99.86
average training accuracy for digit 6 = 99.91, weighted average training accuracy = 99.90
average training accuracy for digit 7 = 99.72, weighted average training accuracy = 99.75
average training accuracy for digit 8 = 99.52, weighted average training accuracy = 99.71
average training accuracy for digit 9 = 99.60, weighted average training accuracy = 99.75
```

Figure 8: Training accuracies' sum up: 2nd part

3.2 Outcomes: Minimum training error, average and weighted average predictors

This section introduces the results of the first part of the analysis, considering three different types of predictors: the *smallest training error* or *Empirical Risk Minimizer*, the *average* predictor and the *weighted average* one. All of them have been run considering 5 epochs and a polynomial degree of order 2 and collecting predictors at the end of the epoch for each digit. As reported in figure 7, only digit number 0, 4 and 7 do not make the minimum number of mistakes in the last epoch, therefore they are the only three ones whose predictor should be computed with 4 epochs as parameter. Training accuracies reported in figure 7 refer always to a predictor trained with 5 epochs, as the outcomes of figure 8 refer to an average or weighted average with respect to 5 collected predictors. It is noteworthy that weighted average training accuracies are usually better than the average ones.

These predictors are evaluated on the test set and the results are reported in table 9 and graphically in figure 10. It is clear how the *minimum training error* predictor is the worst one, because it is always outperformed except for digits 0 and 1. This could be due to the fact that with the selected parameters *number of epochs* equal to 5 and *polynomial degree* equal to 2 the empirical risk minimizer predictors do not have zero training error because higher values of parameters are needed. Moreover, the problem could be related to the order which training data feed the algorithm

	"min training error" test accuracy	average test accuracy	weighted average test accuracy
0	99.70	99.55	99.40
1	99.50	99.40	99.40
2	98.30	98.60	98.65
3	98.35	98.75	98.75
4	98.45	98.55	98.65
5	98.90	99.00	98.85
6	98.40	98.65	98.65
7	98.00	98.20	98.20
8	97.10	97.65	97.75
9	97.45	97.95	97.95

Figure 9: Test results

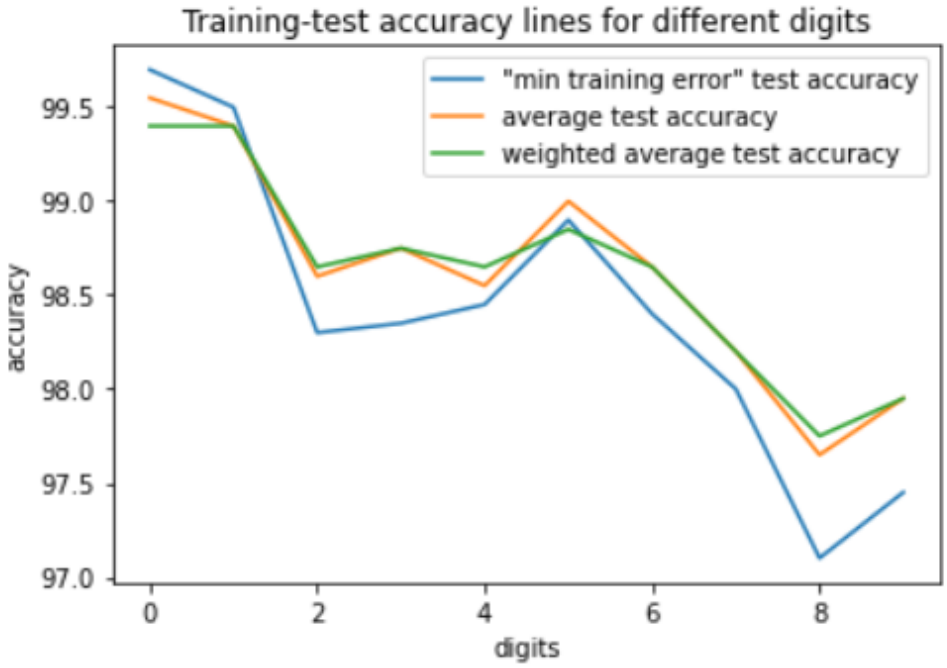

Figure 10: Test lines

with.
Average predictors seem to work better, even if the average is made regardless the accuracy of the classifier of each epoch.
A further improvement has been made introducing a criterion with which the average is carried out: using the number of errors committed in each epoch and assigning proportional weights to the 5 predictors. Bigger weights are here assigned to predictors with lower errors. However, the results on training and test sets are generally really close to the average predictors for all the digits. Theoretically, this last kind of predictor should generalize (before each predictor had the same weight) the previous one and reduce the error estimates, giving less relevance to those predictors with bigger numbers of errors. This might not be happened because hyperparamters should be tuned first. Here it has been shown one way to weight this effect, but there are also other solutions to amend the problem: collecting predictors for each update or using the so-called *voting* system, which takes into consideration how long each predictor is not updated in the learning using counters and then exploiting those counters as weights to compute the final weighted predictor.
Summarising, in this first part there is a slight overfitting: training and test performances are somehow distant between themselves in all the three considered predictors and there is room for improvement. This is because hyperparamters have been taken randomly and not properly set: the algorithm "Kernel Perceptron" is able to output predictors with lower test errors and this is matter of the "tuning hyperparameters" process.

4 Second exercise: Tuning hyperparameters and multi-classification performances

Random parameters have been given in the first part of the analysis and predictors have been trained on these ones. This section aims to tune these hyperparameters, working on the training set of the previous section both with a *validation set approach* and a *k-fold cross-validation approach*.

4.1 Selecting the best hyperparameters

Hyperparameters selection has been performed for each digit considering variations in the parameters *degree of polynomial* and *number of epochs*. The selection takes place in a grid from 5 to 15 with step 3 for the first parameter, while in a range between 2 and 7 for the second one. This process has been done using two different approaches in order to compare their performances:

- A validation set approach.

- A 5-fold cross-validation approach.

4.1.1 Validation set approach

The training set has been splitted into a new smaller training set with 6400 observations and a validation one with 1600 observations, all the possible combinations of hyperparametrs have been trained in a model and then evaluated on the validation set, whose accuracy drives the choice of the best couple of paramaters. In case of a tie, the first used criterion has been the minimum number of epochs and then the minimum degree of the polynomial. This process has been repeated for each combination and the couple with the best validation accuracy has been picked up as the best one and used to train the model again on the whole training dataset. After that, this retrained model has been evaluated on the test set, guaranteeing better performances respect to the previous computation where parameters have been chosen randomly.
What it turned out to be relevant in the tuning hyperparameters process is the role of the variable *polynomial of degree*. This parameter controls the speed with which the stochastic gradient descent process learns from training data because the learning is always faster as the value of the degree increases. Training data are linearly separable, therefore a slow learning process issued by small degrees could not complete its learning process and reach zero training error if a small number of epochs is considered. On the other hand, high degrees of the polynomial could result in a too fast learning which could end up with an hyperplane which separate well the training data, but that is not properly at an half-way between the two different categories of data points and resulting in bad performances on the test set. It is important to select the right number of degrees in order to end up with an hyperplane as efficient as possible in order to find the best possible margin.
This is what is shown in the 0-digit figure 11. Polynomial degree of order 2 is too low to separate completely 6400 training data points with the considered number of epochs until there are not 11 epochs. Polynomial degree 3 becomes a flat curve after 8 epochs, which means training data are completely separate by the hyperplane because no update has been made after that value. All the others polynomial degree levels show flat curves, therefore they are able to classify training data points correctly with less than 5 epochs.
Similar learning processes have been performed for all the other digits, where higher polynomial degrees have ended up with straight lines for all the numbers of epochs selected in the analysis.
The best couple of parameter has been identified considering the best accuracy in the validation set. For example, referring to figure 11, *polynomial degree 5* and *number of epochs 5* parameters have been chosen, since there is a tie among the different number of epochs in correspondence of *polynomial degree* 5. In this case, adding number of epochs does not change the predictor and the validation set accuracy is not affected, simply because the zero training error has been already reached with 5 epochs when the polynomial degree is of order 5. In other words, no further updates are performed by the algorithm.
In a nutshell, as explained above, the *polynomial degree* controls the gradient descent learning of Kernel Perceptron in order to find the "golden zone" in which the predictor can reach its best validation accuracy, without learning too slow or to fast, looking for a hyperplane as efficient as possible.

4.1.2 5-fold cross-validation approach

Validation set approach could suffer by overfitting because of the specificity of the validation set, this is the reason why another selection process has been implemented: the cross-validation over the training set. Considering the original training set composed by 8000 observations, a CV with 5 blocks has been performed using the grid described above. The aim of the cross-validation is to find the expected value of the statistical risk for the selected algorithm and with respect the random draw of the training set with size 8000 observations. Technically, this is done evaluating the learning algorithm multiple times changing training and evaluation sets and averaging performances at the end, softening in this way the potential overfitting of the validation set approach. At the end of the process, the couple of parameters associated with the best accuracy is picked up for each digit. The drawback of this approach is the underestimate of the expected value of the statistical risk, but the difference is typically small respect to the more appropriate approach *nested cross-validation*.

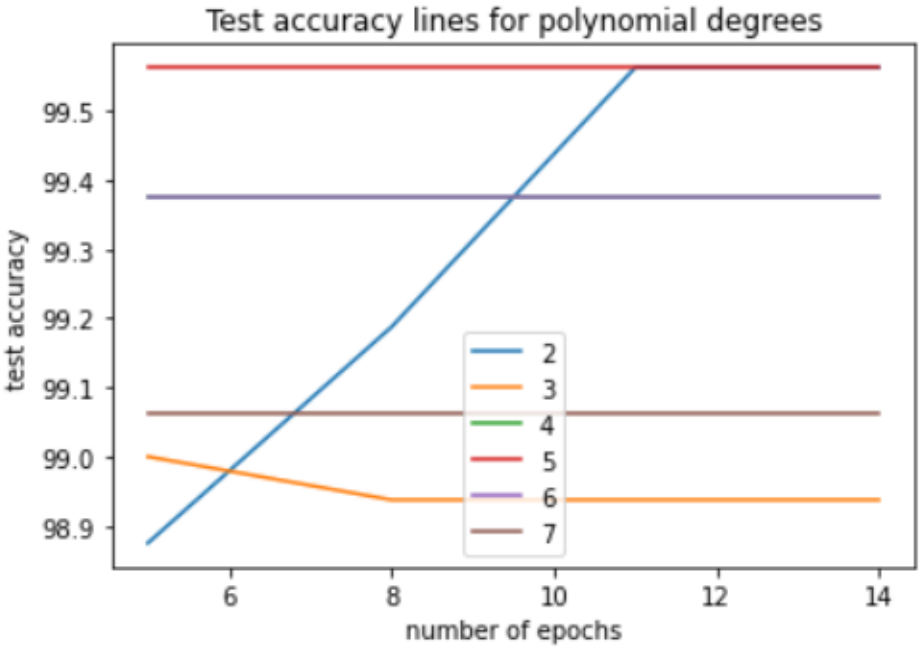

Figure 11: Test results depending on polynomial degrees

```
polynomial degree =5 and epochs =5 guarantee 99.56 accuracy in the validation set for digit 0
polynomial degree =4 and epochs =5 guarantee 99.56 accuracy in the validation set for digit 1
polynomial degree =5 and epochs =5 guarantee 99.00 accuracy in the validation set for digit 2
polynomial degree =5 and epochs =8 guarantee 98.44 accuracy in the validation set for digit 3
polynomial degree =5 and epochs =5 guarantee 99.00 accuracy in the validation set for digit 4
polynomial degree =6 and epochs =5 guarantee 98.88 accuracy in the validation set for digit 5
polynomial degree =3 and epochs =5 guarantee 99.69 accuracy in the validation set for digit 6
polynomial degree =3 and epochs =5 guarantee 99.19 accuracy in the validation set for digit 7
polynomial degree =4 and epochs =5 guarantee 98.19 accuracy in the validation set for digit 8
polynomial degree =4 and epochs =11 guarantee 98.81 accuracy in the validation set for digit 9
```

Figure 12: Tuned hyperparamters with validation set approach

```
polynomial degree =2 and epochs =14 guarantee 99.53 accuracy in cross-validation for digit 0
polynomial degree =4 and epochs =8 guarantee 99.51 accuracy in cross-validation for digit 1
polynomial degree =4 and epochs =8 guarantee 98.92 accuracy in cross-validation for digit 2
polynomial degree =3 and epochs =11 guarantee 98.79 accuracy in cross-validation for digit 3
polynomial degree =2 and epochs =8 guarantee 99.10 accuracy in cross-validation for digit 4
polynomial degree =2 and epochs =14 guarantee 99.06 accuracy in cross-validation for digit 5
polynomial degree =3 and epochs =8 guarantee 99.41 accuracy in cross-validation for digit 6
polynomial degree =3 and epochs =8 guarantee 99.05 accuracy in cross-validation for digit 7
polynomial degree =5 and epochs =11 guarantee 98.56 accuracy in cross-validation for digit 8
polynomial degree =5 and epochs =8 guarantee 98.56 accuracy in cross-validation for digit 9
```

Figure 13: Tuned hyperparamters with cross-validation approach

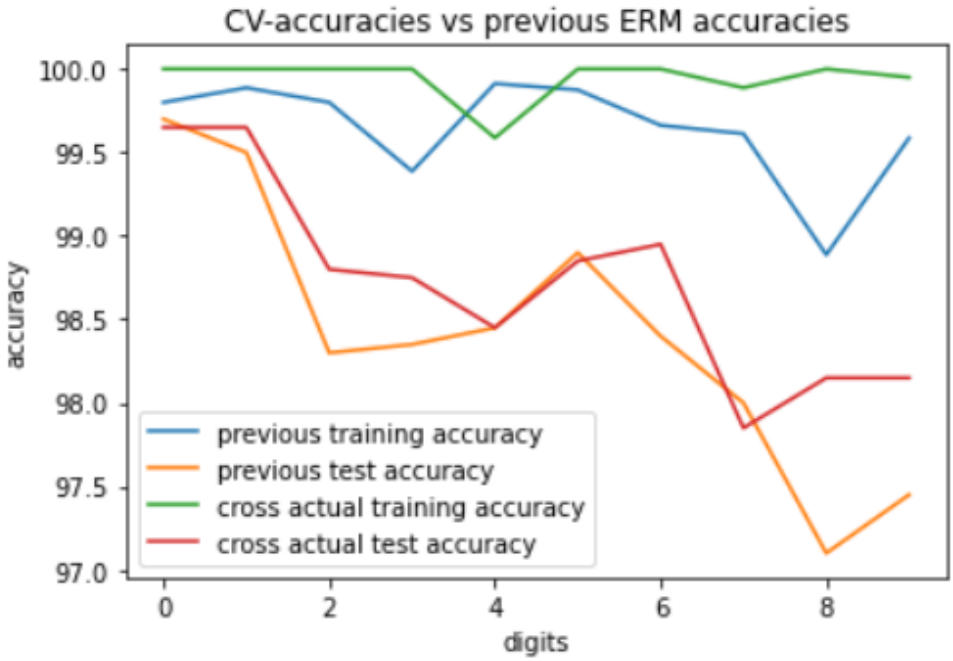

Figure 14: CV-tuned hyperparamters vs previous ERM predictors

4.1.3 Outcomes of the two processes

The results are reported in figure 12 and 13: all the performances are really close, even if the second set of results are more reliable because computed with respect the random draw of the training set, not suffering by peculiarity of data.

4.1.4 Re-training the models

The best couple of parameter for each digit has been found and, exploiting it, ten models have been trained using the ten original training datasets. Training accuracy and weighted average training accuracy of the models have been computed, considering values from both the tuning processes. After that, the predictors have been tested on the ten test sets considered in the first part of the analysis.

Several comparisons with training and test accuracies of the predictors *minimum training error* and *weighted average* of the first part of the analysis, kept as benchmark to check the improvements in the accuracy, have been computed. For example, in figure 14 there is the graphical representation of the re-trained models with hyperparameters tuned with CV against the empirical risk minimizers of the first part of the analysis, which had random starting parameters. There is a widespread improvement in training and test performances, as lines show.
Another interest comparison is reported in figure 15, where lines about the "weighted" predictors are shown. For each digit, the computed predictor has basically zero training error, the test performances are improved respect the first part of the analysis and, on average, predictors with hyperparameters tuned by CV outperformed the ones tuned by the validation set approach.

4.2 Multiclassification models

The last part of the analysis is focused on the multiclassification of the MNIST dataset, whose labels take values between 0 and 9.
Kernel Perceptron is, for its nature, a binary classifier, therefore it is able to classify data points with labels which belong to a class of 2 elements. Since the original dataset is composed with labels included in an finite interval going from 0 to 9, a "One vs All" encoding technique has been previously implemented in order to use the kernel perceptron algorithm. This created ten different datasets based on the principle "belonging to the class vs not-belonging to the class".
In order to carry out a multiclassification, ten binary classifiers have been simultaneosly considered and the classification has been performed following this criterion: each data point is classified according to the label of the binary classifier's prediction which takes the greatest value. Typically,

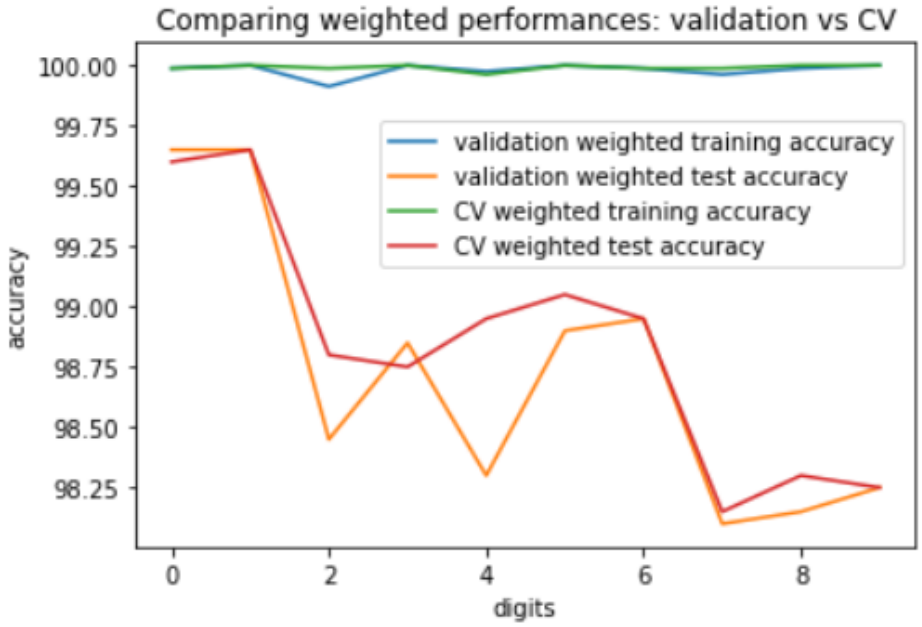

Figure 15: Weighted accuracies: CV vs validation

for each data point only one classifier's prediction takes positive value and therefore targets the point as "belonging to the class". Nevertheless, there are data points which are at an half-way in the vector space and whose prediction is shared by more than one binary classifier. This can also happen simply because one binary classifier misclassifies the data point. In this case, the greatest value identifies the most reliable prediction over lower values.
In the previous section, four different sets of "ten binary classifiers" have been copmuted following four different processes:

- Hyperparamters tuned with a validation set approach for each digit. Using those values, the algorithm has been run and the predictors have been computed.
- Hyperparamters tuned with a validation set approach for each digit. Using them, the learning algorithm has been retrained, and the final predictors have been obtained averaging with weights the predictors collected at the end of each epoch.
- Hyperparamters tuned with a cross-validation approach for each digit. Using those values, the algorithm has been run and the predictors have been computed.
- Hyperparamters tuned with a cross-validation approach for each digit. Using them, the learning algorithm has been retrained, and the final predictors have been obtained averaging with weights the predictors collected at the end of each epoch.

Four multiclassifications have been performed according to these four different sets of binary predictors. The peculiarity of these predictions is that they have been made using binary classifiers having different parameters, because each of them has been previously and individually tuned, therefore maximizing performances with different values of *number of epochs* and *polynomial degree*. Respecting the order of the previous list, the accuracies respectively reached by the multiclassifers have been 92.4, 92.95, 91.95 and 92.4. All these four results are really similar, even if the weighted accuracies seem to perform slightly worse.
In order to evaluate the accuracy of the previous performances, other multiclassifications have been implemented, but this time using ten binary classifiers sharing the values of parameters. Technically, the algorithm has been run for each digit and for each possible couple of parameters taking the usual grid. Predictions of each evaluated point have been compared across the outputs of the ten classifiers as the values of parameters changed.
The results are shown in figure 16. Accuracies are pretty high and it is partly due to the fact that the MNIST dataset is balanced, since each class has more or less the same number of labels. The best accuracy has been obtained when *polynomial degree* is equal to 3 and it is 95.25. This value has been reached considering 8 epochs and it remains constant increasing the value of the epoch parameter when *pol_degree* is equal to 3. This pattern holds for all the polynomial degrees and

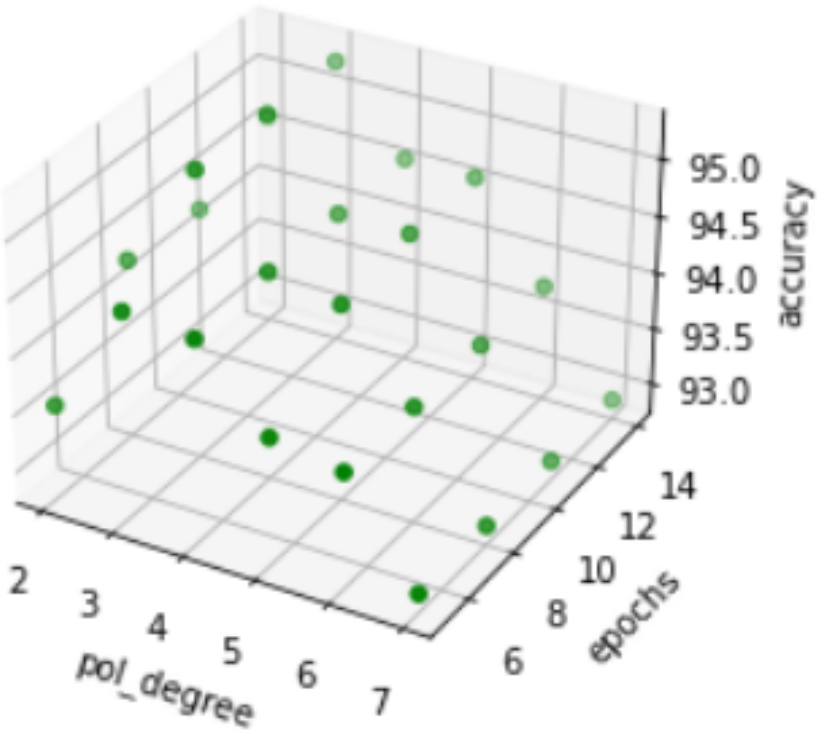

Figure 16: Multiclass predictions with shared paramters

becomes sharper with values 6 and 7. On the other hand, value 2 behaves differently, because, in this case, binary predictors are not able to reach the zero training error for each considered number of epochs and according to the size of the given dataset.
These accuracy values are higher than the ones found before. Thus, multiclassifiers computed with binary predictors which share parameters perform better than multiclassifiers computed with binary predictors individually tuned. This holds for all the possible combination of *number of epochs* and *polynomial degree*.
Summarising, shared parameters across the binary classifiers are able to generalize and perform better in this multiclassification problem.

5 Conclusions and possible insights

Several variants of the same algorithm have been developed, using ERM predictors, averages and weighted averages ones. ERM classifiers have the lowest training error by definition, but it has been computed how they are outperformed considering the test accuracy.
The tuning process has depicted the key role of the *polyonomial degree* parameter in the speed of the learning process, while the *number of epochs* has been selected adapting the outcomes to a proper definition of Kernel Perceptron algorithm in case of equal performances.
The multiclassification has reported an interesting feature: performances obtained with classifiers sharing the same parameters are really great, with a peak of 95.25 accuracy and they outperform by around 2 percentage points accuracies obtained with classifiers whose hyperparamters have been individually tuned.
Clearly, there is room for improvement. It's possible to evaluate even better the performance of the Kernel Perceptron algorithm on MNIST dataset.

- First, the improvement could come from an higher number of observations, in order to feed the model with more instances.
- Second, predictors for each evaluated point could be computed instead of at the end of the epoch, in order to find out a wider range of classifiers and a more accurate best predictor.
- Third, comparisons with the baseline Perceptron algorithm.

- Fourth, it could be performed cross-validation or, even better, a nested cross-validation analysis, to check the accuracy of the multiclassification. This would be a more reliable estimate than the one found in this paper, since the performance would not rely on the specificity of used training and test sets, but instead it would be an average of outcomes coming from different splits of the starting set.

In the end, the Kernel Perceptron algorithm is able to classify really well data points of this dataset, their linear separability fits perfectly the features of this algorithm and this guarantees accuracies moving in a range between 98.20 and 99.80 percentage points for binary classifiers and between 93.00 and 95.25 for the multiclass predictors.

6 Code

Link for the code: MATTEO.CIARROCCHI_MLPROJECT