Machine learning Assignment1

Chiara Saporetti

October 2020

Abstract The purpose of this assignment is to implement a Naive Bayes Classifier capable of working on data sets of different dimensions and internal characteristics. It also tries to solve some inaccuracies of the basic implementation by adding a Laplacian smoothing to the probabilistic model. Results are tested on a given data set.

1 Introduction

The Naive Bayes classifier is a probabilistic classifier based on the Bayes equation and on the the naive assumption that all input variables are independent from each other. The a posteriori probability will then be:

$$P(c|X) \propto \prod P(xi|c)P(c) \tag{1}$$

The code to implement the classifier is written in the MATLAB environment [2] and divided in functions. It is tested on a given "weather" data set. The latter is composed of 14 examples, 4 features, 2 possible classes and at maximum 3 different outcomes for each feature.

2 Task 1: Preprocessing the data set

The first step of the work was to preprocess the given data set by eliminating the names of the features and coding each possible instance of the features with a numerical value. This was done on Microsoft Word through the substitute command.

3 Task 2: Coding

The second step was to prepare code to implement the classifier. The code is structured in a main function that calls multiple functions to compute important data and that subsequently prints the results.

3.1 Main.m

The main loads the data set and saves information about the number of examples/observations (rows of the data set), the number of features (columns of the data set), the number of levels of each feature (possible values of the observations) and the number of classes (possible outcomes of the observations). It also controls that no value in the data set is below 1. It then shuffles the data set rows and randomly separates them into a training subset and a test subset, by previously asking the user how many lines to use for the training part.
It subsequently calls two functions that respectively compute the model of our training set, i.e. the $P(X|c)$ and $P(c)$, and the smoothed model of our training set, that reaches the same goal by although optimizing the logic.
After defining the model of our set, the main calls a function that tests the probabilities computed by the previous one by considering the test set. If the results of the test set are available, the main calls a function that evaluates the accuracy of the predictions. Otherwise, the main just shows the predictions and the test set. Hereafter are presented the functions.

3.2 NaiveModel.m

Input:

- my_set: data set of attributes
- my_res: results of data set
- CLASSES: number of possible results
- LEVELS: levels of each attribute

Output:

- p_class: $P(c)$
- p_feature_class: $P(X|c)$

This function counts the number of the different class instances in the result column of the data set (N_c) and the number of instances of each feature level for class (N_fc). Then the function computes the ratio between N_c over the total number of examples and the the ratio between N_fc over the total number of examples of that class. In formulas:

$$p\_class(c) = \frac{N\_c(c)}{N\_examples} \tag{2}$$

$$p\_feature\_class(f, l, c) = \frac{N\_fc(f, l, c)}{N\_c(c)} \tag{3}$$

where p_feat_class is the likelihood matrix (with dimensions features x levels x classes).
Finally the function checks if the likelihoods of all levels of a feature sum up to 1. The problem with this function is that, if a specific level of a feature doesn't appear in the training set, its likelihood appears to be zero. In order to solve this inaccuracy the formula is slightly modified in Task 3 (see chapter 4.1)

3.3 NaiveClassifier.m

Input:

- test_set: data set of attributes
- p_c: P(c)
- p_feature_class: $P(X|c)$
- N_TEST: number of examples of the test set
- CLASSES: number of possible results

Output:

- p_class_feature: $P(c|X)$

This function implements the Naive Bayes classifier. [1] It computes the $P(c|X)$ for every line of the test, by applying the Bayes equation:

$$P(c|X) = \frac{\prod P(xi|c)P(c)}{P(X)} \tag{4}$$

However, the denominator of the Bayes equation is omitted in the code. The reason is that, since it is the same for all classes and since we only need to compare the values of P for each class, we can just compare the numerators of the fraction. The resulting equation is:

$$P(c|X) \propto \prod P(xi|c)P(c) \tag{5}$$

The problem of applying this kind of equation is that, if probabilities are very low, i.e. tend to zero, the total product goes quickly to zero and may not be right. An attempt to solve this problem may be found at chapter 5.1.

3.4 EvaluateAccuracy.m

Input:

- prob_c: $P(c|X)$
- test_res: results of the test set

Output:

- accuracy: how many guesses were right

This function evaluates the percentage of correct predictions in the results with reference to the real values, and can thus be used only when the results are available. In order to do so, for each example the function takes the the index of the maximum value of $P(c)$ and compares it to the real result. Then it computes the ratio of correct guesses over the total number of tests and gives the result in percentage.

4 Task 3

As previously anticipated, this task optimizes the function described in chapter 3.2. The result is hereby described.

4.1 NaiveModelSmooth.m

Input:

- my_set: data set of attributes
- my_res: results of data set
- CLASSES: number of possible results
- LEVELS: levels of each attributes

Output:

- p_class: P(c)
- p_feature_class: $P(X|c)$

This function is very similar to NaiveModel.m but computes the $P(c|X)$ by applying Laplace smoothing:

$$p\_feat\_class(f, l, c) = \frac{N\_fc(f, l, c) + a}{N\_c(c) + av} \tag{6}$$

This equation tries to solve the problem of some values of a feature not appearing in the data set. It takes into account the idea that all values in a feature are equally probable. So it adds a term a=1 to the numerator, and a x v to the denominator, where v is the number of levels of that feature.

5 Probability with logarithm

As previously anticipated, this function tries to solve the problem highlighted in chapter 3.3. The result is hereby described.

5.1 NaiveClassifierLog.m

Input:

- test_set: data set of attributes
- p_c: P(c)
- p_feature_class: $P(X|c)$

- N_TEST: number of examples of the test set

Output:

- p_class_feature: $P(c|X)$

This function is very similar to NaiveClassifier.m but uses a sum of logarithms of the probabilities instead of a product of the probabilities:

$$P(c|X) = \sum log(P(X|c))log(P(c)) \tag{7}$$

The logarithm keeps the relations between the probabilities while simplifying the calculus.

6 Results

The accuracy of the classifier depends on the data lines that were randomly chosen at the beginning. Since the data set is very small, in order to obtain more meaningful results, the code has been run 1000 times and the mean of the accuracy was calculated for each loop. The final results are:
mean = 59.4500 % and mean_smoothed = 58.9250 %

Tests were made by setting all "1" values of the first feature to "2", in order to test the smoothing effect. The final results are:
mean = 55.5500 % and mean_smoothed = 60.5250 %
The smoothing effect is more evident from this second test, since the mean value is increased of about 8%. In general, however, a bigger data set would be needed in order to obtain better results.

References

[1] *https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c.*

[2] *Mathwork documentation.*

Machine learning Assignment2

November 2020

Abstract This assignment aims to implement in MATLAB three different linear regression models and to also calculate their mean square error objective. Results are tested on two different datasets and on different percentages of the data.

1 Introduction

1.1 Linear regression problem

Linear regression is a machine learning tool whose purpose is to model the relationship between a dependent variable t (target) and a set of one or more dependent variables X (data). To do so, it looks for a linear model y(x) that predicts t given X:

$$y = wX \tag{1}$$

Where the w vector represents the weights that must be calculated. In matrix form:

$$\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix} \simeq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} x_1 1 & ... & x_1 d \\ x_2 1 & ... & x_2 d \\ \vdots & \ddots & \vdots \\ x_N 1 & ... & x_N d \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \tag{2}$$

Since the problem above would have no closed form solution, we look for a solution based on optimization: we try to minimize an objective function J which is the expectation of a chosen loss function (the mean square error function).
So we calculate the mean square error (loss function):

$$\lambda_S E = (t - y)^2 \tag{3}$$

We then calculate the function that minimizes the mean value of the loss over the whole dataset (objective function):

$$J = \frac{1}{N} \sum_{i=1}^{N} (t_i - y_i)^2 \tag{4}$$

And we look for the minimum of this function, which is a quadratic function. [1]

1.2 The datasets

The datasets are:
- Motor Trend Car Road Tests set, made of 32 observations and 4 variables.
- Turkish stock exchange dataset, made of 536 observations and 2 variables.

2 Task 1 and 2

The main loads the two datasets through two different functions. For the turkish dataset it uses the MATLAB load() function, while for the car dataset it uses the MATLAB readtable() function, since it also contains string data together with numeric data.
Then it calls three functions to solve the linear regression problem with different models and plots the results.

Firstly it calls oneDimLinReg() to compute the slope for the linear model y of the first dataset.
Secondly it does the same but on different subsets created by randomly picking 10% of the original dataset.
Thirdly it calls oneDimLinReg_intercept() to compute the slope and intercept of the model of the second dataset by considering two of its variables only.
Lastly it normalizes the values of the Car data set, calls multiDimLinReg() on the entire second dataset and shows a table of the de-normalized results.

2.1 oneDimLinReg.m

Input:

- x: data
- t: target

Output:

- w: weights

This function computes the weights for the 1D linear model (i.e. the slope of y = w1 x). In formula:

$$w = \frac{\sum_{i=1}^{N}(x_i t_i)^2}{\sum_{i=1}^{N}(x_i)^2} \tag{5}$$

2.2 oneDimLinReg_intercept.m

Input:

- x: data
- t: target

Output:

- w1: slope
- w0: intercept

his function computes the weights for the 1D linear model (i.e. the slope and intercept of y = w1 x + w0). In formula:

$$w_1 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(t_i - \bar{t})}{\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{6}$$

$$w_0 = (t_i - w1\bar{x}) \tag{7}$$

2.3 multiDimLinReg.m

Input:

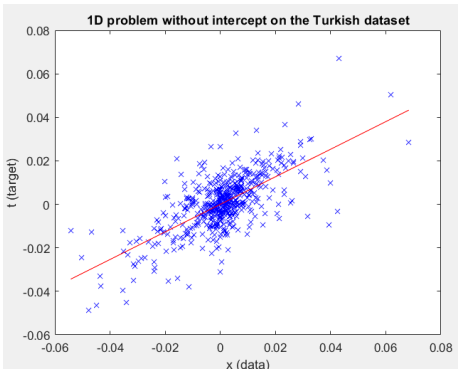
- X: data matrix
- t: target vector

Output:

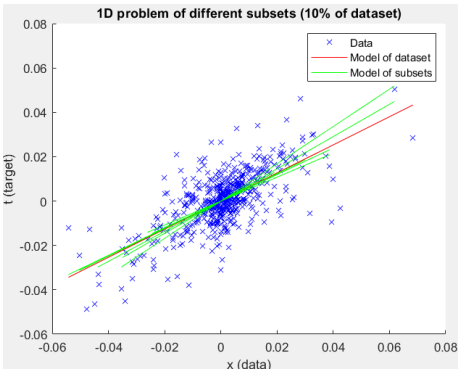
- W: weights vector

This function computes the weights vector for the multi dimensional linear regression problem. In formula:

$$w = (X^T X)^{-1} X^T t \tag{8}$$



(a) 1D problem without intercept



(b) 1D problem without intercept on different subsets

Figure 1: 1D problem

3 Task 3

The main divides the datasets in two parts: 5% is used to build a training set, while 95% is used to build a test set. In this particular case, the training set of the Turkish set is made of 27 lines, while the one of the car set is made of 2 lines only. The code then enters a loop and re-computes the 1-D problem and multi-D problems using the training sets. It subsequently tests the model on the test sets and computes the objective functions J for all cases. Finally it averages the results of all the J and shows them.

3.1 meanSquareError.m

Input:

- x: data
- t: target
- w1: slope
- w0: intercept
- problem: type of linear regression problem. can be 1 for 1-D, 2 for multi-D.

Output:

- objective: objective function

This function computes the mean square error objective of a linear regression problem. In particular, based on the *problem* variable it computes the error for either the one or the multi dimensional problem. The 1D problem's objective is calculated as:

$$J = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_i)^2 \tag{9}$$

While the multi dimensional problem's one is calculated through the MATLAB function immse().

4 Results

Hereby are shown the results that were obtained. Figure 1 (a) shows the solution to the linear regression problem applied to the entire Turkish dataset.

Figure 1 (b) shows the different results obtained by applying the same problem to subsets created by randomly picking 10% of the original dataset. The red line is the model computed on the entire dataset, while the green lines are the ones computed on the subsets. In the second case the results are less accurate then the first case, since the number of observations is lower (54 observations each). Figure 2 shows the solution to the linear regression problem applied to the car dataset considering

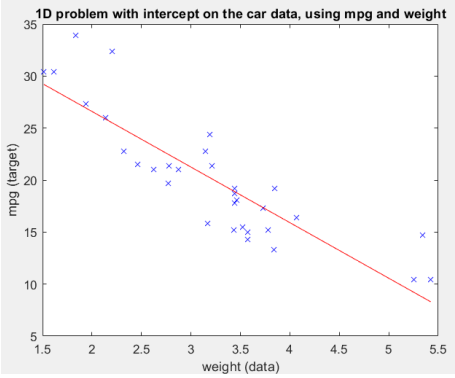


Figure 2: 1D problem with intercept

	Real Target t	Predicted Target y
1	21	17.7300
2	21	20.8063
3	22.8000	19.4427
4	21.4000	13.4556
5	18.7000	7.0986
6	18.1000	20.0486
7	14.3000	11.7344
8	24.4000	24.0573
9	22.8000	25.7103
10	19.2000	27.3040
11	17.8000	27.3040
12	16.4000	24.7577
13	17.3000	20.6560
14	15.2000	21.2592
15	10.4000	17.1604
16	10.4000	21.1001
17	14.7000	23.1416
18	32.4000	20.2359
19	30.4000	12.9155
20	33.9000	16.6769
21	21.5000	19.9532
22	15.5000	11.8763
23	15.2000	12.4870
24	13.3000	16.1603
25	19.2000	7.3100
26	27.3000	17.0039
27	26	15.7461
28	30.4000	12.0911
29	15.8000	8.7933
30	19.7000	24.1410
31	15	22.5733
32	21.4000	24.1740

(a) multi-D problem intercept

	Training	Test
1-D	2.2383e-04	8.7229e-05
1-D offset	2.0521e-04	1.4206e-04
multi-D	1.8972e-04	2.6306e-04

(b) Training-Test results

Figure 3

just the mpg as data and the weight as target.

Figure 3 (a) shows the t and y of the multi-D problem applied to the car set. Results are denormalized. Figure 3 (b) shows the results on the training set and on the test set, which have in general the same magnitude. The 1-D model is computed with more observations with respect to the other two, so it should have lower errors. Here, this only happens in the test set. The multi-D model has higher errors since the W is computed through the Moore-Penrose inverse which can bring to numerical errors. To solve this problem it would be useful to make an iterative computation by successive approximations. Finally, results on the training sets are mostly lower then results on the training set, which is reasonable since the model was computed on the training sets.

References

[1] *https://2020.aulaweb.unige.it/pluginfile.php/276344/mod*$_f$*older/content/0/ml* − 2020 − 21 − −03.*pdf*?f*orcedownload* = 1.

Machine learning Assignment3

November 2020

Abstract This assignment aims to implement in MATLAB a k-nearest neighbour classifier.

1 Introduction

1.1 kNN classifier

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. It assumes that similar things exist in close proximity. [1] Example:

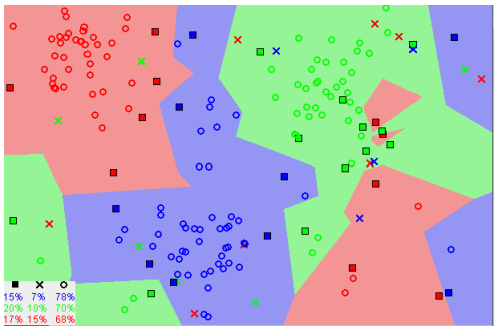


Figure 1: Similar data points typically exist close to each other

In order to implement this algorithm we need a training set X, a query point x and a value k. Then, given a set of n training examples, when the kNN classifier receives a new instance to predict, it firstly identifies the k nearest neighboring training examples of the new instance and then assigns the class label holding by the highest number of neighbors to the new instance. In formulas:

$$\{n_1, n_2, ...n_i, ...n_k\} = top_k||x_i - \bar{x}|| \tag{1}$$

$$y = mode\{t_n1, t_n2, ..., t_nk\} \tag{2}$$

1.2 The dataset

In order to try our code we used the MNIST dataset, which is composed of images of handwritten digits from 0 to 9. The training set has 600000 examples and the test set has 100000 examples.

2 Matlab code

2.1 main.m

The main first loads the MNIST training set and test set. Then it reduces the lines of both datasets by taking random samples, since the code takes a very long time to execute by taking the whole dataset. Then it applies the kNN classifier on the training set and tests it on the test set, by also showing some of the classified images. It averages accuracies and plots them with reference to the k. Finally it tests the kNN by considering each single digit and plots its accuracy.

2.2 knnClassifier.m

Inputs:

- TrainSet: training set + training labels
- TestSet: test set without labels
- k: parameter of the kNN classifier
- TestLabels: test labels (optional input)

Outputs:

- Predictions: predicted class for each observation
- Error: total error of the classifier

This function implements a kNN classifier. It first controls that the number of inputs received is at least equal to the number of required inputs and that the number of columns of the test set has exactly one column less than the training set. Finally it also checks that k >0 and k≤ (cardinality of training set).
Then it uses the Matlab function pdist2() on the training and test sets. This function computes the pairwise distance between two sets of observations by using a certain metric. In here the metric is euclidean for the distance, so the function outputs:

- matrix D of the K smallest pairwise distances to observations in the training set for each observation in the test set in ascending order.
  Size: k x TEST_COLS
- matrix TrainIdx of the indices of the observations in the training set corresponding to the distances in D.
  Size: kxTEST_COLS

The matrix TrainIxd is used to index the training set labels that correspond to the k nearest neighbours. Then, since it is a classification problem, the mode of the k labels is computed in order to define the output. The resulting matrix is the Predictions matrix.
Finally, if the actual labels of the test set are present, the error is computed, thus obtaining the Error matrix.

3 Results

By calculating the accuracy on the reduced training set, we can see by its plot that it decreases with respect to increasing values of k, starting from approximately 93% for k=10 to 86% for k=110. This is due to the fact that a higher k takes into consideration too many neighbouring pixels that are not important to determine the query point's class.
By considering the single digits, the trend is the same, however the accuracy is generally higher. In fact, it's easier to recognize if a digit is for example a 0 or not, compared to actually understanding which digit it is.

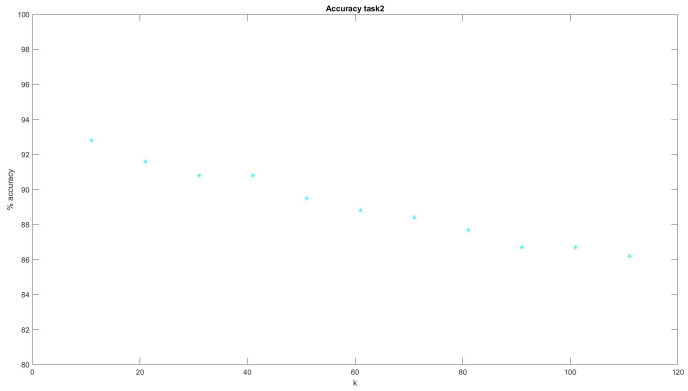


Figure 2: Accuracy

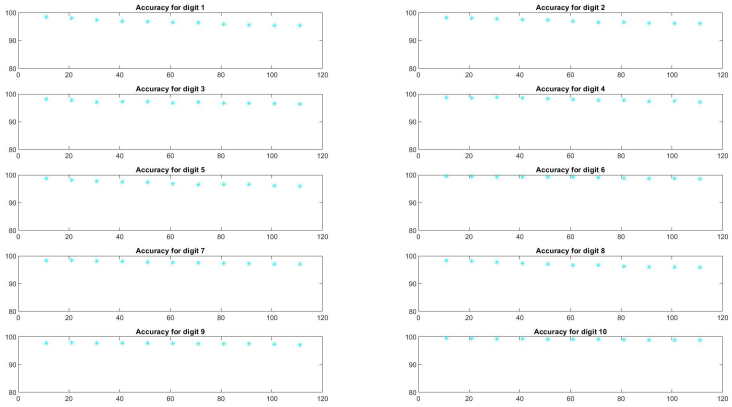


Figure 3: Accuracy on each digit

References

[1] *https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761.*

Machine learning Assignment4

December 2020

Abstract This assignment is divided in two parts:
Task A aims at implementing in MATLAB two simple single-layer neural networks. Specifically they are the perceptron and the adaline algorithms.
Task B addresses more complex cases and aims at getting acquainted with Matlab built-in functions and GUI for neural networks.

1 Introduction

Adaline and Perceptron are two examples of single-unit neural networks [4]. They are by-pattern algorithms, in which weight modifications are computed upon receiving each individual pattern. We assume a linear threshold unit with activation function f(r)=sign(r). The basic implementation is represented in Figure 1: all inputs are weighed and summed together before being multiplied for the activation function.

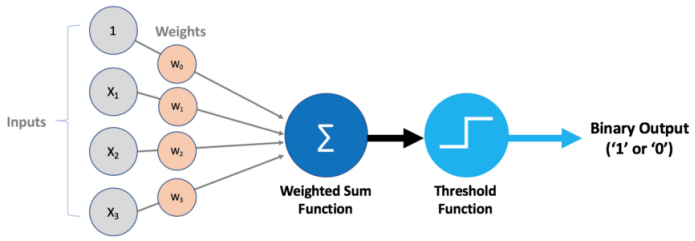


Figure 1: Basic algorithm model

1.1 Perceptron algorithm

The idea for this model is the same as before, but the output of the threshold is used as a feedback value. If the training set is linearly separable, the perceptron learning procedure will find a separating hyperplane for it in a finite number of steps. If not, the procedure doesn't converge.
In pseudo code:

```
w=rand(0,1)
xl=1
for count=1:10000
    r=x(xl)*w
    a=sign(r)
    delta=0.5*(t(xl)-a)
    w=w+eta*delta*x(xl)
    xl++
```

1.2 Adaline algorithm

The adaline algorithm is a perceptron in which, in addiction to a, also r is available outside. It is better than the perceptron in the sense that it converges even for non-linearly separable problems and achieves more robust solutions. In pseudo code:

```
w=rand(0,1)
xl=1
for count=1:10000
    r=x(xl)*w
    delta=0.5*(t(xl)-r)
    if delta<threshold
        break
    w=w+eta*delta*x(xl)
    xl++
```

1.3 Confusion matrix

The results of both algorithms are shown as confusion matrices. This is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (like in this case). Each row of the matrix represents the instances in the predicted class while each column represents the instances in the actual class. This means that the right guesses areon the main diagonal, so it is here that we expect to see higher values.

2 Task A

2.1 The datasets

- **MNIST dataset:** Dataset composed of images of handwritten digits from 0 to 9. Examples: in training set: 600000, in test set: 100000. Classes: 10.
- **Iris dataset:** Reduced version of the original dataset. The original version contains a set of 150 records under five attributes - sepal length, sepal width, petal length, petal width and species. In this example it is modified to have only two linearly-separable classes. Examples in data set: 150. Classes: 2.
- **XOR dataset:** Simple dataset that comes from the truth table of the XOR. Examples in data set: 8. Classes: 2.

2.2 Matlab implementation

2.2.1 main.m

The main loads the three datasets and makes them compliant with the rest of the code. In particular, for the iris dataset: it transforms the labels values into ± 1; for the MNIST dataset: it chooses only two digits to recognize and sets their labels to ± 1; for the XOR dataset: it transforms the labels values into ± 1. Then it calls the functions in order to perform the tasks.

2.2.2 splitDataSet.m

Input:

- x: original dataset
- k: number of training + test set couples

Output:

- x_res: structure containing the training sets and test sets
- num_sets: number of generated sets

This function splits a given dataset in n couples of training set + test set. The number n depends on the k value:

- if k=2: n=1. It splits the set in 2 equal parts, thus generating one couple only.
- if k=ROWS: n=ROWS. Where ROWS indicates the number of examples of the set. It performs leave-one out cross validation. This means that it splits the sample into ROWS training sets of size ROWS-1 and correnspondingROWS test sets of size 1.
- if 2 <k <ROWS: n=k. It performs k-fold cross validation. This means that it splits the sample into k training sets of size ROWS/k and correnspondingtest sets of size ROWS-ROWS/k.

2.2.3 perceptron.m

Input:

- x_total: structure with n training sets and n test sets
- eta: learning rate
- num_sets: how many training+test sets are given in x_total

Output:

- myconfusionMatrix: confusion matrix
- w: weight matrix

This function implements the perceptron algorithm, and returns the confusion matrix and the weight matrix.
In detail: it enters a loop and, at each step, takes a different couple of training and test sets. It generates a random vector or weights, sets a desired learning rate and a threshold to which δ will be compared to. Then it starts a for loop of ten thousands iterations. At each of them, starting from xl=1, it calculates r, a and δ and updates the weight. Then it increases xl and starts again.
At the end of the loop it has computed the desired set of weights and it uses it in the testing phase. It predicts the labels of the test set and then compares them to the real ones and builds the relative confusion matrix. After completing the for loop it averages the matrix and sets it in output.

2.2.4 adaline.m

Input:

- x_total: structure with n training sets and n test sets
- eta: learning rate
- num_sets: how many training+test sets are given in x-total

Output:

- myconfusionMatrix: confusion matrix
- w: weight matrix

This function implements the adaline algorithm, and returns the confusion matrix and the weight matrix.
In detail: it enters a loop and, at each step, takes a different couple of training and test sets. It generates a random vector or weights, sets a desired learning rate and a threshold to which δ will be compared to. Then it starts a for loop of ten thousands iterations. At each of them, starting from xl=1, it calculates r, a and δ, then checks the stopping condition: if δ is lower than a threshold it exits. It finally updates the weight, increases xl and starts again.
At the end of the loop it has computed the desired set of weights and it uses it in the testing phase. It predicts the labels of the test set and then compares them to the real ones and builds the relative confusion matrix. After completing the for loop it averages the matrix and sets it in output.

2.2.5 images.m

This function simply creates the tables of the confusion matrices and saves them as images.

2.3 Results

By considering a small learning rate we obtained the following results.
The perceptron algorithms works well on both the Iris (Figure 2) and the MNIST (Figure 3) datasets, obtaining less than 2% of errors in classification in all cases. In general, it performs better with k=2.
The Adaline algorithm seems to perform worse than the Perceptron but still obtains good results (Figures 4 and 6).
The XOR dataset was used to study the difference in convergence of the two algorithms. The XOr, or "exclusive or", problem is a classic problem in artificial neural networks research. It is the problem of using a neural network to predict the outputs of XOr logic gates given two binary inputs. The problem with these data is that XOr inputs are not linearly separable [3]. The results obtained were the same for Perceptron and Adaline, and they showed that neither of them can cope with the non linearly separability case.

	Positive outcome	Negative outcome
Positive label	38.6667	0
Negative label	0	61.3333

(a) k=2

	Positive outcome	Negative outcome
Positive label	33.3333	0.74074
Negative label	0.2963	65.6296

(b) k=10

	Positive outcome	Negative outcome
Positive label	33.3333	0
Negative label	0	66.6667

(c) k=150

Figure 2: Perceptron on iris dataset

	Positive outcome	Negative outcome
Positive label	44.7244	0.7874
Negative label	0.62992	53.8583

(a) k=2

	Positive outcome	Negative outcome
Positive label	46.0542	0.75241
Negative label	1.3648	51.8285

(b) k=10

	Positive outcome	Negative outcome
Positive label	46.2205	0.70866
Negative label	0.86614	52.2047

(c) k=150

Figure 3: Perceptron on MNIST dataset

	Positive outcome	Negative outcome
Positive label	38.6667	0
Negative label	13.3333	48

(a) k=2

	Positive outcome	Negative outcome
Positive label	34.0741	0
Negative label	23.7037	42.2222

(b) k=10

	Positive outcome	Negative outcome
Positive label	33.3333	0
Negative label	2.6667	64

(c) k=150

Figure 4: Adaline on iris dataset

	Positive outcome	Negative outcome
Positive label	34.4882	11.0236
Negative label	2.5197	51.9685

(a) k=2

	Positive outcome	Negative outcome
Positive label	34.7069	12.0997
Negative label	4.4444	48.7489

(b) k=10

	Positive outcome	Negative outcome
Positive label	32.7559	14.1732
Negative label	4.252	48.8189

(c) k=150

Figure 5: Adaline on MNIST dataset

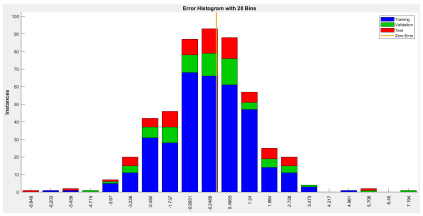
	Positive outcome	Negative outcome
Positive label	25	0
Negative label	50	0

(a) k=2

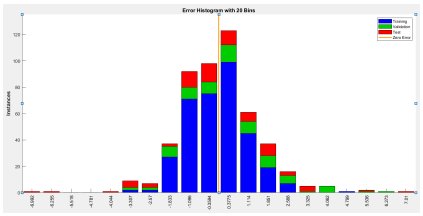
	Positive outcome	Negative outcome
Positive label	25	0
Negative label	50	0

(b) k=10

Figure 6: Perceptron and Adaline on XOR dataset

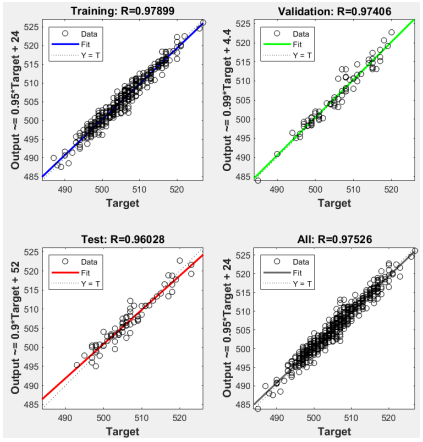


(a) Chemical dataset, 10 hidden units

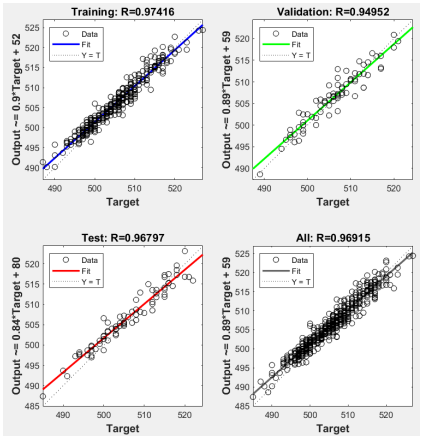


(b) Chemical dataset, 20 hidden units

Figure 7: Error histogram



(a) Chemical dataset, 10 hidden units



(b) Chemical dataset, 20 hidden units

Figure 8: Regression plot

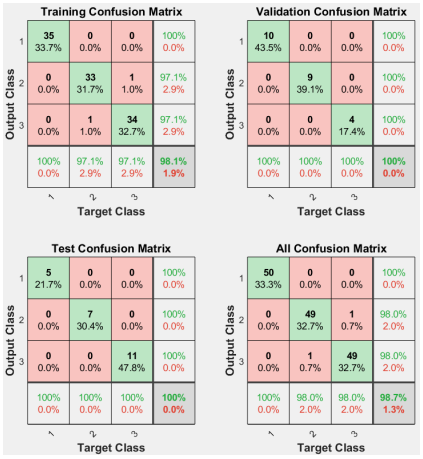
3 Task B

3.1 Fitting functions

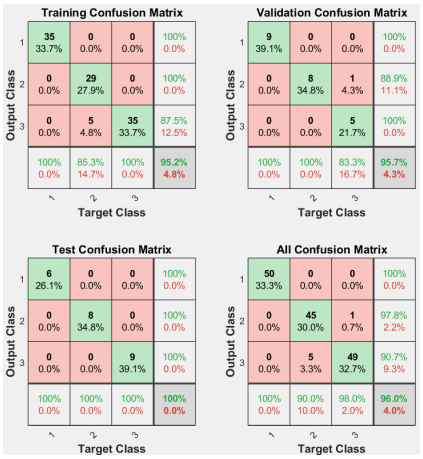
This task is about following Matlab's tutorial on neural networks used to find fitting functions for different datasets: [2]. Here are shown the results obtained on the chemical dataset. Figure 7 represents the error histograms of the chemical dataset for different values of hidden layers. The error is computed by computing the difference between output ans target.
Figure 8 represents the regression plots of the chemical dataset for different values of hidden layers. The small circles represent data, the line indicates the fitting line, the dotted line indicates Y=T, where Y is the output, and T is the target.

3.2 Pattern recognition

This task is about following Matlab's tutorial on neural networks used to classify patterns: [1].
Figure 7 represents the confusion matrices of the iris dataset for different values of hidden layers. On the main diagonal of the matrix (in green) are shown the correct classifications. By increasing the number of hidden units the results are generally improved.



(a) Iris dataset, 10 hidden units



(b) Iris dataset, 20 hidden units

Figure 9: Confusion matrix

References

[1] *https://it.mathworks.com/help/deeplearning/gs/classify-patterns-with-a-neural-network.html.*

[2] *https://it.mathworks.com/help/deeplearning/gs/fit-data-with-a-neural-network.html.*

[3] *https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b.*

[4] *https://towardsdatascience.com/understanding-basic-machine-learning-with-python-perceptrons-and-artificial-neurons-dfae8fe61700.*

Machine learning Assignment5

December 2020

Abstract This assignment is divided in two parts:
Task A aims at implementing in MATLAB a simple autoencoder.
Task B addresses more complex cases and aims at getting acquainted with Matlab's pretrained convolutional neural networks and their uses.

1 Task A

1.1 Introduction

The autoencoder is an unsupervised artificial neural network that firstly compressed and encodes data and then reconstructs it back from the reduced representation to a representation that is as close to the original input as possible. By design, it reduces data dimensions by learning how to ignore the noise in the data. Its structure is shown in Figure 1.
In this assignment the autoencoder is a multi-layer perceptron neural network with one input layer,

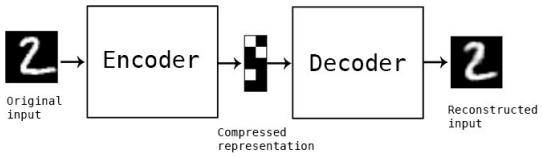


Figure 1: Autoencoder model

one hidden layer and one output layer.
It implements an identity mapping: input = target. So it is an unsupervised problem, specifically a self-supervised problem.
The number of hidden units should be smaller than the number of inputs and outputs since it learns a more compact version of the input. The pattern of activations in the hidden layer is the representation of the problem: what we desire is that similar patterns have similar representations.
The number of hidden layers of this autoencoder is nh = 2 because the 2 activations are used as axis of the activation space to plot the data.

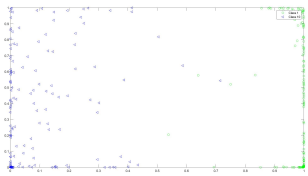
2 Matlab code

2.0.1 main.m

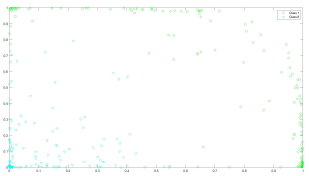
The code loads the MNIST dataset lines relative to two chosen digits. It shuffles the examples and extracts subsets, by finally merging the datasets.
It then trains an autoencoder with the dataset by using the Matlab function trainAutoencoder() and obtains the compressed representation of the data by using the Matlab function encode(). In the end it generates the plots by using the given function plotcl() and saves the plotted results as images.

2.1 Results

Hereby are shown the results obtained. Each point represents a digit, and the same digits have the same color and shape. Figure 2 shows two couples of digits with different appearance: they are linearly separable. Figure 3 shows two couples of digits with similar appearance, and the results are more mixed than in Figure 2, so they're not linearly separable.

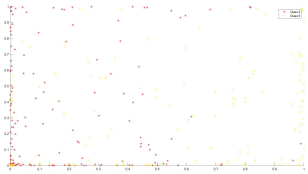


(a) Digits 1 and 0

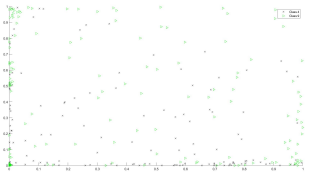


(b) Digits 1 and 8

Figure 2: Digits with different appearance



(a) Digits 3 and 5



(b) Digits 4 and 9

Figure 3: Digits with same appearance

3 Task B

3.1 Convolutional neural networks

A Convolutional Neural Network is a Deep Learning algorithm which can take an image in input and assign importance (learnable weights and biases) to various aspects/objects in the image. [6] CNNs vary in structure but have some common layers:

- **Convolutional layer**: The layer's parameters consist of a set of learnable filters K. Each filter shifts over the whole image, performing convolution (Figure 4).

  The objective of the Convolution Operation is to first extract Low-Level features such as edges, color, gradient orientation, etc. With added convolutional layers, the architecture adapts to the High-Level features as well. [1]

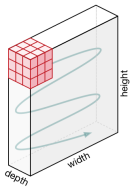


Figure 4: Kernel shifting on image

- **Pooling layer**. It is a form of non-linear down-sampling useful for extracting dominant features which are rotational and positional invariant.
  There are two types of Pooling: Max Pooling (that returns the maximum value from the portion P of the image covered by the Kernel) and Average Pooling (that returns the average of all the values from P). The pooling layer serves to progressively reduce the spatial size of the representation, the memory footprint and the amount of computation in the network, and hence to also control overfitting.

- **ReLu**: the name is the abbreviation of rectified linear unit, which applies the non-saturating activation function f(x)=max(0,x). It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.

- **Fully connected layer**: the high-level reasoning in the neural network. Neurons in a fully connected layer have connections to all activations in the previous layer.

- **Loss layer**: it specifies how training penalizes the deviation between the predicted (output) and true labels and is normally the final layer of a neural network. Various loss functions may be used and usually Softmax loss is used for predicting a single class of K mutually exclusive classes.

3.2 The pretrained neural networks

Pretrained networks have different characteristics that matter when choosing one to apply to a problem. The most important characteristics are network accuracy, speed, and size. Here are the convolutional neural networks used in the following tasks. They are all trained on the ImageNet database and classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.[4]

- **GoogLeNet** is 22 layers deep and has an image input size of 224-by-224.
- **AlexNet** is 8 layers deep and has an image input size of 227-by-227.
- **ResNet-18** is 18 layers deep and has an image input size of 224-by-224.

3.3 The datasets

- **ImageNet** is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided.
- **Merchdata** is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (cap, cube, playing cards, screwdriver, and torch). This data can be used to quickly try transfer learning and image classification. The images are of size 227-by-227-by-3.

3.4 Goals when using a pretrained neural network

The goals, when using a pre-trained neural networks, can be:

- **classification**: use them as black boxes to classify new images.
- **feature extraction**: use them by reading the activation values from a layer other than the output one. In this way all the previous layers are used as feature extractors. The activations from an intermediate layer can be used as an input for a final classifier, e.g., a shallow multi-layer perceptron.
- **transfer learning**: use them as the initialization to learn a new (but similar) task.

3.5 Deep learning Matlab demo

This task consisted in following Matlab's tutorial on deep learning [2]. The first step in this task is to understand how to use deep learning to identify objects on a live webcam by using only 10 lines of Matlab code. The neural network used for this goal is the AlexNet and the camera is simply the computer's camera.

3.5.1 Results

Some of the results I obtained are shown in Figures 5 to 8.

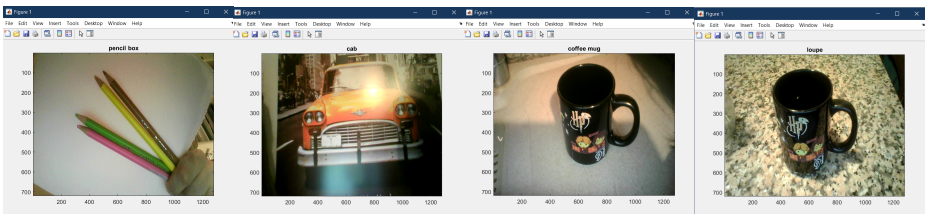


Figure 5: Pencils Figure 6: NY cab Figure 7: Cup ver1 Figure 8: Cup ver2

3.6 Use pretrained Convolutional Neural Network

This task is about using a pretrained deep neural network as a starting point to learn a new task [5]. The neural nework is the GoogLeNet and we use it by giving it some new images as inputs to see if it can recognize them.

3.6.1 Results

In order to try it I downloaded some images from Google, resized them to match the desired input size, and used the classify() function of the neural network on them. Some of the results I obtained are shown in Figures from 9 to 14.

Figure 9: Pizza

Figure 10: Penguin

Figure 11: Quokka

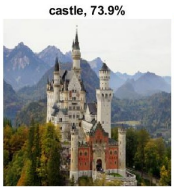


Figure 12: Castle

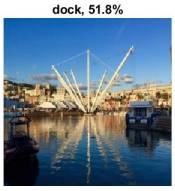


Figure 13: Genova

Figure 14: Bologna

It also possible to display the top five predicted labels and their associated probabilities as a histogram: since many categories are similar, it is common to consider the top-five accuracy when evaluating networks.

3.7 Transfer learning via feature extraction

This task consists in extracting learned image features from a pretrained CNN and using those features to train an image classifier. This is done by following Matlab's tutorial [3].
The dataset is the MerchData dataset, that is split to have 55 training images and 20 validation images. The neural network is the ResNet-18 network.
The feature extracted from the training images at the selected internal layer of the ResNet-18 layer are used as predictor variables to fit a multiclass support vector machines (SVM).

3.7.1 Results

In Figure 15 is reported a classification of four different random images from the MerchData dataset.

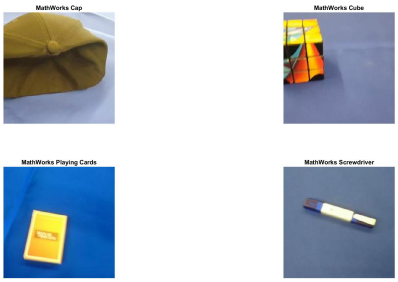


Figure 15: Predictions

3.8 Transfer learning via retraining

This task consists in using transfer learning to retrain a convolutional neural network to classify a new set of images. This was done by following Matlab's tutorial [1]. The dataset is the MerchData dataset and the CNN is the GoogLeNet.

- The main idea is to replace the last layers of the CNN with new ones that adapt to our problem. The new last layer we'll specify is a fully connected layer with the number of outputs equal to the desired number of classes in the new data set.
- Optionally, it's possible to "freeze" the weights of earlier layers in the network by setting their learning rates to zero: this means that their gradient won't be updated and this can significantly speed up network training. Here I've frozen the first 10 layers. In general, it's desirable to have fast learning in the new layers, slower learning in the middle layers, and no learning in the earlier, frozen layers.
- Then it's a good approach to use an augmented image datastore to resize the training images and optionally perform additional operations to perform on the training images: flipping, translating, scaling. Data augmentation helps prevent the network from overfitting.
- You also need to specify the number of epochs to train for, the mini-batch size and validation data.
- At last, you can actually train the CNN.

References

[1] *https://en.wikipedia.org/wiki/Convolutional*$_n$*eural*$_n$*etwork*.

[2] *https://it.mathworks.com/help/deeplearning/gs/try-deep-learning-in-10-lines-of-matlab-code.html*.

[3] *https://it.mathworks.com/help/deeplearning/ug/classify-image-using-googlenet.html*.

[4] *https://it.mathworks.com/help/deeplearning/ug/data-sets-for-deep-learning.html*.

[5] *https://it.mathworks.com/help/deeplearning/ug/pretrained-convolutional-neural-networks.html*.

[6] *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*.