UNIVERSITÀ DEGLI STUDI DI GENOVA

DIBRIS

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

BIOENGINEERING, ROBOTICS AND SYSTEM ENGINEERING

MACHINE LEARNING FOR ROBOTICS 1

Cumulative reports

Assignment labs 1, 2, 3, 4

Author:

Gavagna Veronica

Student ID:

S5487110

Professors:

Stefano Rovetta

s.y. 2022/23

Contents

Implementation of the Naive Bayes Classifier in Python

Veronica Gavagna, *Student*, UNIGE

October 2022

Abstract

This report shows the implementation of the Naive Bayes Classifier in Python programming language. The aim of this algorithm is to classify the elements of a certain test data set assuming that the classes are known a priory. In order to do that, the program has to be trained first, by making it process a training data set which, for each observation, also include the output class. In this assignment, it is considered a data set containing observations about weather and the target about going or not going out to play tennis. Each observation is associated to a binary class that represents either the decision to stay at home (no) or to go playing tennis outside (yes). Three classifier are implemented, the first using the Naive Bayes Classifier, the second one adding the Laplace (additive) smoothing to avoid assigning zero-probability to values of the attributes that do not appear in correspondence of a certain class and, the third one using the log probabilities if you have large data sets.

Introduction

A classifier is a special learning machine whose purpose is to solve classification problems. All common decision-making problems are characterized by inputs, outputs, and references. The inputs are observations, each a vector consisting of different values of a given random variable. The output is the decision that the learning machine makes. Reference is the natural state. In general, the goal of a learning machine is to find the rule (known as the decision rule) that produces the most appropriate decision $y(w)$ given observations and random conditions in nature. Classification is a decision problem where a set of decisions agrees with a set of natural states (classes). In other words, the decision is only to recognize the correct state of nature. So the classifier is just a rule that takes an observation ex and outputs the class $w = y(x)$. In particular, according to Bayesian Decision Theory, the desired algorithm should output the class that minimizes the conditional risk $R(y(e_x)|x)$.
A convenient way to implement this is given certain observations. B. Evaluate the discriminant functions for all possible natural states. Each of these functions is defined as $g_i(x) = R(w_i|x)$ (i = 1, ..., c number of classes).
The classifier then selects the class w_i associated with the largest discriminant function $g_i(x)$ for the given pattern x. Furthermore, it is possible to consider transformed discriminant functions and show that as long as the transformation used is monotonically increasing, the results do not change.

The Naive Bayes Classifier (NBC) is one of the simplest classifiers. Its simplicity stems from the fact that it is based on simple assumptions. This approximation consists of the assumption that the input variables are all independent of each other. as a result:

$$P(\mathbf{x}|w_i) = P(x_1|w_i)*P(x_2|w_i)*...*P(x_d|w_i) \quad i = 1, ..., c$$

In fact, it is easy to see that NBC is not the most efficient classifier choice, since observed attributes are often interdependent. The NBC discriminant function is obtained by applying a monotonically increasing transformation to the Bayesian formula (hence the name of the classifier). Specifically, each decision function is defined as the numerator of the problem formula. The denominator is not shown as it is the same for each class considered. The formula is:

$$g_i = P(w_i) \prod_{j=1}^{d} P(x_j|w_j) \qquad i = 1, ..., c$$

Since the purpose of a classifier is to compute and compare all discriminant functions, the algorithm necessarily knows both the natural state probabilities and the probabilities of each input variable's value given its class. must be Classifiers typically don't have this kind of information, so frequencies are used instead. This is an additional approximation, since frequencies and probabilities only match for an almost infinite number of observations. Another problem arises from lack of data. Some values of the input variables may not appear according to certain classes in the training dataset and have associated conditional frequencies of zero. So-called Laplacian (additive) smoothing can be used to solve this problem. This technique takes into account hypothetical prior information about attribute values and assumes that attribute values are all equally likely to occur in the absence of observations. However, to do this, we need to know in advance how many possible values there are. Taking this into account, the conditional frequency of the problem changes to: Before smoothing:

$$P(x_[j] = k|w_i) = \frac{n.of times x_i = k in class wi}{n.of observations of class w_i} = \frac{N_{k,w_i}}{N_{w_i}}$$

After smoothing:

$$P(x_[j] = k|w_i) = \frac{N_{k,w_i} + a}{N_{w_i} + av_i}$$

where a is a number that represents the degree of mistrust in the data set: if $a < 1$ the data set is considered more legitimate than the hypothetical prior information about the data, if $a > 1$ the data set is considered less legitimate than the hypothetical prior information about the data and, finally, if $a = 1$ the two are considered to have the same legitimacy. v_j instead is the number of possible values that the feature x_j can take. If no observations are available, the formula shown above simply becomes:

$$P(x_j = k|w_i) = \frac{1}{v_j}$$

This equation explicitly states the assumption about the hypothetical equal probability of the values underlying Laplace (additive) smoothing. Another issue related to limited datasets is that not all classes appear in the training dataset, but this paper assumes that this is not possible.

Implementation in Python

The Naive Bayes Classifier (hereafter NBC) was implemented as function in Python language. A main function was created to manage data preprocessing and NBC functions. The data set used contains 14 weather observations that consist of 4 attributes (overcast, temperature, humidity, wind). Each observation is assigned a binary class that represents a decision to stay at home (no) or play tennis outside (yes). Note, that the following algorithm can support any type of classification base data set, and can be easily be personalize and adapted by changing only few parameters.

Function Main

First, the main function does the data preprocessing: The first task when working with datasets is to process the data before using it. This is because they are often not presented in a form suitable for handling. For example, record elements usually need to be converted from non-numeric format to numeric format. Other common operations include: handle missing data and removing patterns with out-of-range values.
In the case under consideration, the sample data set "weather.txt" is first converted into numerical form using the library "preprocessing" of "sklearn" of Python (adding 1 to each values to avoid having 0 values).

```
************************** Original DataSet **************************

    #Outlook Temperature Humidity  Windy Play
0   overcast         hot     high  False  yes
1   overcast        cool   normal   True  yes
2   overcast        mild     high   True  yes
3   overcast         hot   normal  False  yes
4      rainy        mild     high  False  yes
5      rainy        cool   normal  False  yes
6      rainy        cool   normal   True   no
7      rainy        mild   normal  False  yes
8      rainy        mild     high   True   no
9      sunny         hot     high  False   no
10     sunny         hot     high   True   no
11     sunny        mild     high  False   no
12     sunny        cool   normal  False  yes
13     sunny        mild   normal   True  yes
```

Figure 1: Original Dataset

```
############## Numeric DataSet Matrix ##############

[[1 2 1 1 2]
 [1 1 2 2 2]
 [1 3 1 2 2]
 [1 2 2 1 2]
 [2 3 1 1 2]
 [2 1 2 1 2]
 [2 1 2 2 1]
 [2 3 2 1 2]
 [2 3 1 2 1]
 [3 2 1 1 1]
 [3 2 1 2 1]
 [3 3 1 1 1]
 [3 1 2 1 2]
 [3 3 2 2 2]]
```

Figure 2: Numeric Dataset

After data preprocessing, the data set is randomly split into a training set containing 10 randomly selected observations and test set containing the remaining 4 patterns. The results are then checked if 0 (or less) values are present, if not the program raise an exception, otherwise NBC function is called.

```
************************* Training Set *************************

[[2 3 2 1 2]
 [2 3 1 1 2]
 [3 2 1 1 1]
 [1 2 1 1 2]
 [1 3 1 2 2]
 [3 3 2 2 2]
 [3 1 2 1 2]
 [2 1 2 2 1]
 [1 1 2 2 2]]

************************* Test Set *************************

[[2 1 2 1 2]
 [2 3 1 2 1]
 [3 2 1 2 1]
 [1 2 2 1 2]
 [3 3 1 1 1]]
```

Figure 3: Training and Test Set

Before starting the classification, it is important to check if the number of columns of the test set and the training set corresponds. If the test set has one column less, one column is added:

```
if (test set columns are equal
    to training set columns - 1):

    insert a column of zeros at
    the end of the test set;
```

After that, NB classifier function is called.

NB classifier functions

First, to compute the likelihood probabilities it is necessary to:

- Compute the prior probabilities:

$$P(play = YES) = \frac{n.\,of\,YES}{n.\,YES + n.\,NO}$$

$$P(play = NO) = \frac{n.\,of\,NO}{n.\,YES + n.\,NO}$$

- Compute the posterior probabilities for each values of each classes.
  Example sunny/yes:

$$P(Sunny|Yes) = \frac{n.\,of\,SUNNY\ assoc.\,to\,YES}{total\,n.\,of\,SUNNY}$$

- Then, all the posterior probabilities are stored into a 3D matrix so all the components of the likelihood probability are available for each pattern. Example of one row:

$$P(Play = Yes|Weather = Sunny,$$

$$Temperature = Mild, Humidity = normal,$$

$$windy = false) = P(play = YES) * P(Sunny|Yes) *$$

$$*P(Mild|Yes) * P(normal|Yes) * P(false|Yes)$$

```
for every values of the target:

    compute the prior probability;

for each possible values
      of the target:

    for every possible values
          of each classes
          (except for the target):

          compute the posterior
          probability;
```

After computing all the probabilities and stored them into a 3D matrix, to predict the target value is only necessary to compare the likelihood probability for target class "YES" and target class "NO" and choose the higher one.

```
for every row of the test set:

    pick the index of the
    max probability;

    assign the target values
    corresponding to the
    max probability;
```

NB classifier Laplace Smoothing

The first implementation that can be done of NB classifier was its improvement using Laplacian additive smoothing. This extension is done by simply introducing the parameters a and v_j into the evaluation of the conditional frequency of attribute values according to formula explained in the introduction part:

$$P(x_j = k|w_i) = \frac{N_{k,w_i} + a}{N_{w_i} + av_i}$$

NB classifier Logarithmic

An addictive implementation that can be done of the NB classifier is to use the logarithmic probabilities. This implementation is useful in case of large data sets to avoid numerical errors while multiplying lots of frequencies together, and it can be done by transforming logarithmically all the probabilities and turning all the multiplications into sums.

Think further

Using the NB classifier with logarithmic transformation, it is possible to apply the algorithm previous explained to classify dataset with continuous variables (like "Iris dataset").
Variable values are used to compute probabilities by counting, but theory tells us that probabilities may be obtained by probability mass functions directly if they are known analytically. The four features was made binary by computing the average of each and replacing each individual value with 1 if it is above the mean and 0 if it is below.
After converting values into binary values, the same logarithmic NB classifier is used to classify the target.

Results

To evaluate the classification, the accuracy using the actual target values and the predicted values is computed through the following formula:

$$error\ rate = \frac{total\ error}{n.\ of\ row\ of\ the\ testset}$$

Conclusion

After running the script a few times with different test set selected randomly, we see that the error rate reaches values very high. The inefficiency of this classifier is probably due to the different approximations considered in this particular example. First of all, a very limited data set means that the estimated frequencies are far from the actual probabilities. Furthermore, the characteristically simple assumptions of classifiers make them unsuitable for datasets such as the one being analyzed where the properties of the observations are interdependent. In summary, the naive Bayes classifier is certainly not the most effective classifier, but its conceptual clarity and ease of implementation make it an interesting study case.

References

[1] S. Rovetta, *Machine Learning for Robotics I notes and slides*, Genova University, Master Course in Robotics Engineering

Implementation of Linear Regression in Python

Veronica Gavagna, *Student*, UNIGE

October 2022

Abstract

This report shows an implementation of a linear regression modeling technique in Python.
The purpose of this method is to use measured data to determine a linear model that approximates functional dependence. Here we consider three linear regression analysis cases. Two of them consist of one-dimensional problems. This means that observations in your dataset consist of only one feature, except that they are assigned to targets. Therefore, the linear relationship that fits the data is just a straight line. The difference between the first two problems is simply that in the first case the regression line must pass through the origin of the data space, whereas in the second case it does not.
For the third regression problem, it is multidimensional so, in this case, the observation consists of multiple features. Therefore, the linear relationship that fits the data belongs to 4-dimensional space and therefore cannot be represented on the plane. It is important to note that the proposed experiment did not use the entire dataset to derive the fitted linear relationship. In practice, the dataset is split into a training set and a test set, and only the former is used to evaluate the relationship weights in question. This approach allows us to compare the average error associated with the training set elements with the average error associated with the test set elements.

Introduction

Linear regression modeling techniques are often used to solve supervised problems with numerical targets. This simply means that each observation in the training set is associated with the target, and targets are just numbers. As mentioned earlier, this technique aims to determine the linear model that best approximates the dependence of the observed features on the target. A minimization is performed on the mean of the loss function to find the optimal weights that accomplish the task at hand. This function of the target and corresponding linear model output can be defined in various ways. Here the so-called "squared error loss function" is considered and defined as:

$$\lambda_{SE}(t, y) = (t - y)^2$$

This loss function is widely used because it has several advantages. First of all, it is flat, so all errors contribute positively. Then it increases more than linearly. Which means, the larger the error, the greater the weight. Finally, it is differentiable with respect to the model output.
By evaluating the average of λ_{SE} over the entire dataset, the mean squared error objective function is derived:

$$J_{MSE} = \frac{1}{N}\sum_{l=1}^{N}(t_l - y_l)^2$$

For fixed data, it can be shown that the "mean squared error target" as a function of the model weights is parabolic. Therefore, the formula for calculating the optimal weights is obtained by solving the following formula with a vector of variables w:

$$\nabla J_{MSE} = \frac{\delta J_{MSE}}{\delta \mathbf{w}} = 0$$

As mentioned earlier, this paper considers three different study cases. However, when the number of weights to compute changes, the formulas for determining them are slightly different.
The three problems are:

- One-dimensional linear regression problem:
  A problem characterized by a one-dimensional dataset. So the linear model to be determined must be a straight line and pass through the origin, so we have the following equation:

  $$y = w_1 x_1$$

  In this case, the weights that minimize J_{MSE} are given by:

  $$w_1 = \frac{\sum_{l=1}^{N}(x_l t_l)}{\sum_{l=1}^{N} x_l^2}$$

- One-dimensional linear regression problem with offset:
  A problem characterized by a one-dimensional dataset. So the linear model to be determined is a line and we do not have to force it through the origin, so we have the following equation:

  $$y = w_1 x_1 + w_0$$

  In this case, the weights that minimize J_{MSE} are given by:

  $$w_1 = \frac{\sum_{l=1}^{N}(x_l - \bar{x})(t_l - \bar{t})}{\sum_{l=1}^{N}(x_l - \bar{x})^2}$$

$$w_0 = \bar{t} - w_1\bar{x}$$

Where:

$$\bar{x} = \frac{1}{N}\sum_{l=1}^{N} x_l \quad \bar{t} = \frac{1}{N}\sum_{l=1}^{N} t_l$$

- Multidimensional linear regression problem:
  A problem featuring a three-dimensional data set. Therefore, the linear model to be determined belongs to 4D space and has the equation:

  $$y = w_1x_1 + w_2x_2 + w_3x_3$$

  In this case, the vector of three weights that minimizes J_{MSE} is given by the formula:

  $$\mathbf{w} = (X^TX)^{-1}X^T\mathbf{t}$$

  where X is the matrix consisting of the 3D observations of the dataset and t is the vector of targets.

Two data sets come into examination, depending on the research case:

- For the first task, the dataset "Turkish Stock Exchange" is considered. Consisting of 536 one-dimensional observations (Standard and Poor's 500 Return Indices) and their corresponding 536 targets (MSCI Europe Indices), this data collection contains information about two equity indices.
- For the second task, consider the "Motor Trend Cars" data set. Consisting of 32 3D (displacement, horsepower, weight) observations and corresponding 32 targets (miles per gallon), this data collection contains information about many different types of vehicles. The second study case is one-dimensional, so only one of the following three characteristics is considered for each observation: weight.
- The third task also considers the "Motor Trend Cars" data set, but with all features.

Implementation and Results

The linear regression modeling technique was implemented in Python. A main function is created and manages the application of the linear regression model to the above data set.
However, it is important to note that the linear regression function can handle any type of data set that can be organized as a matrix: the rows correspond to observations, the last column is the target, and all other columns correspond to attributes of observations.

Function Main

The main function performs three main tasks:

1. Data preprocessing:
   The first task when working with datasets is to process the data before using it. This is because they are often not presented in a form suitable for handling. Here, the "Turkish Stock Exchange" and "Motor Trend Cars" datasets are simply imported as dataframes from .csv files. The two data frames are then converted into numeric matrices.
2. A linear regression model fits as follows:
   This second task is to evaluate the weights of the linear regression model on several different cases. To do this, call the one-dimensional linear regression function and the multidimensional linear regression function to compute the least-squares solution. After the desired weights are returned from the function, a linear model is determined and plotted in data space if possible. Five different situations are considered:
   - First case: The regression problem considered is the first one (one-dimensional). For linear model fitting, the entire Turkish Stock Exchange dataset is considered as the training dataset.

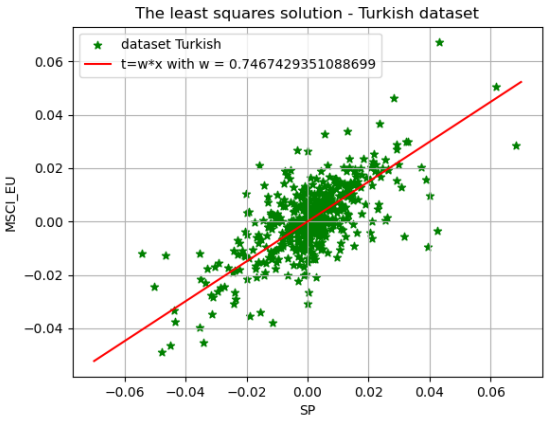


Figure 4: One-dimensional problem without intercept - Turkish Dataset

- Second case: The regression problem considered is the first one (one-dimensional). For linear model fitting, 10% of the random "Turkish Stock Exchange" data set is considered as the training data set. This experiment is repeated for a 10 different subsets and compered two by two.

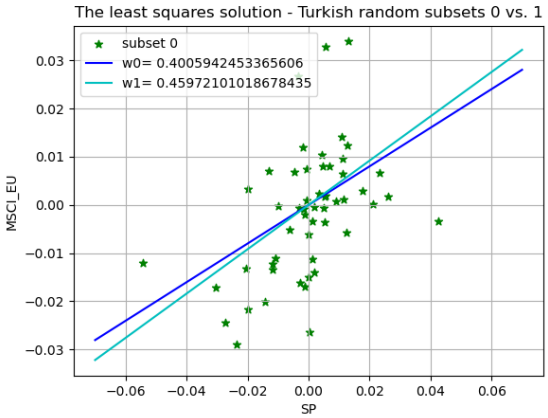


Figure 5: One-dimensional problem without intercept Turkish random Subset 0 vs. 1

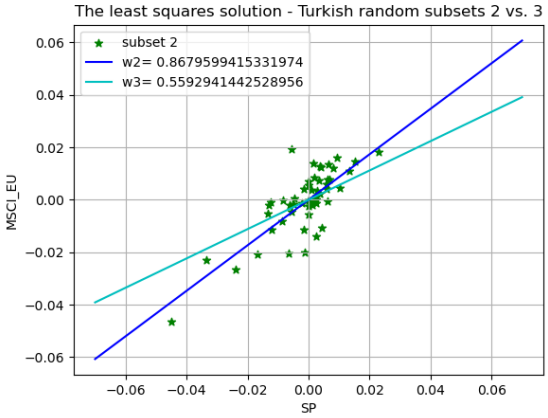


Figure 6: One-dimensional problem without intercept Turkish random Subset 2 vs. 3

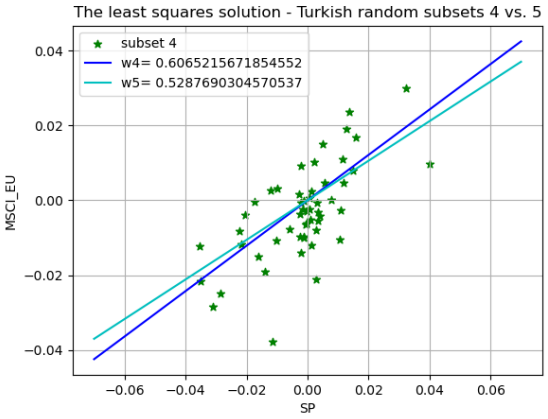


Figure 7: One-dimensional problem without intercept Turkish random Subset 4 vs. 5

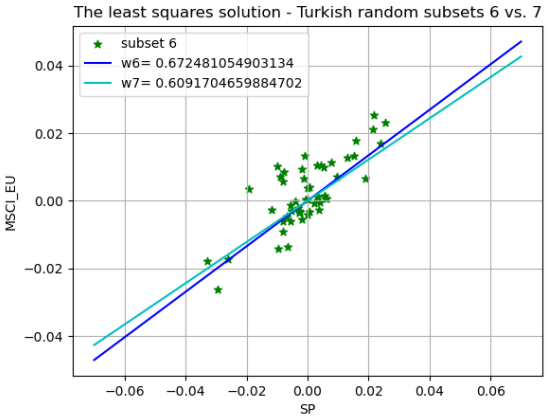


Figure 8: One-dimensional problem without intercept Turkish random Subset 6 vs. 7

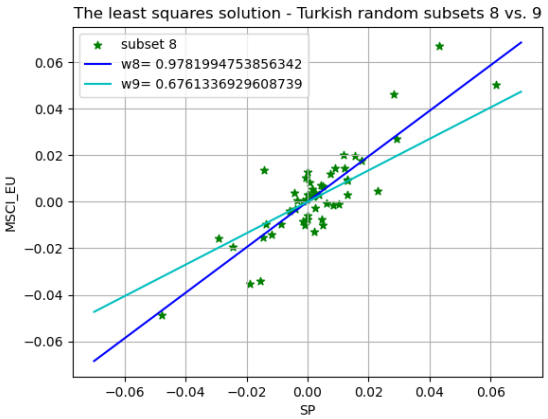


Figure 9: One-dimensional problem without intercept Turkish random Subset 8 vs. 9

- Third case: The regression problem considered is the first one (one-dimensional). For linear model fitting, 10% of the "Turkish Stock Exchange" data set is considered as the training data set. This experiment is repeated for a 10 different subsets and compered two by two.

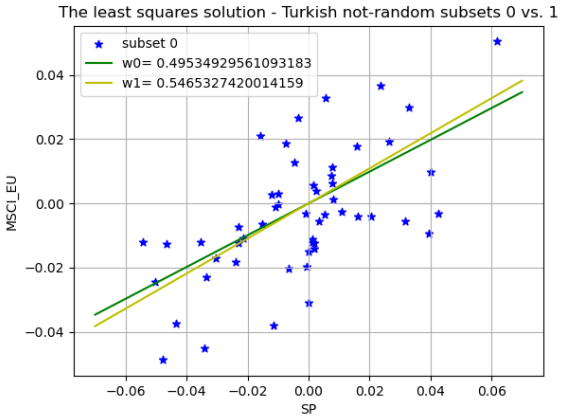


Figure 10: One-dimensional problem without intercept Turkish Subset 0 vs. 1

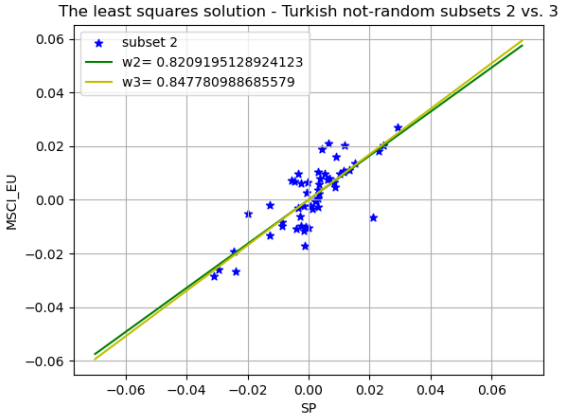


Figure 11: One-dimensional problem without intercept Turkish Subset 2 vs. 3

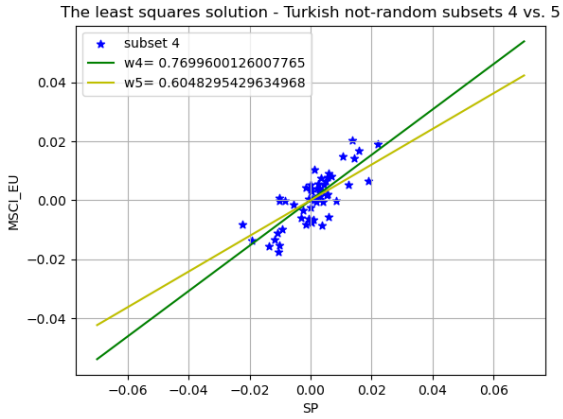


Figure 12: One-dimensional problem without intercept Turkish Subset 4 vs. 5

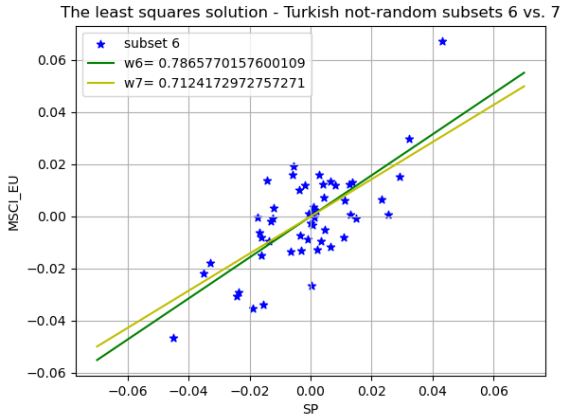


Figure 13: One-dimensional problem without intercept Turkish Subset 6 vs. 7

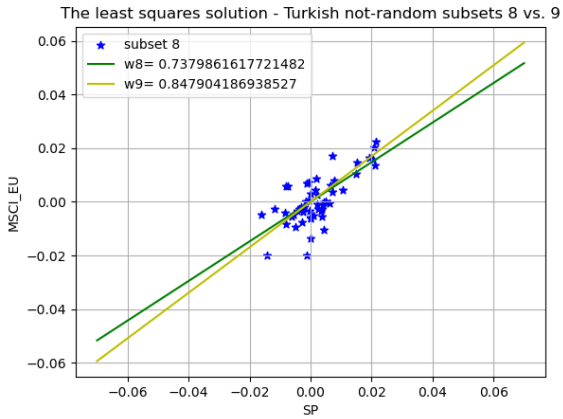


Figure 14: One-dimensional problem without intercept Turkish Subset 8 vs. 9

- Fourth case: The regression problem considered is the second (one-dimensional with offset). For linear model fitting, the "Motor Trend Cars" dataset is considered as the training dataset.

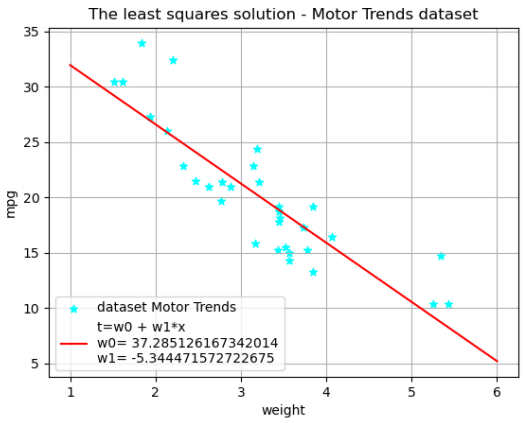


Figure 15: One-dimensional problem with offset Motor Trends Dataset

- Fifth case: The regression problem considered is the third (multidimensional). For linear model fitting, the entire Motor Trend Cars dataset is considered the training set and used to predict the mpg parameter.

Original mpg	Predicted mpg	Error
21,00	23,57	12,24%
21,00	22,60	7,62%
22,80	25,29	10,92%
21,40	21,22	0,86%
18,70	18,24	2,46%
18,10	20,47	13,11%
14,30	15,57	8,85%
24,40	22,91	6,10%
22,80	22,04	3,33%
19,20	20,04	4,38%
17,80	20,04	12,59%
16,40	15,77	3,85%
17,30	17,06	1,38%
15,20	16,87	11,00%
10,40	10,32	0,76%
10,40	9,36	10,00%
14,70	9,21	37,34%
32,40	26,61	17,86%
30,40	29,28	3,70%
33,90	28,04	17,29%
21,50	24,60	14,43%
15,50	18,75	21,00%
15,20	19,09	25,60%
13,30	14,55	9,39%
19,20	16,66	13,21%
27,30	27,62	1,17%
26,00	26,02	0,09%
30,40	27,74	8,73%
15,80	16,50	4,45%
19,70	20,99	6,54%
15,00	12,82	14,55%
21,40	23,03	7,61%

Table 1: Comparison with original mpg and predicted mpg

3. The final task consists of both tuning and testing parts:
   For the first task, the same rules apply as for the previous task. The only difference is that in this case only 5% of the dataset is used to derive the fitted linear model. The test part simply takes the remaining 95% of the data set, computes the squared error of each item with respect to the model's corresponding output, and finally evaluates the mean error across the test set. The mean squared error associated with the training set elements is also obtained. These two procedures are repeated 1000 times with different samples. Finally, the mean squared error values associated with the training set and the mean squared error values associated with the test set are displayed using histograms. The procedure described here is performed for each of the three study cases described in the previous section. The only difference is that the percentages for the "Motor Trends" dataset are 30% and 70% because of the limited number of observations.

   - One-dimensional linear regression problem:

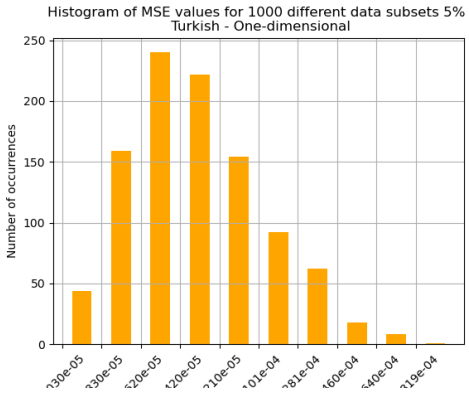


Figure 16: Mean square error - 1000 x 5% Turkish Dataset - One-dimensional

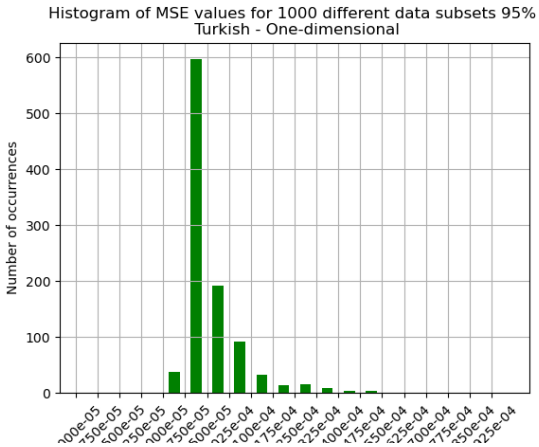


Figure 17: Mean square error - 1000 x 95% Turkish Dataset - One-dimensional

- One-dimensional linear regression problem with offset:

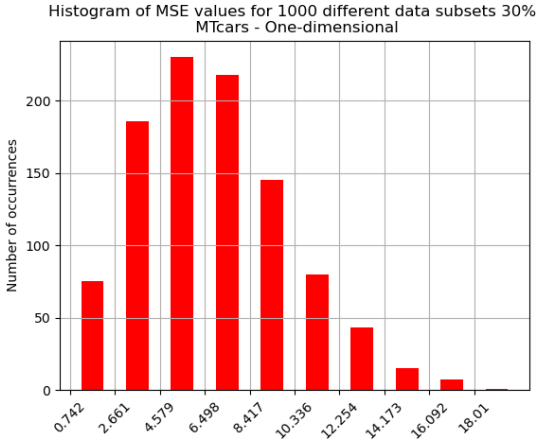


Figure 18: Mean square error - 1000 x 30% Turkish Dataset - One-dimensional

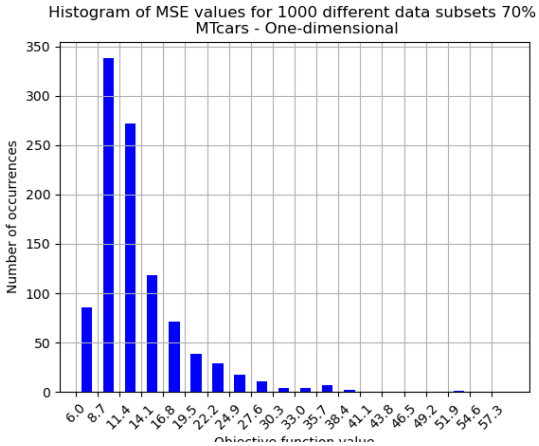


Figure 19: Mean square error - 1000 x 70% Motor Trends Dataset - One-dimensional

- Multidimensional linear regression problem:

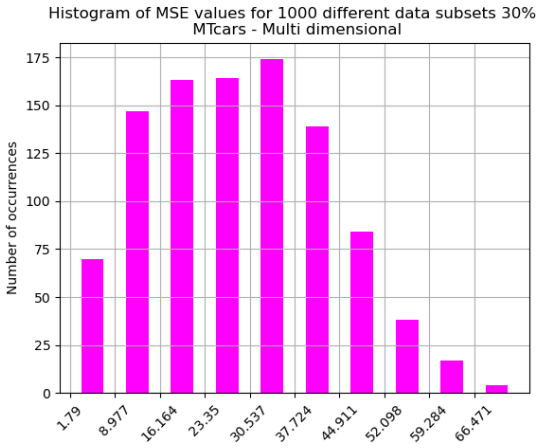


Figure 20: Mean square error - 1000 x 30% Turkish Dataset - Multidimensional

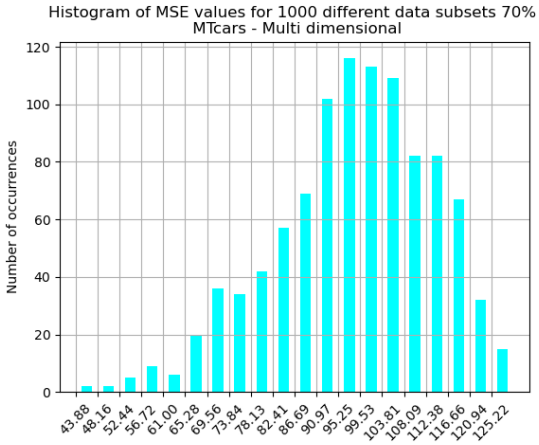


Figure 21: Mean square error - 1000 x 70% Motor Trends Dataset - Multidimensional

One-dimensional function

The One-dimensional Regression function that has been implemented is characterized by four functions:

- To compute the least square solution without intercept:

```
for every rows of the dataset:
{
    compute the numerator;
    compute the denominator;
}
compute w;
return w;
```

- To compute the least square solution with intercept:

```
for every rows of the dataset:
{
    compute the x sum;
    compute the t sum;
}
compute x mean;
compute t mean;

for every rows of the dataset:
{
    compute w1 numerator;
    compute w1 denominator;
}
compute w1;
compute w0;
return w0, w1;
```

- To compute the mean square error without intercept:

```
for every rows of the dataset:
{
    compute y;
    compute j numerator;
}
compute j;
return j;
```

- To compute the mean square error with intercept:

```
for every rows of the dataset:
{
    compute y;
    compute j numerator;
}
compute j;
return j;
```

Multidimensional function

The Multidimensional Regression function that has been implemented is characterized by two functions:

- To compute the least square solution matrix:

```
compute pseudoinverse
          of x-matrix;
compute w-matrix;

return w-matrix;
```

- To compute the mean square error:

```
compute the first member;
compute the second member;
compute the third member;

compute j;
return j;
```

Conclusion

Linear regression modeling techniques have significant limitations, as can easily be inferred from graphs showing fitted linear models. If the functional relationship that exists between the dataset elements and the target is not linear and the points in the data space are sparse, the fitted linear model will certainly lack accuracy. There is also a big difference if we randomize the subsets and if we do not: in case of non random subset there are less difference between two closed subset instead of two random. Because data is collected over a period of time (a multidimensional time series), data collected during similar periods (non-random subsets) may be more similar.
As for histograms, looking at them instead gives us another important information. The variance of the mean squared error values associated with the test set is significantly higher than that evaluated for the training set elements. The reason for this is simply that the linear model is fit to the training set and thus we aim to minimize the corresponding "mean squared error objective function". As a result, the mean squared error value associated with the training set is the smallest possible.
On the other hand, the regression model weights for the elements of the test set are not optimized, so the associated mean squared error values can be very high and different from each other. Hence higher variance.
This phenomenon can be amplified by reducing the proportion of elements in the original data set used as the training set. This is because the model is derived from a data set that is not at all representative of the test set.
The low number of data for "Motor Trends" dataset creates a limitation not only for the computation of the mean square errors but also for the prediction of mpg value, so the predicted is not so close to the original one.

References

[1] S. Rovetta, *Machine Learning for Robotics I notes and slides*, Genova University, Master Course in Robotics Engineering

Implementation of the kNN Classifier in Python

Veronica Gavagna, *Student*, UNIGE

November 2022

Abstract

This report shows an implementation of the k Nearest Neighbors classifier in Python language.
As the name suggests, the purpose of this algorithm is to classify the elements of a given test data set, assuming the class is known in advance. Specifically, this task is solved by processing classes of k observations from the training set. These classes are characterized by the smallest Euclidean distance to the considered element of the test set.
In this paper, two different study cases are considered, even though the database from which the training and test sets were extracted is the same. The first classification problem is to determine which of the 10 classes the test item belongs to. The second method instead asks the classifier to recognize whether a test observation belongs to a particular class, for different values of k. So the class is binary: one is called "belonging" and the other is called "non-belonging" or "other".
An addendum final part is about computing the confusion matrix and the classification quality indexes over 10 random subsets made of 10% of data of the training set.

Introduction

The k Nearest Neighbors Classifier (henceforth kNNC) is commonly used to solve supervised problems with categorical targets (or labels). This means that each observation in the training set is associated with a label that identifies the class to which it belongs.
The learning machine in question is a nonparametric classifier. In other words, instead of using parameters to implement the model, you build the discriminant rules directly from the data. Also known as nonparametric models, this type of model has complexity (such as model size and set of parameters) that is not predefined but depends on the data. The working principle of kNNC is based on the majority approach. It consists of classifying each observation in the test set by considering the most frequently assigned class among a fixed number of voters drawn from the training set. For the classifier in question, k voters are selected from the elements of the training set. This choice is made considering the Euclidean distance between the processed test observation $\bar{x}$ and the training set elements x^i (where i = 1,...,n obs).

$$d(\bar{x}, x^i) = ||\bar{x} - x^i|| = \sqrt{\sum_{l=1}^{m} (\bar{x}_l - x_l^i)^2}$$

Where *m* stands for the number of observation attribute.
The k training observations characterized by the smallest distance to the considered element of the test set are extracted and the relevant class is considered. The classifier output is the most common element (mode) among these classes. Like any learning machine, kNNC has strengths and weaknesses. On the plus side, "learning" the classifier in question simply means storing the training set, so the "learning" complexity is very low. Moreover, for datasets with an almost infinite number of training patterns, it guarantees that the error rate is no more than twice the Bayesian error rate. As for the drawbacks, firstly kNNC can "overfit" and thus solve the problem presented by the training set in a non-generalized way. Second, the complexity of classification increases as the number of Euclidean distances calculated increases. This phenomenon is consistent with what was previously demonstrated for nonparametric models. The data used to train and test the above algorithm comes from the MNIST database containing 28x28 pixel images of handwritten grayscale digits. This database is also already organized into a training set of 60000 samples and a test set of 10000 samples.
As previously mentioned, this experiment examines two different classification problems. Both are checked using the same database. The only difference is that in the second study case, the target class of training patterns is manipulated to obtain only two labels. Because the second problem is to determine whether a particular test observation belongs to one of the 10 original labels. So all her 9 classes in question are mapped to a common helper class.
Finally, for the addendum part, confusion matrix and the classification quality indexes are computed considering 10 random subsets made of 10% of data of the training set.
A confusion matrix is an (NxN) matrix used to evaluate the performance of a classification model. where N is the number of target classes. The matrix compares the actual target value with the value predicted by the machine learning model.

The four sections are divided as follows:

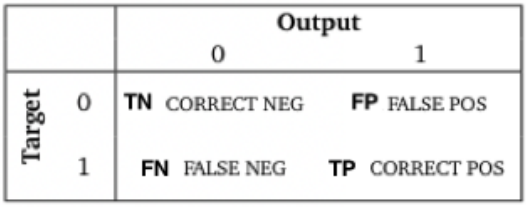


Figure 22: Confusion matrix

- True positive (TP): if the actual value is positive and the predicted value is also positive.
- True Negative (TN): if the actual value is negative and the forecast is also negative.
- False Positives (FP): the actual is negative, but the forecast is positive.
- False Negatives (FN): if the actual value is positive but the forecast is negative.

For binary classification problems, as in our case, we get a 2x2 matrix. The target variable has two values: positive or negative. The columns represent the actual values of the target variable and the rows represent the predicted values of the target variable. So, a good model has a high percentage of TP and TN and a low percentage of FP and FN.
A confusion matrix is a tabular summary of the number of correct and incorrect predictions made by a classifier. Used to measure the performance of classification models. It can be used to evaluate the performance of classification models by calculating performance metrics such as accuracy, precision, recall, and F1 score.

Accuracy just measures how often the classifier makes a correct prediction. This is the ratio of the number of correct predictions to the total number of predictions. This is a measure of correctness achieved if the prediction is true. Simply put, it shows how many of the overall positive predictions are actually positive.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision is a measure of accuracy achieved if the prediction is true. Simply put, it shows how many of the overall positive predictions are actually positive. Precision is defined as the ratio of the total number of correctly classified positive classes divided by the total number of predicted positive classes.

$$Precision = \frac{TP}{TP + FN}$$

Recall is a measure of the actual observed value correctly predicted. H. Number of observations in the positive class predicted to actually be positive. Also called sensitivity. Recall is a useful metric when you want to get as many positive results as possible. Recall is defined as the ratio of the total number of correctly classified positive classes divided by the total number of positive classes.

$$Recall = \frac{TP}{TP + FP}$$

The F1 score is a number between 0 and 1 and is the harmonic mean of precision and recall. The harmonic mean is used because, unlike the simple mean, it is not affected by extremely large values. The F1 score effectively maintains a balance between classifier precision and recall. A low precision results in a low F1, and a low recall results in a low F1 score.

$$F1 = 2\frac{Precision * Recall}{Precision + Recall}$$

Then, over the 10 subsets, the average value of each indexes is computed, which indicates the typical value, and the standard deviation is calculated, to show a measure of spread.

Implementation

The classifier k Nearest Neighbor is implemented as a function in the Python language. A main function is created and applied to the above data set. However, it is important to note that the kNNC function can handle any type of supervised training dataset. However, sometimes the data needs to be processed first in order to pass it to the function in the correct format.

Function Main

The main function performs four main tasks:

- Data preprocessing:
  The first task is to get the data from the "mnist.load_data" function included in the "tensorflow.keras.datasets" library.
  The output of this function is the training and test datasets split into images and labels. To use it, you need to convert the image size (28x28) to a unique array of 784 elements. An array is obtained by simply lining up 28 rows of pixels in the image. Pixels are represented by numbers ranging from 0 (black pixels) to 1 (white pixels) and determine the shade of gray. Next, combine the images and labels to get unique matrices for the training and test sets.
- kNN classifier testing and accuracy evaluation:
  This second task is simply to classify patterns in the test set. To do this, the kNNC function is called. This function passes as parameters a training set of images, a corresponding training set of labels, a test set of images, a value for k, and optionally a corresponding test set of original labels. Note that you do not need to provide the test purpose column as an input.
  The kNN classifier function is invoked 11 times

		Output	
		0	1
Target	0	**TN** CORRECT NEG	**FP** FALSE POS
	1	**FN** FALSE NEG	**TP** CORRECT POS

Figure 23: Example of an element of the test set and corresponding target class

one for each k values and the following set of 11 numbers is considered: [1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50].
After the output class and error rate are returned from the function, 10 items from the test set are displayed as examples of classification. Each diagram also shows the corresponding target and output classes. An example is shown below:

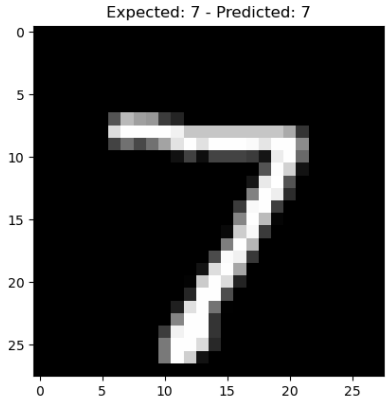


Figure 24: Example of an element of the test set and corresponding original and corresponding target class

- kNN classifier binary testing and accuracy evaluation: The purpose of this task is to assess the accuracy associated with individual classes with different numbers of nearest neighbors. To achieve this, the same procedure is used and the same set of 11 numbers ranging from 1 to 50 are considered. The only difference is that the training and test sets are modified to be binary. One class is selected and all other classes are converted to '-1'. After this process runs, the kNN classification function is called 11 times, once for every k. Accuracy for an individual class is obtained by checking whether elements in the test set belong to that class. This is only possible if the test set labels are present, otherwise it is not possible to tell whether the predictions are correct. Exist or not. The required accuracy is defined as:

$$accuracy = 1 - error\_rate$$

And the error rate is computed as:

$$error\_rate = \frac{n.correct}{n.predicted}$$

- kNN classifier evaluation: At the end, to summarize, a table for each label is made by indicates the average value of each indexes computed over the 10 subsets of the 10% of the training set, which indicates the typical value, and the standard deviation between brackets, to show a measure of spread.

kNN Classifier Function

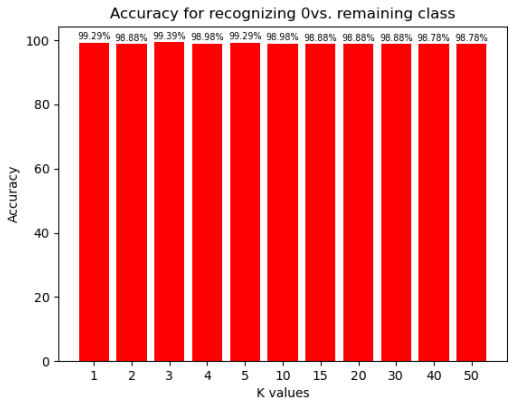
The implemented function k Nearest Neighbors Classifier is characterized by five inputs (the last being optional) and two outputs. The input are: training set of images, corresponding labels, k-values, test set images, corresponding labels (optional). Instead, the output is the predicted label class with respect to the test set and error rate. As mention before, this last variable will be correctly evaluated by the function only if the test target column exists. As for the body of the function, some debug checks are done first. Then, the following tasks are then performed for each pattern in the test set:

- Euclidean distances between the element of interest in the test set and all elements in the training set are computed
- k training observations characterized by minimum distances and respective indices are obtained
- k labels corresponding to the extracted training patterns are obtained
- The most popular class among the k selected labels is extracted

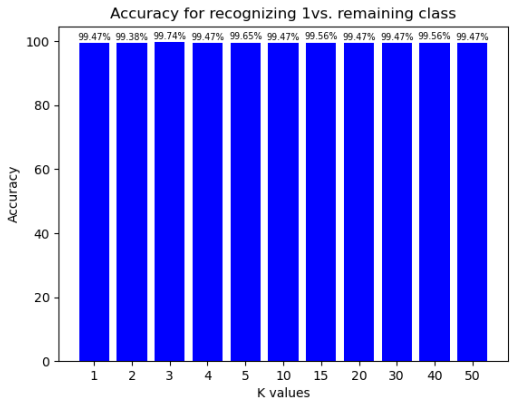
Finally, given the original labels, the output class is compared to the target class to determine the error rate.

Results

The 11 graph for the accuracy computed for the third class are shown below:

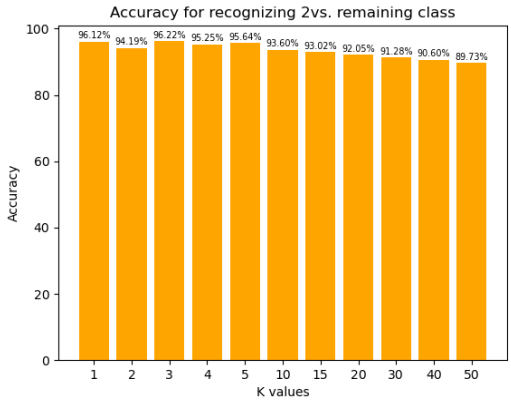


(a) Accuracy of 0 vs. other class

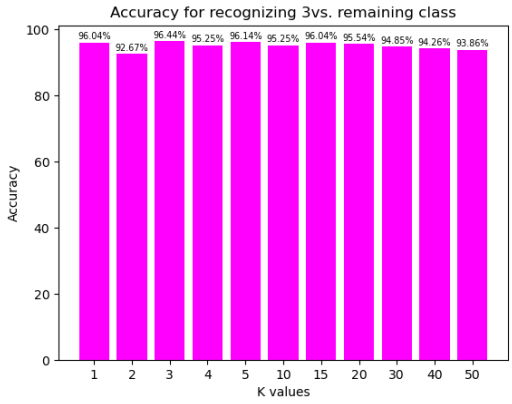


(b) Accuracy of 0 vs. other class

Figure 25: Binary accuracy graphs

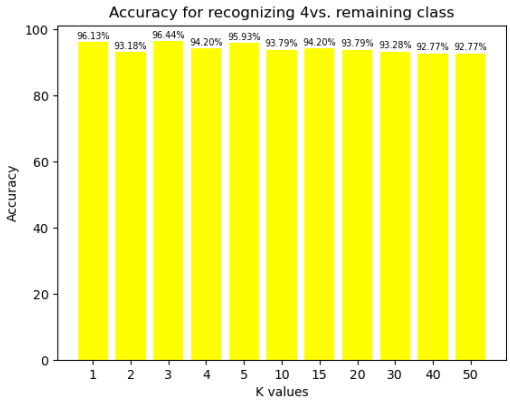


(a) Accuracy of 2 vs. other class

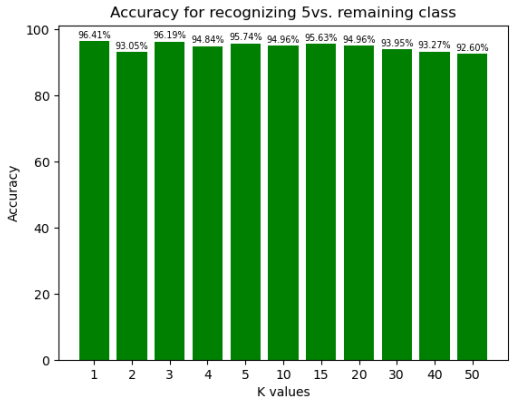


(b) Accuracy of 3 vs. other class

Figure 26: Binary accuracy graphs

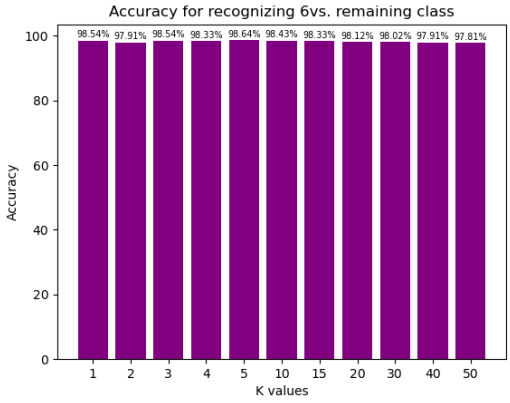


(a) Accuracy of 4 vs. other class

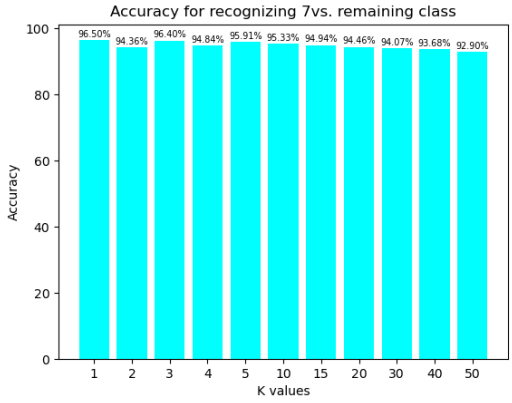


(b) Accuracy of 5 vs. other class

Figure 27: Binary accuracy graphs

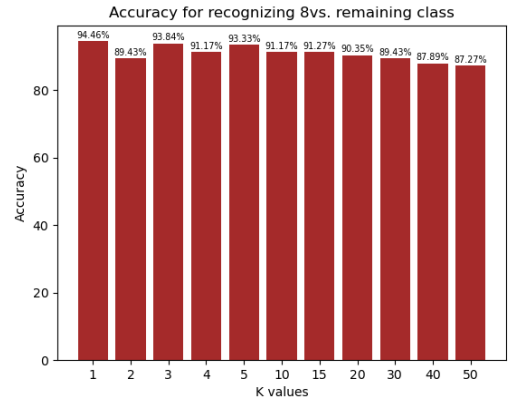


(a) Accuracy of 6 vs. other class

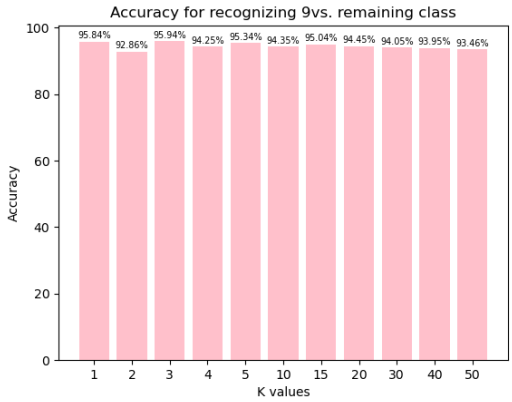


(b) Accuracy of 7 vs. other class

Figure 28: Binary accuracy graphs



(a) Accuracy of 8 vs. other class



(b) Accuracy of 9 vs. other class

Figure 29: Binary accuracy graphs

Due to the large amount of data, it is not possible to show the confusion matrices because the total number of tables will be 110 (10 classes x 11 k values).
The tables that summarize the typical values and the measure of spread of every indexes are reported below:

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0971 (0.0003)	0.995 (0.000447)	0.9557 (0.004451)	0.9875 (0.002729)	0.9713 (0.016428)
2	0.0961 (0.0003)	0.9976 (0.00049)	0.9776 (0.002577)	0.9775 (0.004006)	0.9773 (0.004011)
3	0.097 (0.0)	0.9953 (0.000458)	0.9588 (0.002786)	0.9869 (0.0013)	0.9726 (0.014359)
4	0.0966 (0.00049)	0.9971 (0.0003)	0.9722 (0.002786)	0.9821 (0.002982)	0.9771 (0.005822)
5	0.0971 (0.0003)	0.9953 (0.000458)	0.9592 (0.004308)	0.986 (0.002049)	0.9723 (0.013852)
10	0.0966 (0.00049)	0.9964 (0.00049)	0.9676 (0.004005)	0.9805 (0.002579)	0.9737 (0.007273)
15	0.0968 (0.0004)	0.9955 (0.0005)	0.9584 (0.003353)	0.9804 (0.003137)	0.9692 (0.011631)
20	0.096 (0.000447)	0.9959 (0.000539)	0.9634 (0.003382)	0.973 (0.004919)	0.9679 (0.007086)
30	0.0954 (0.00049)	0.9961 (0.0003)	0.9636 (0.002905)	0.9676 (0.003382)	0.9654 (0.004035)
40	0.0949 (0.0003)	0.9959 (0.000539)	0.962 (0.003924)	0.9619 (0.003646)	0.9618 (0.003647)
50	0.0945 (0.0005)	0.9958 (0.0006)	0.9619 (0.003833)	0.9558 (0.003655)	0.9585 (0.004544)

Table 2: Class 0 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.114 (0.0)	0.9909 (0.000943)	0.9344 (0.006151)	0.9953 (0.0009)	0.9639 (0.031413)
2	0.1133 (0.000458)	0.9936 (0.00049)	0.952 (0.004382)	0.9926 (0.001685)	0.9719 (0.020768)
3	0.114 (0.0)	0.9896 (0.00102)	0.9242 (0.00551)	0.9952 (0.000748)	0.9584 (0.036808)
4	0.114 (0.0)	0.991 (0.000632)	0.9334 (0.005678)	0.994 (0.001183)	0.9629 (0.031122)
5	0.114 (0.0)	0.9878 (0.001166)	0.9127 (0.007142)	0.9952 (0.00098)	0.9522 (0.043011)
10	0.114 (0.0)	0.9862 (0.001077)	0.9018 (0.006882)	0.9942 (0.000748)	0.9458 (0.048406)
15	0.115 (0.0)	0.9813 (0.001345)	0.8723 (0.007963)	0.9952 (0.0006)	0.9296 (0.065603)
20	0.115 (0.0)	0.98 (0.001183)	0.8654 (0.00656)	0.9948 (0.0004)	0.9256 (0.069201)
30	0.1152 (0.0004)	0.9759 (0.0013)	0.8414 (0.007102)	0.9952 (0.0004)	0.9118 (0.083401)
40	0.116 (0.0)	0.9727 (0.001487)	0.8236 (0.007499)	0.9953 (0.000458)	0.9012 (0.094101)
50	0.116 (0.0)	0.97 (0.001414)	0.8092 (0.007068)	0.9958 (0.0004)	0.8929 (0.102901)

Table 3: Class 1 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0956 (0.00049)	0.9915 (0.001204)	0.9243 (0.009456)	0.9351 (0.004636)	0.9294 (0.007347)
2	0.0903 (0.0009)	0.9976 (0.00049)	0.9762 (0.003789)	0.8821 (0.008006)	0.9268 (0.045411)
3	0.0957 (0.000458)	0.994 (0.000632)	0.9465 (0.003828)	0.9367 (0.005178)	0.9415 (0.00706)
4	0.0924 (0.000663)	0.997 (0.000447)	0.9705 (0.0035)	0.9045 (0.007658)	0.9363 (0.032709)
5	0.0952 (0.0006)	0.9948 (0.000748)	0.952 (0.004583)	0.9328 (0.00551)	0.9422 (0.010896)
10	0.0929 (0.000539)	0.9967 (0.000458)	0.9683 (0.003195)	0.9089 (0.00589)	0.9377 (0.029396)
15	0.0935 (0.000671)	0.9955 (0.0005)	0.9594 (0.003555)	0.9147 (0.006885)	0.9365 (0.022862)
20	0.0923 (0.00064)	0.9962 (0.0006)	0.9638 (0.004331)	0.9006 (0.00771)	0.9311 (0.031459)
30	0.0912 (0.0006)	0.9959 (0.0007)	0.9622 (0.005845)	0.8893 (0.006451)	0.9242 (0.035491)
40	0.0899 (0.000943)	0.9959 (0.000539)	0.9612 (0.004285)	0.8742 (0.009336)	0.9158 (0.042635)
50	0.0886 (0.000917)	0.9962 (0.0006)	0.9618 (0.004423)	0.8615 (0.008453)	0.9087 (0.047951)

Table 4: Class 2 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.096 (0.000447)	0.9972 (0.0006)	0.9732 (0.004094)	0.919 (0.00395)	0.9453 (0.026595)
2	0.0903 (0.000458)	0.999 (0.0)	0.9909 (0.001446)	0.8618 (0.005075)	0.9218 (0.060214)
3	0.0944 (0.00049)	0.9981 (0.000539)	0.9815 (0.003956)	0.9051 (0.004437)	0.9418 (0.036967)
4	0.0911 (0.000831)	0.9989 (0.0003)	0.9902 (0.003092)	0.8697 (0.009045)	0.9261 (0.057121)
5	0.0929 (0.0007)	0.9985 (0.0005)	0.9859 (0.00333)	0.8896 (0.007378)	0.9353 (0.046292)
10	0.0887 (0.000458)	0.999 (0.0)	0.9917 (0.001487)	0.8441 (0.003015)	0.9119 (0.067867)
15	0.0874 (0.0008)	0.999 (0.0)	0.9912 (0.001077)	0.8321 (0.00673)	0.9048 (0.073011)
20	0.0845 (0.000922)	0.9991 (0.0003)	0.9926 (0.001497)	0.8018 (0.008459)	0.8871 (0.085718)
30	0.0814 (0.0008)	0.9992 (0.0004)	0.9933 (0.001187)	0.7677 (0.008989)	0.8659 (0.098611)
40	0.0783 (0.001005)	0.9994 (0.00049)	0.9939 (0.001513)	0.7369 (0.009884)	0.8461 (0.109646)
50	0.0755 (0.001285)	0.9995 (0.0005)	0.9943 (0.001676)	0.7097 (0.011127)	0.8281 (0.118922)

Table 5: Class 3 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0913 (0.000781)	0.9951 (0.000539)	0.953 (0.005477)	0.9201 (0.007778)	0.9362 (0.01788)
2	0.0853 (0.001269)	0.9986 (0.00049)	0.9844 (0.003583)	0.8537 (0.012378)	0.9144 (0.061949)
3	0.0908 (0.000748)	0.9966 (0.00049)	0.9671 (0.003448)	0.9151 (0.008142)	0.9403 (0.026483)
4	0.0871 (0.000943)	0.9985 (0.0005)	0.9834 (0.003353)	0.875 (0.009602)	0.9258 (0.0517)
5	0.0903 (0.0009)	0.9971 (0.000539)	0.9715 (0.003722)	0.9081 (0.009731)	0.9387 (0.03211)
10	0.0874 (0.000663)	0.9983 (0.000458)	0.9837 (0.002795)	0.8775 (0.008823)	0.9276 (0.050871)
15	0.0881 (0.000943)	0.9982 (0.0004)	0.9813 (0.003848)	0.8833 (0.009034)	0.9296 (0.047173)
20	0.0853 (0.0009)	0.9986 (0.00049)	0.9844 (0.00372)	0.8561 (0.008526)	0.9158 (0.060306)
30	0.0837 (0.000781)	0.9988 (0.0004)	0.986 (0.003715)	0.8375 (0.007736)	0.9059 (0.068836)
40	0.0818 (0.000872)	0.9988 (0.0004)	0.9877 (0.003848)	0.8188 (0.009516)	0.8952 (0.07699)
50	0.0806 (0.000917)	0.9988 (0.0004)	0.9872 (0.004285)	0.8025 (0.00977)	0.8853 (0.083374)

Table 6: Class 4 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0838 (0.0006)	0.9927 (0.0009)	0.9258 (0.006735)	0.9266 (0.005161)	0.9259 (0.005209)
2	0.0775 (0.0005)	0.998 (0.0)	0.9783 (0.002238)	0.8566 (0.006344)	0.9135 (0.057253)
3	0.0832 (0.000748)	0.9952 (0.0006)	0.9495 (0.006469)	0.9229 (0.008031)	0.936 (0.015366)
4	0.0795 (0.000671)	0.9977 (0.000458)	0.9743 (0.003743)	0.8809 (0.006379)	0.9253 (0.044856)
5	0.0822 (0.000748)	0.9956 (0.000663)	0.9546 (0.005122)	0.9125 (0.006607)	0.9332 (0.021729)
10	0.0787 (0.000781)	0.9975 (0.0005)	0.9728 (0.00402)	0.872 (0.00827)	0.9197 (0.048412)
15	0.0791 (0.000831)	0.9971 (0.0003)	0.9669 (0.003534)	0.874 (0.00911)	0.9181 (0.045031)
20	0.076 (0.000775)	0.9977 (0.000458)	0.9742 (0.005896)	0.8386 (0.008464)	0.9012 (0.06317)
30	0.0734 (0.000917)	0.998 (0.000775)	0.9763 (0.007824)	0.8074 (0.011482)	0.8839 (0.077357)
40	0.0709 (0.001044)	0.998 (0.000775)	0.9767 (0.007457)	0.7772 (0.010581)	0.8655 (0.088932)
50	0.0679 (0.001221)	0.9983 (0.00064)	0.977 (0.008198)	0.7424 (0.013662)	0.8435 (0.102019)

Table 7: Class 5 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0939 (0.000539)	0.9956 (0.0008)	0.9597 (0.005984)	0.9734 (0.004477)	0.9663 (0.008393)
2	0.0917 (0.00064)	0.9982 (0.0004)	0.9825 (0.003828)	0.9507 (0.005967)	0.9664 (0.016796)
3	0.0939 (0.0003)	0.9958 (0.0004)	0.9612 (0.003945)	0.9736 (0.002764)	0.9672 (0.006971)
4	0.0928 (0.0004)	0.9974 (0.00049)	0.9742 (0.0034)	0.9615 (0.003956)	0.9676 (0.00727)
5	0.0939 (0.0003)	0.996 (0.000632)	0.9631 (0.00446)	0.9724 (0.004104)	0.9675 (0.006391)
10	0.0929 (0.0003)	0.9969 (0.0003)	0.9695 (0.001688)	0.9633 (0.00492)	0.9662 (0.005711)
15	0.093 (0.0)	0.996 (0.000447)	0.9634 (0.003353)	0.9639 (0.003048)	0.9634 (0.003089)
20	0.0925 (0.0005)	0.9965 (0.0005)	0.9666 (0.002538)	0.9576 (0.00403)	0.9617 (0.005749)
30	0.0918 (0.0004)	0.9963 (0.000458)	0.9659 (0.001814)	0.9527 (0.005442)	0.9591 (0.008401)
40	0.0914 (0.000663)	0.9965 (0.0005)	0.9662 (0.003709)	0.9467 (0.006294)	0.9562 (0.011396)
50	0.0908 (0.0006)	0.9966 (0.00049)	0.9668 (0.003572)	0.9406 (0.006422)	0.9534 (0.014321)

Table 8: Class 6 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0978 (0.0004)	0.9917 (0.000781)	0.9275 (0.0055)	0.939 (0.005254)	0.9331 (0.0079)
2	0.094 (0.000894)	0.996 (0.000447)	0.963 (0.004879)	0.9019 (0.007981)	0.9316 (0.030754)
3	0.0973 (0.00064)	0.9931 (0.0007)	0.9379 (0.006188)	0.933 (0.007057)	0.9352 (0.007392)
4	0.0949 (0.0007)	0.9952 (0.0004)	0.9565 (0.003233)	0.9104 (0.007255)	0.9328 (0.023546)
5	0.0965 (0.0005)	0.9933 (0.00064)	0.9406 (0.004363)	0.9266 (0.004454)	0.9334 (0.008129)
10	0.0945 (0.0005)	0.9944 (0.000663)	0.9484 (0.003955)	0.9041 (0.004847)	0.9257 (0.022137)
15	0.0945 (0.000806)	0.9935 (0.0005)	0.9408 (0.0044)	0.9046 (0.007255)	0.9224 (0.019222)
20	0.0931 (0.0007)	0.9941 (0.000539)	0.9454 (0.0042)	0.89 (0.006229)	0.9168 (0.027514)
30	0.0919 (0.0007)	0.9942 (0.0004)	0.9456 (0.004104)	0.8779 (0.006833)	0.9105 (0.033308)
40	0.0906 (0.000663)	0.9942 (0.000748)	0.9453 (0.006001)	0.8668 (0.008376)	0.9042 (0.038326)
50	0.0899 (0.0007)	0.9945 (0.0005)	0.9456 (0.004821)	0.8567 (0.006404)	0.899 (0.042782)

Table 9: Class 7 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.087 (0.001)	0.9963 (0.00064)	0.9612 (0.004285)	0.88 (0.00999)	0.919 (0.040259)
2	0.0786 (0.001281)	0.999 (0.000447)	0.9884 (0.00398)	0.7899 (0.013412)	0.8779 (0.089016)
3	0.086 (0.001095)	0.9977 (0.00064)	0.9753 (0.005311)	0.8705 (0.011952)	0.9198 (0.050728)
4	0.0806 (0.001356)	0.9989 (0.0003)	0.9861 (0.003477)	0.8114 (0.012627)	0.8903 (0.079904)
5	0.0844 (0.001428)	0.9981 (0.0003)	0.9785 (0.003413)	0.8523 (0.014471)	0.911 (0.060457)
10	0.0797 (0.001187)	0.9985 (0.0005)	0.9839 (0.003208)	0.8012 (0.013445)	0.8832 (0.083095)
15	0.0799 (0.0013)	0.9983 (0.000458)	0.9812 (0.003187)	0.8032 (0.013526)	0.8833 (0.081234)
20	0.0769 (0.001044)	0.9984 (0.00049)	0.9824 (0.003262)	0.7714 (0.012085)	0.8641 (0.093484)
30	0.0745 (0.0015)	0.9984 (0.00049)	0.9813 (0.003035)	0.7449 (0.015852)	0.8468 (0.103126)
40	0.0722 (0.001327)	0.9982 (0.0004)	0.9774 (0.003929)	0.7199 (0.01455)	0.829 (0.110066)
50	0.0699 (0.0013)	0.9982 (0.0004)	0.9758 (0.004094)	0.6966 (0.014637)	0.8128 (0.117118)

Table 10: Class 8 - Evaluation table

k	Average_Sensitivity	Average_Specificity	Average_Precision	Average_Recall	Average_F1
1	0.0952 (0.000748)	0.9885 (0.000806)	0.9008 (0.005582)	0.9275 (0.00727)	0.9139 (0.015421)
2	0.0889 (0.001136)	0.9953 (0.000781)	0.9532 (0.007427)	0.8636 (0.010828)	0.906 (0.043761)
3	0.095 (0.000447)	0.9904 (0.001114)	0.9142 (0.006969)	0.9269 (0.004253)	0.9204 (0.007768)
4	0.0919 (0.000831)	0.9942 (0.0006)	0.9451 (0.004989)	0.8956 (0.007552)	0.9198 (0.025351)
5	0.0949 (0.0007)	0.9902 (0.000748)	0.9141 (0.005682)	0.9255 (0.00706)	0.9195 (0.009266)
10	0.0929 (0.000539)	0.9923 (0.000781)	0.9299 (0.005522)	0.9059 (0.00661)	0.9177 (0.013525)
15	0.094 (0.000632)	0.9896 (0.00102)	0.9083 (0.007537)	0.9152 (0.006369)	0.9115 (0.007365)
20	0.0924 (0.000917)	0.9905 (0.001285)	0.9137 (0.010517)	0.8991 (0.007368)	0.9062 (0.010232)
30	0.0918 (0.000872)	0.9899 (0.0013)	0.9078 (0.009152)	0.8898 (0.008704)	0.8987 (0.012449)
40	0.0908 (0.000748)	0.9892 (0.001249)	0.9034 (0.009243)	0.8807 (0.008149)	0.8917 (0.01369)
50	0.0899 (0.001044)	0.9891 (0.001375)	0.8993 (0.008787)	0.8706 (0.010385)	0.8846 (0.017431)

Table 11: Class 9 - Evaluation table

Conclusion

As can be seen from the plot, the accuracy of each class is very high (over 90% except for those over 80%) as a function of different k values. Note that higher values of k generally tend to result in lower accuracy. The reason is that if k is too high, some of the closest observations are not so close and the output is also affected by them. So some correct answers may be missed. On the other hand, some of the closest observations are assigned to the wrong class, so there are some small values of k that don't give good accuracy, and the result is also wrong. Another thing you can see from the graph is that not all values of k are suitable for all classes. This makes sense. This is because if the number of nearest neighbors is even, the label for the third task can only be -1 or the value of the class, which could result in a tie. In other words, the number of minimum distance training observations for the class label may equal the number of minimum distance training observations for the 1 label, consistent with the particular observations in the test set. This phenomenon can cause errors in choosing the correct label and lead to a loss of accuracy.
Second, notice that the slope of the line can decrease as k increases. This is simply because the more nearest neighbors considered, the less effective the nearest neighbors are in determining the final output class.
Finally, some classes are less accurate than others. This simply means that classifiers have a harder time identifying these particular labels and often confuse them with other classes.
Overall, however, the accuracy values are high, indicating that kNNC is an efficient learning machine despite all the above negative characteristics. We can also look at the tables of the average indexes and see that indexes are all very high (close to 1) and the standard deviation is very low, which indicates that there are not so many difference between a random subsets and another. This confirm that the kNN classifier implemented is efficient.

References

[1] S. Rovetta, *Machine Learning for Robotics I notes and slides*, Genova University, Master Course in Robotics Engineering

Artificial Neural Networks and Matlab nprtool Usage

Veronica Gavagna, *Student*, UNIGE

December 2022

Abstract

This report provides a brief introduction to the structure and working principles of artificial neural networks and shows how to use the MATLAB Neural Network Pattern Recognition Tool (aka "nprtool"). Artificial neural networks are special learning machines that mimic the inner workings of the human mind. It then consists of multiple interconnected computing units or neurons. This type of agent can be used to solve many machine learning problems. Now consider a classification problem. MATLAB's "nprtool" implements a two-layer artificial neural network designed to solve this kind of problem. In this paper, we show the performance of this particular network in the form of a confusion matrix by training and testing it on two different datasets. Altering the number of hidden neurons in the network is also performed to analyze the effect on network accuracy.

Introduction

An artificial neural network (ANN) is a network of interconnected units (or neurons) that simulate the internal mechanics of human brain processing. A typical structure of an ANN is shown below.

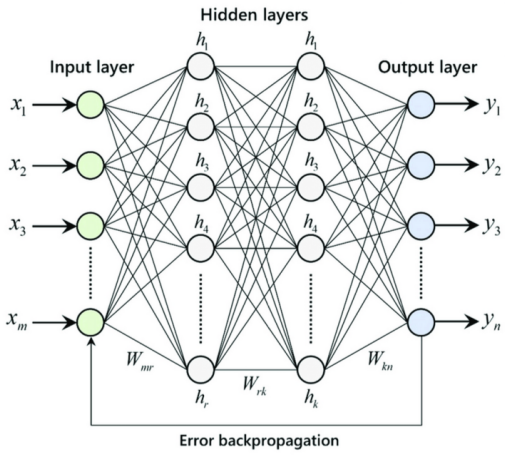


Figure 30: Typical structure of an Artificial Neural Network

As the figure shows, neurons in ANNs are typically organized in layers. The outermost layers are the input layer and the output layer. However, in most cases the former is not a real layer as it does not consist of computational units. In fact, the input layer nodes typically represent only the neural network inputs.
As a result, from left to right, the first real layer becomes his first hidden layer.
Assuming the neural network is fully connected, each unit of it receives as input all the inputs of the learning machine, as shown in the figure. That is, every input is connected to every unit in the first hidden layer. Each of these compounds has several properties summarized in a parameter called weight. The same is true for connections between units. When it comes to the simple behavior of neurons, it is basically possible to transform a single input into an output. The following diagram shows the structure of a formal neuron.

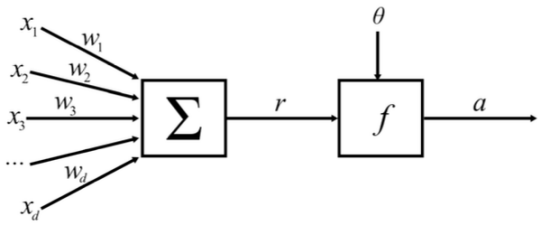


Figure 31: Typical structure of a formal neuron of an ANN

As can be seen, each arithmetic unit primarily evaluates the weighted sum of all inputs, giving a number r called the net input (or stimulus). The difference between this number and the threshold (or bias) Θ is remapped to the specified interval. This task is performed by a function f (also called the activation function) and the generated output a, called the activation value, is the output of the neuron. Here are some examples of activation functions you can use:

- Heaviside step function:
  a discontinuous function defined on
  $[-\infty, +\infty] \rightarrow [0, +1]$.

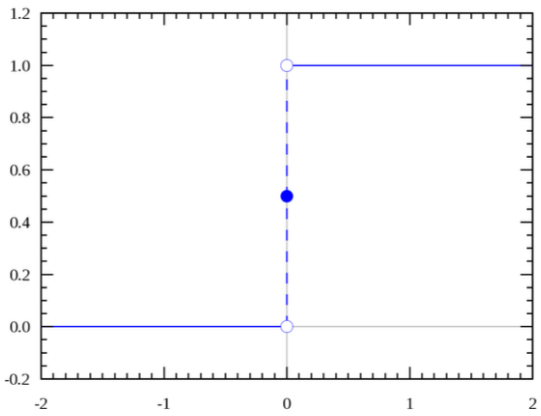


Figure 32: Heaviside step activation function

- Signum function:
  symmetrization in the interval $[1, +1]$ of the Heaviside step, therefore defined on $[-\infty, +\infty] \rightarrow [1, +1]$.

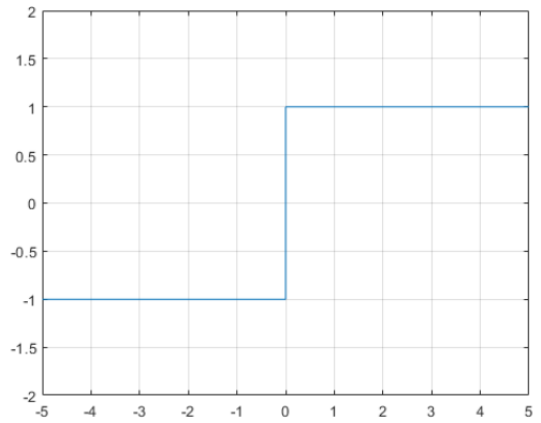


Figure 33: Signum activation function

- Sigmoid (or Logistic) function:
  continuous function defined on $[-\infty, +\infty] \rightarrow [0, +1]$.

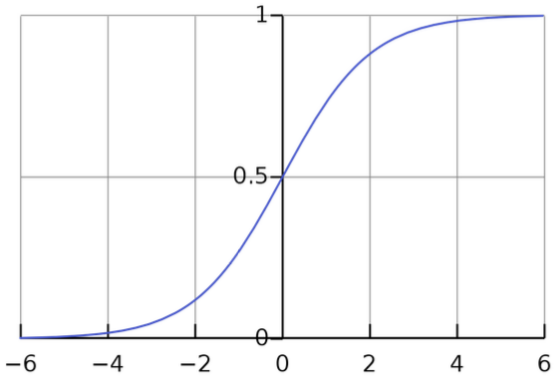


Figure 34: Sigmoid (or Logistic) activation function

- Hyperbolic tangent function:
  symmetrization in the interval $[1, +1]$ of the Sigmoid function, therefore defined on $[-\infty, +\infty] \rightarrow [1, +1]$.

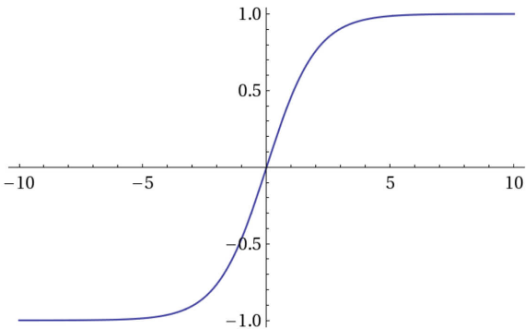


Figure 35: Hyperbolic tangent activation function

- Rectifier-Linear (usually abbreviated as ReL) function: discontinuous function defined on [-∞, +∞] $\rightarrow$[0, +∞].

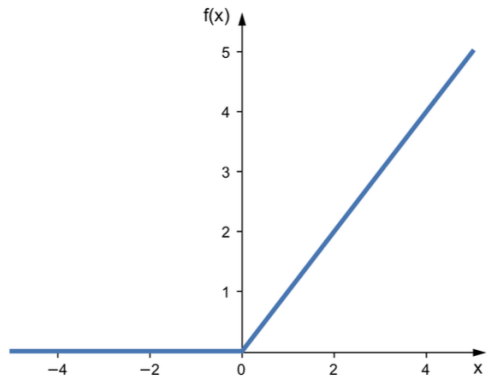


Figure 36: Rectifier-Linear activation function

- Softplus function:
  continuous function similiar to the ReL, therefore defined on [+∞, +∞] $\rightarrow$[0, +∞].

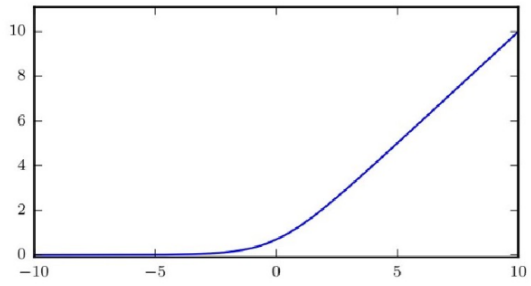


Figure 37: Softplus activation function

As mentioned above, all these activation functions take into account the net input of a single processing unit. However, it is sometimes useful for each neuron to produce an output that depends on both its stimulus and the stimuli of other units in the layer. For example, in a classification problem, simulating predicted probabilities makes the output easier to interpret. That is, real values between 0 and 1 that sum to 1. To achieve this, the stimuli of the output layer units must be processed in such a way that they are all considered. This is the idea behind the multi-neuron activation function. Among them, the two most common are:

- Max (or Argmax) function:
  A function that considers the neuron's net input and compares it to the stimuli of all other units in the layer: Returns 1 if the net inputs considered are their maximum value. Returns 0 otherwise.
- Softmax function:
  A function that, given a neuron's net input, evaluates the corresponding predicted probability given the stimuli of all other units in the layer.

The first is typically used in classification problems when testing already trained neural networks. This is because the argmax function makes the output very easy to interpret, but on the other hand he cannot customize the ANN.
Conversely, the Softmax function can also be used for training, but the output is a bit confusing.
When it comes to training and testing neural networks for classification problems, we can summarize:

- Training phase:
  During the training phase of the artificial neural network, we change the connection weights and unit biases to minimize a certain average cost function. Since the average cost function is computed over the entire training set, parameter tuning is repeated at each epoch (an epoch consists of feeding all elements of the training set individually to the ANN).
  Of course, it is assumed that the issues under consideration have been dealt with. Otherwise, the cost function cannot be evaluated if no targets are specified in the training set. This is because it is basically calculated by comparing your spending to your goals.
  Even though the cost function always takes all the parameters of the ANN as input and results in a single number, it can still distinguish between different types of parameters. Two of the most common choices are the mean squared error cost function and the cross entropy cost function. The latter is used more often, especially in classification problems. This is to determine the larger adjustment of weights and biases.
  The technique used to perform parameter fitting is called (stochastic) gradient descent. As the name suggests, it uses the gradient of the cost function to minimize it. In particular, at each epoch the gradient of the cost function is evaluated and the parameters are changed by metaphorically stepping in opposite directions with respect to the gradient. Stepping in the opposite direction on the gradient simply means summing each weight and biasing the opposite side of the corresponding element of the gradient multiplied by a factor. The higher the factor, the wider the step. As a result, a local minimum of the cost function is reached as quickly as possible. The reason for this is that the slope points in the direction of the steepest rise, while the opposite direction is only the steepest descent to a local minimum. Finally, note that an automatic differencing algorithm known as error backpropagation is used to evaluate gradients in ANNs.

- Testing stage:
  Once the parameter set that minimizes the cost function is found, the neural network is fed with test set elements. Therefore, the goal of ANN is to accurately classify this observation. Targets associated with the test set may or may not be provided. In the first case, we can determine the error rate and accuracy of the network.

In addition to these two phases, we actually need a phase that runs in parallel with training. This is called the validation phase and aims to avoid overfitting the ANN. In other words, during the training phase, the neural network can be very efficient at classifying the observations in the training set, but it cannot generalize the data to give an accurate untrained classification. You may not be able to. ANN validation aims to avoid this scenario. To accomplish this task, a neural network is trained and validated on two separate datasets simultaneously. Validation simply classifies each input from the validation set based only on what the network has learned up to that epoch. If the results associated with the validation data are less accurate than those associated with the training data, the model may be overfitting. Therefore, some measures should be taken. For example, training may end prematurely.

After examining the main features of the structure and working principle of artificial neural networks, we will consider the MATLAB "nprtool". This tool implements a two-layer feedforward neural network for the classification problem shown below.

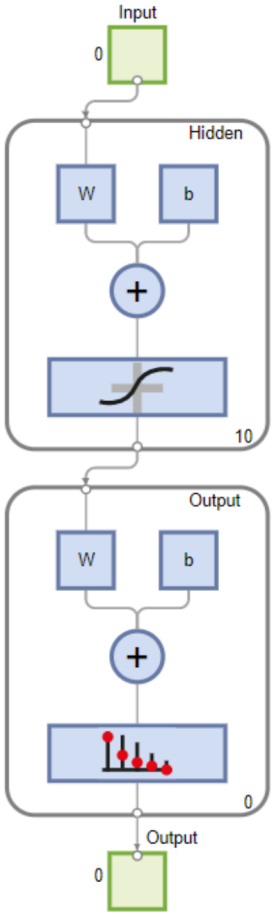


Figure 38: Structure of the ANN implemented by the Matlab nprtool

As mentioned earlier, any possible neural network has two layers: a hidden layer and an output layer.
The net input of each hidden layer neuron is processed by a sigmoidal activation function. Instead, the output is generated from the stimulation of the output unit by a softmax activation function.
If you choose a classification problem, the number of inputs equals the number of attributes for each observation. On the other hand, the number of outputs and output units is clearly equal to the number of output classes. The number of hidden units is the degree of freedom, the percentage of the data set reserved for training, validation, and test sets. Alternatively, the number of hidden layers cannot be changed.

MATLAB nprtool usage

s described on the MATLAB website associated with the Neural Network Pattern Recognition Tool, this application allows users to:

- Import data from a file or the MATLAB workspace, or use one of our sample datasets
- Split the data into training, validation, and test sets
- Define and train a neural network
- Evaluate network performance using cross-entropy and misclassification errors
- Analyze results using visualization plots, such as confusion matrices and receiver operating characteristic curves
- Generate MATLAB scripts to reproduce results and customize the training process

In this paper, two sample datasets are considered: The so-called Cancer dataset and the Glass dataset. The first identifies a classification problem that classifies cancers as benign or malignant based on specimen biopsy characteristics. Therefore, the classes are 2 (benign and malignant) and the observed attribute (biopsy feature) are 9. The number of patterns are 699.
The second dataset relates to the classification problem of classifying glasses as windows or non-windows based on glass chemical features. So the classes are 2 (windowed and non-windowed) and the observed attribute (chemical property of glass) are 9. There are 214 patterns.
These two data sets are divided into a training set, a validation set, and a test set according to their respective percentages.
70%, 15%, 15%. Then train and test a two-layer neural network three times for each data set. Use a different number of hidden units each time. In particular, a number of hidden neurons close to the number of output units are considered for the first time. A third time, we assume that the number of hidden neurons is approximately half the number of inputs. The second time, the intermediate numbers between the first and her third choice are considered.
For each case considered, the performance of the neural network on the test set is shown below in the form of a confusion matrix.

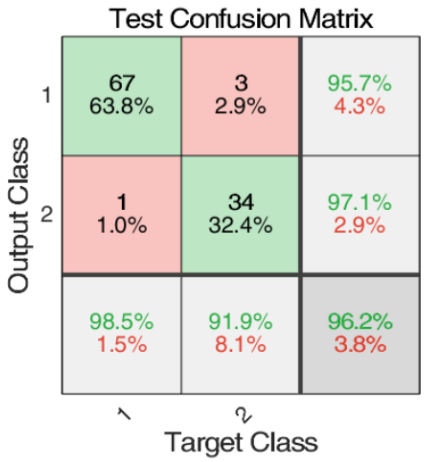


Figure 39: Two-layers NN performance on the Cancer test set - 40 Hidden units

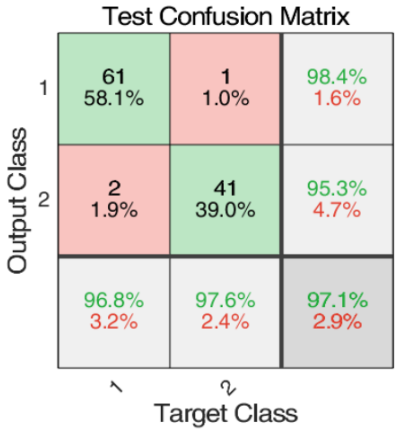


Figure 40: Two-layers NN performance on the Cancer test set - 100 Hidden units

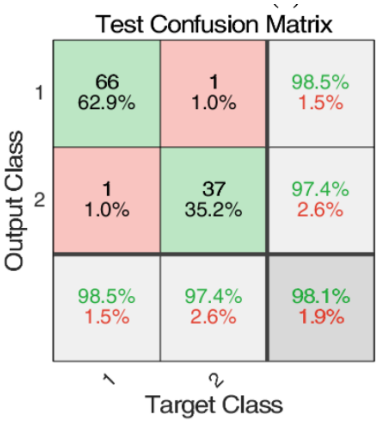


Figure 41: Two-layers NN performance on the Cancer test set - 400 Hidden units

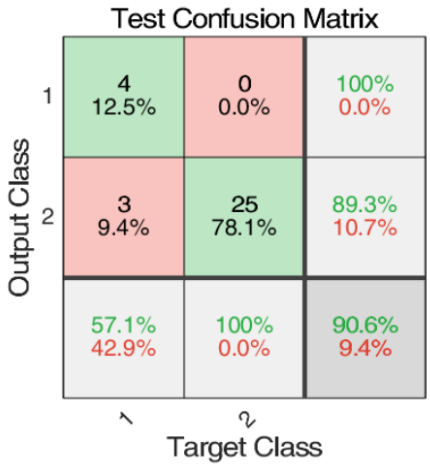


Figure 42: Two-layers NN performance on the Glass test set - 5 Hidden units

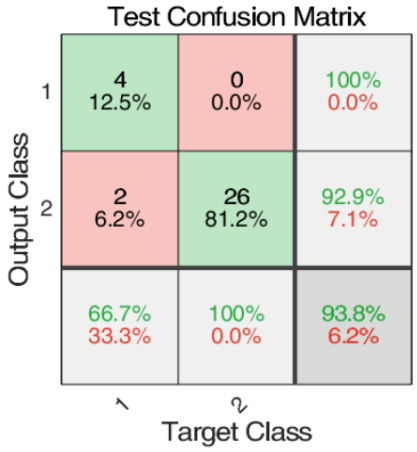


Figure 43: Two-layers NN performance on the Glass test set - 50 Hidden units

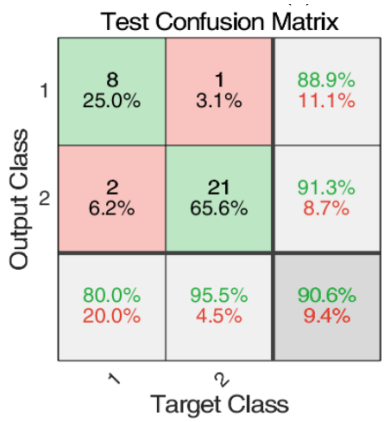


Figure 44: Two-layers NN performance on the Glass test set - 100 Hidden units

To better interpret these numbers, here are the keys: Start in the upper left corner and go row by row. The terms displayed are:

- The number of observations that were properly classified as class 1
- Number of observations incorrectly classified as class 1 and belonging to class 2
- Precision and error bounds associated with the total number of class 1 observations
- Number of observations incorrectly assigned to class 2 and belonging to class 1
- The number of observations that were properly classified as class 2
- Accuracy and error rate relative to the total number of class 2 observations
- Precision and error bars for the total number of class 1 observations
- Precision and error bars associated with the total number of class 2 observations
- Precision and error bounds for the total number of observations

Conclusion

As can be seen from the confusion matrix, the overall accuracy of the neural network is very good, especially in the first experiment as well as in the second. However, it doesn't seem to be strictly related to the number of hidden units. In the first case, the overall accuracy improves as the number of hidden neurons increases. The second precision, on the other hand, peaks according to the average number of units. It will be interesting to see how adding hidden layers affects the performance of the tester's network, but unfortunately this is not possible with his MATLAB nprtool.

References

[1] S. Rovetta, *Machine Learning for Robotics I notes and slides*, Genova University, Master Course in Robotics Engineering

[2] Matlab Tutorial: Classify Patterns with a Shallow Neural Network *https://it.mathworks.com/help/deeplearning/gs/classify-patterns-with-a-neural-network.html?lang=en*

[3] Matlab Tutorial: Shallow Networks for Pattern Recognition, Clustering and Time Series *https://it.mathworks.com/help/deeplearning/gs/shallow-networks-for-pattern-recognition-clustering-and-time-series.html?lang=en*

Training of a two-layer Autoencoder in Matlab

Veronica Gavagna, *Student*, UNIGE

December 2022

Abstract

This report shows the training of a two-layer autoencoder using MATLAB software. An autoencoder is a specific type of neural network used to solve self-supervised problems. In other words, this learning machine doesn't need a training dataset with an attached goal, because the output goal itself is the input. In this paper, we use the MNIST database, a collection of images representing handwritten digits, to extract a sample dataset. A stratified subset consisting of observations related to it 2 of the 10 classes is then extracted from the original subset. This subset is used for autoencoder training, which is performed using MATLAB built-in functions. Finally, the tests are run and the activation values of the units hidden in the network are displayed in layers with different colors depending on the corresponding class. Plotting is possible because the number of hidden neurons is set to 2.

Introduction

As mentioned earlier, an autoencoder is a specific neural network whose purpose is to reproduce the input in its output. For this reason, his training does not require a target class.
The destination is the input itself, and the number of output units is obviously the same as the number of inputs. As for the number of hidden layers, we can alternatively specify any layer, but the networks considered here are characterized by only one of them.
The following diagram shows the general structure of a two-layer autoencoder.

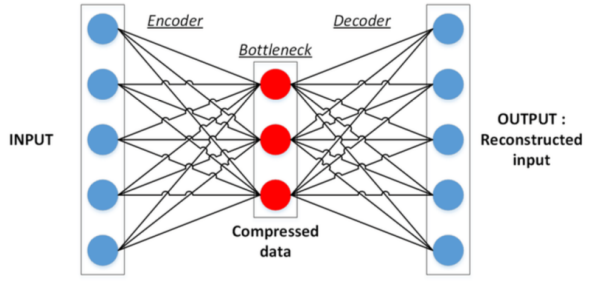


Figure 45: Typical structure of a two-layer Autoencoder

As the image shows, the network performs two main actions.
Encode the input and decode the hidden unit activation values. In other words, the input observations are first compressed into a set of activation values well below the characteristics of the input (hence the name "bottleneck"). These activation values are then decompressed to provide the output.
Regarding the training phase of the network, the concepts are exactly the same as those considered in previous reports, with the only difference that the output is compared to the input to evaluate the cost function.
In the proposed experiment, the dataset used for training the autoencoder is extracted from his MNIST database. This database contains 28x28 pixel images of handwritten grayscale digits, already organized into a training set consisting of 60000 samples and a test set consisting of 10000 samples. However, a subset of the MNIST training set is extracted here to reduce the computational load on the network. In particular, stratified subsets related to only 2 of the 10 classes are considered to maintain their proportions.

So the autoencoder in question looks like this:

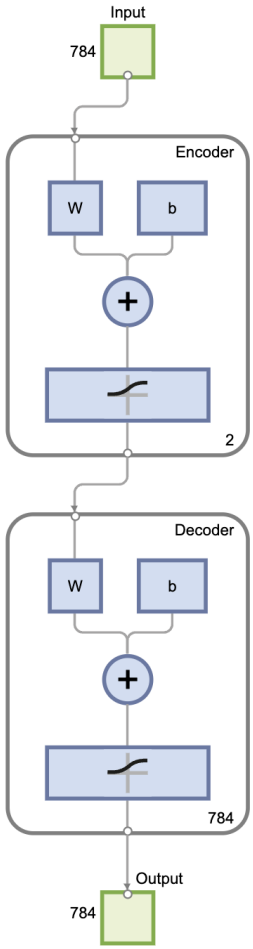


Figure 46: Structure of the considered Autoencoder

As you can see from the picture, the encoder part of the network is just a hidden layer and consists of only two neurons. This number was chosen to allow the activation value to be expressed in one level and to be better interpreted. Another thing we can observe is that both the encoder and decoder units use sigmoid (or logistic) activation functions to process the net input.

Implementation

The package attached to this assignment contains a MATLAB function and the main MATLAB script that calls this function.

Main script

The main script does some simple tasks.

- Test the autoencoder by setting up the two classes you want and making sure they are correct
- Specifies the percentage of data that should be extracted from the MNIST training set
- Call the function, passing the previously specified information as input
- Represents the hidden Autencoder unit activation value returned by the function (in different colors and shapes depending on the corresponding class)

Autoencoder function

The implemented MATLAB function is characterized by three inputs and two outputs. Here is the input:
Two targeted classes in the MNIST database against which the autoencoder will be tested, and the percentage of data extracted from the MNIST training set.
Instead the output looks like this:
A matrix containing hidden unit activation values obtained by testing the trained network and a vector of labels associated with each training observation. As for the main part of the function, some data processing is performed first. In particular, as mentioned above, after importing the data, stratified subsets containing observations associated with the two classes of interest are generated. The "loadMNIST" function is used for importing. This function takes two arguments as input. The first is 0 or 1, distinguishing whether the data was drawn from her MNIST training set or test set. The second, on the other hand, is a range of numbers chosen between 0 and 9, indicating the class associated with the extracted observations. Instead, the output of the loadMNIST function is a column vector containing the requested data set in array form and the associated target class. Note that each row of the resulting data matrix contains an MNIST image of her in the form of a 784-element vector. A vector is obtained by simply arranging 28 rows of pixels in the image. Pixels are represented by numbers ranging from 0 (black pixels) to 1 (white pixels) and determine the shade of gray.
The "loadMNIST" function is called twice here, returning 0 as the first input and the two classes of interest as the second input one after the other. A random subset is then extracted from each of his two parts of the MNIST training set. Note that the percentage of random items extracted is passed as an input from the main script. Finally, all subsets are stacked to form

a new stratified training set. Corresponding stratified subsets related to the target class are also obtained. This is not used to train the encoder, but is returned by the function and used by the main script to distinguish which class the activation value refers to.
After data processing, the above autoencoder is trained using his MATLAB built-in function "train-Autoencoder". The function is given a stratified subset of training and the number of hidden units (2 in this case). Returns a multi-field structure representing the trained network. Of course, this process takes time. Specifically, 1000 epochs are run over the entire training set. As a final step, the encode function is called, taking the trained autoencoder and the training set as input, to obtain a matrix containing the hidden unit activation values associated with each training observation. The encode function runs tests on the trained network using the training set and stores the encoded data in a matrix.

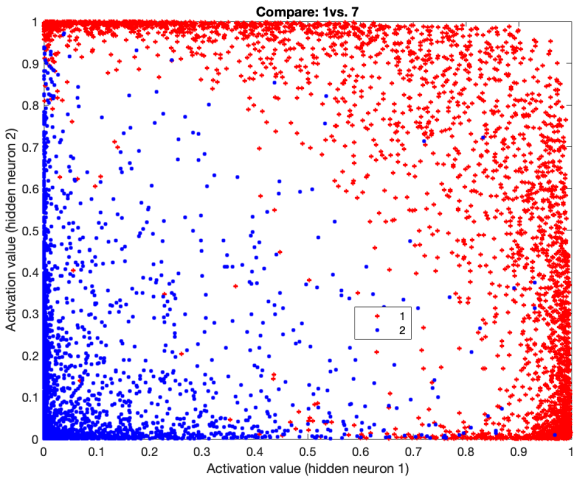


Figure 48: Hidden units activation values (Classes: 1, 7 _ Percentage 50%)

Results

Running the main script with different pairs of classes of interest and different percentages of data produces the following results:

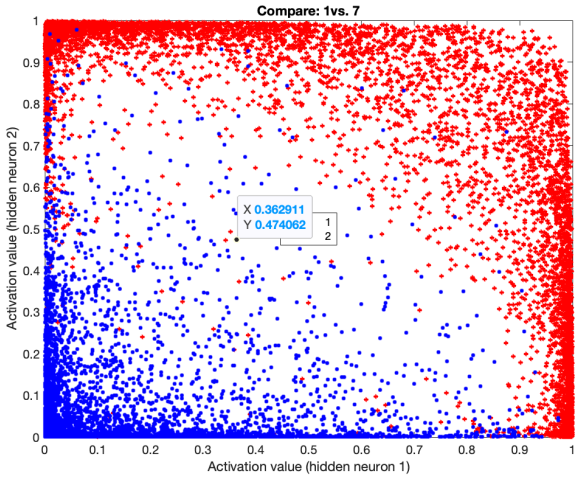


Figure 47: Hidden units activation values (Classes: 1, 7 _ Percentage 100%)

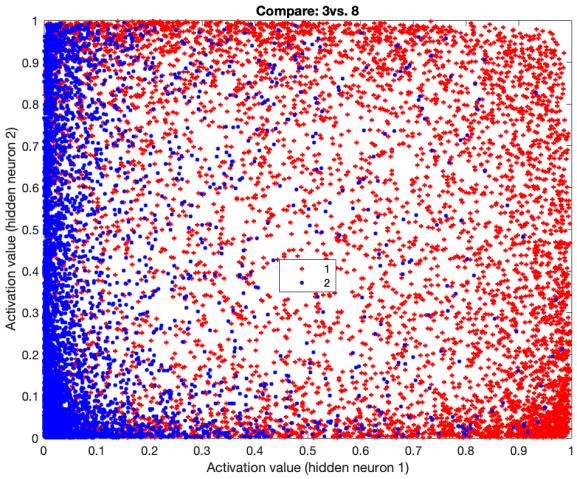


Figure 49: Hidden units activation values (Classes: 3, 8 _ Percentage 100%)

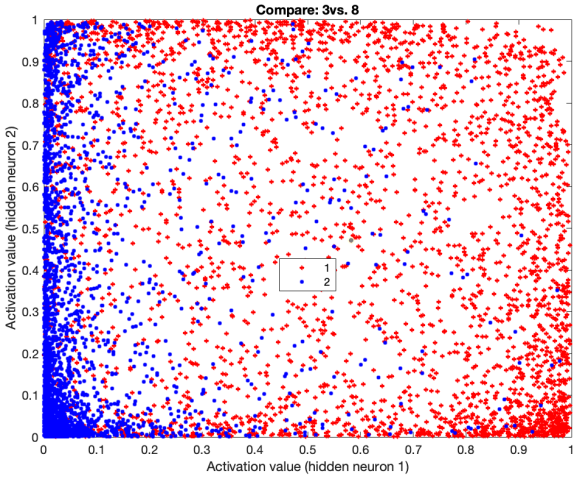


Figure 50: Hidden units activation values (Classes: 3, 8 _ Percentage 50%)

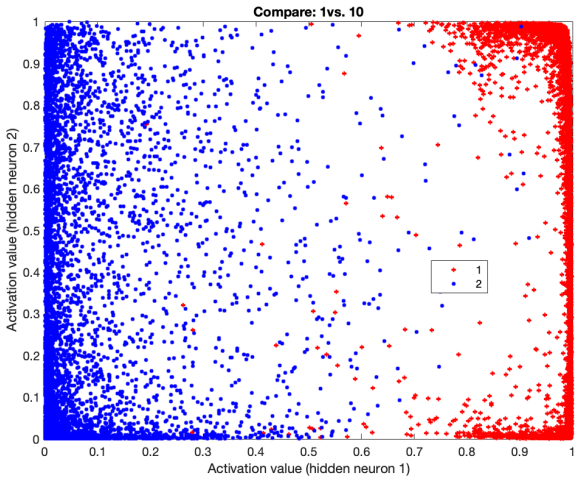


Figure 51: Hidden units activation values
(Classes: 1, 0 _ Percentage 100%)

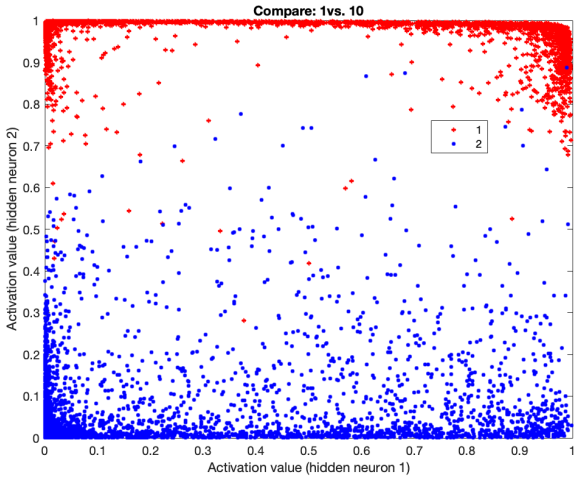


Figure 52: Hidden units activation values
(Classes: 1, 0 _ Percentage 50%)

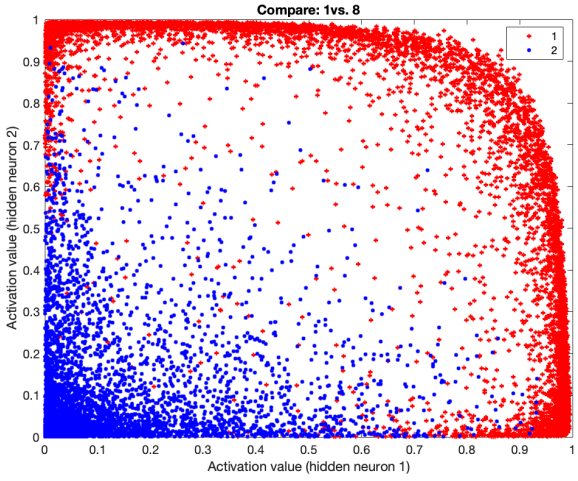


Figure 53: Hidden units activation values
(Classes: 1, 8 _ Percentage 100%)

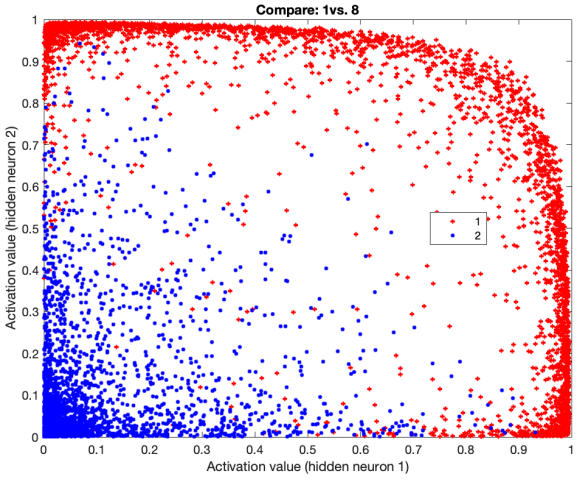


Figure 54: Hidden units activation values
(Classes: 1, 8 _ Percentage 50%)

Conclusion

Looking at the previous figure, we can see that the activation values of hidden units associated with two different classes tend to separate on different sides of the level. It can also be observed that the considered classes differ graphically from each other. In fact, 3 and 8 are very similar in shape, but (1 and 0) or (1 and 8) are very different. For (1 and 7) it is neutral instead. Since the hidden unit activation values represent the compressed training data, these results are reasonable and should somehow explain the differences or similarities between the training patterns. Needless to say, the outermost points of the graph may relate to observations that can be clearly identified as belonging to one of the two classes. Instead, items that tend to mix with items corresponding to other classes may be associated with ambiguous patterns.

References

[1] S. Rovetta, *Machine Learning for Robotics I notes and slides*, Genova University, Master Course in Robotics Engineering

List of Tables

List of Figures