CA' FOSCARI UNIVERSITY - VENICE

Department of Enviroment sciences, Informatics and Statistics

[CM0623-2] FOUNDATIONS OF ARTIFICIAL INTELLIGENCE (CM90)

Handwritten digit classifiers for the MNIST database

Student:
Michele Lotto 875922

a.y. 2022-23

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Assignment

Write a handwritten digit classifier for the MNIST database. These are composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set.

Train the following classifiers on the data-set:

- SVM using linear, polynomial of degree 2, and RBF kernels;
- Random Forest;
- Naive Bayes classifier where each pixel is distributed according to a Beta distribution of parameters α, β:

$$d(x, \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{(\alpha-1)}(1-x)^{(\beta-1)}$$

- k-NN.

You can use scikit-learn or any other library for SVM and random forests, but you must implement the Naive Bayes and k-NN classifiers yourself.

Use 10 way cross validation to optimize the parameters for each classifier.

Provide the code, the models on the training set, and the respective performances in testing and in 10 way cross validation.

Explain the differences between the models, both in terms of classification performance, and in terms of computational requirements (timings) in training and in prediction.

1.2 Project Structure

The project uses Python 3.10.6 64 bit along with the packages: "numpy" [1], "pandas" [2], "scikit-learn" [3], "matplotlib" [4]. Therefore, it is necessary to install them using "`pip3 install package_name`".

The project includes:

- 6 notebooks, one for each classifier:
  - `SVM_linear.ipynb` for the SVM classifier with linear kernel;
  - `SVM_poly.ipynb` for the SVM classifier with polynomial kernel of degree 2;
  - `SVM_RBF.ipynb` for the SVM classifier with RBF kernel;
  - `RF.ipynb` for the Random Forest classifier;
  - `naive_bayes.ipynb` for the Naive Bayes classifier;
  - `KNN.ipynb` for the KNN classifier;
- `KNN.py`, containing the implementation for the k-NN classifier;
- `naive_bayes.py`, containing the implementation for the Naive Bayes classifier;
- `utility.py`, containing utility functions used in each model notebook;
- `dataset.ipynb`, containing initial operations on data.

1.2.1 Notebook "dataset.ipynb"

The dataset.ipynb notebook has the following purposes:

- doing an initial analysis of the data-set;
- creating a common train/test split of the data for all the models to better compare them;
- saving the data to file to speed up the computation.

It is necessary to run `dataset.ipynb` before everything else to ensure the creation of initial data-set. For more detail see Section 1.3.

1.2.2 Model notebooks structure

All the model notebooks have a common structure that can be described as follows:

1. initial imports: including the library for the specific model in analysis and all the `utility.py` functions;
2. data-set loading from file and train/test split;
3. hyper-parameters tuning using " utility .model_selector(...)" or model loading from file using " utility .load(...)";
4. test of the model using test data;
5. confusion matrix.

1.2.3 utility.py

The `utility.py` Python file has the purpose of making the model notebooks more readable and avoiding code redundancy across them. It includes:

- common imports:

```
import pandas as pd
from sklearn.metrics import accuracy_score
```

- utility functions:
  - model_selector(...) ->Tuple[BaseEstimator, pd.DataFrame]: function that, given a model object, a set of hyper-parameters and a training data-set, tunes the model with the given hyper-parameters on the given training data-set using 10 folds cross-validation technique with accuracy score metric. It wraps the sklearn "GridSearchCV" function. The strategy to evaluate the performance of the cross-validated model on the test set is accuracy. It has the following input parameters:
    * model:BaseEstimator: estimator to be tuned;
    * properties:dict: hyper-parameters to tune;
    * X:pd.DataFrame;
    * Y:pd.DataFrame;
    * n_jobs:int = 1: number of parallel tunings. It parallelizes the computation on the given number of CPU core. Default to 1.
    * y_as_numpy:bool = True: True if Y needs to be converted in numpy format, False otherwise.
  - load(model_name:str)->Tuple[BaseEstimator,pd.DataFrame]: function that, given a model name, loads it and its tuning results from file.
  - save(model:BaseEstimator,result:pd.DataFrame,model_name:str)->None: function that, given a model object, its results and the model's name, saves the model and its results to file.
  - plot_confusion_matrix(test_y:pd.DataFrame, pred_y:pd.DataFrame): function that, given the true class labels (test_y) and the predicted class labels (pred_y), plots the confusion matrix.

1.3 The MNIST database

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. [5] [6] It was created by "re-mixing" the samples from NIST's original datasets, from which it takes the name. [7] Furthermore, the black and white images

from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. [5]

In the mentioned above `dataset.ipynb` notebook, the following operations on the database are done:

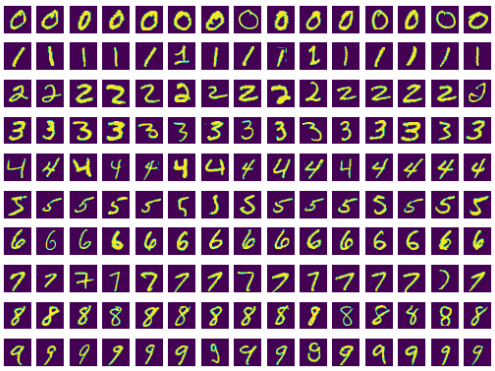
1. fetch database from the net:

```
X,y = fetch_openml('mnist_784', version=1, return_X_y=True)
y = y.astype(int)
X = X/255

```

2. save data to file:

```
X.to_csv(r'./dataset/X.csv',index=False)
y.to_csv(r'./dataset/y.csv',index=False)

```

3. plot some digits from each class:



4. drop unique value pixels (they are constant for each image, this will speed up the computation):

```
def drop_unique_col():
    count=0
    for col in X:
        if len(X[col].unique()) == 1:
            X.drop(col,axis=1,inplace=True)
            count+=1
    print("Dropped "+str(count)+" columns")

drop_unique_col()

```

This will result in 65 column drops.

5. save data with dropped columns to file:

```
X.to_csv(r'./dataset/X_dropped.csv',index=False)
y.to_csv(r'./dataset/y_dropped.csv',index=False)

```

6. save train/test split for data with dropped values to file:

```
train_X,test_X,train_y,test_y=train_test_split(X,y,
                                               test_size=10000,
                                               random_state=42)

train_X.to_csv(r'./dataset/train_X.csv',index=False)
test_X.to_csv(r'./dataset/test_X.csv',index=False)

train_y.to_csv(r'./dataset/train_y.csv',index=False)
test_y.to_csv(r'./dataset/test_y.csv',index=False)

```

   Note that the parameter random_state=42 ensures that the train/test split is the same for each execution.

Note that the database is saved to file in tree different ways:

- "as fetched from the net and not split"
- "without unique value pixels and not split"
- "without unique value pixels and split"

This is done to allow all kind of possible operations by the model notebooks.

Chapter 2

Supervised Learning: Classification

Supervised learning is a type of machine learning in which machine learns from known datasets (set of training examples), and then predict the output. [8] A supervised learning agent needs to find out the function that matches a given sample set. [8] Supervised learning further can be classified into two categories of algorithms: [8]

- Classification
- Regression

Classification is the problem of identifying which of a set of categories an observation belongs to. [9] An algorithm that implements classification is known as a classifier. [9]

2.1 Discriminative and Generative Classifiers

The goal for a classifier is to find the class label which has the maximum probability given the input data. [10] In mathematical form, given an input x and an output y, its goal is to find: [10]

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(y \mid x)$$

Thus it is necessary to estimate $P(y \mid x)$. There are two distinct approaches: [10]

- Discriminative approach: model $p(y \mid x)$ directly. In training phase, an algorithm like this tries to find the best decision boundaries that separate each class label. [10] In classification phase a new example is classified accordingly on which side of the decision boundaries it falls. [10] Support Vector Machine (SVM) and Random Forest (RF) are Discriminative algorithms. [10]
- Generative approach: model $p(x \mid y)$ and $p(y)$ separately and then use theorem 1 (Bayes theorem) to estimate $P(y \mid x)$. [10]

Theorem 1 (Bayes theorem [11]) *Bayes theorem is stated mathematically as the following equation:*

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B \mid A)P(A)}{P(B)}$$

where A and B are events and $P(B) \neq 0$.

- *$P(A \mid B)$ is a conditional probability: the probability of event A occurring given that B is true. It is also called the posterior probability of A given B.*
- *$P(B \mid A)$ is also a conditional probability: the probability of event B occurring given that A is true.*
- *$P(A)$ and $P(B)$ are the probabilities of observing A and B respectively without any given condition; they are known as the prior probability and marginal probability respectively.*

The problem can be written as: [10]

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(y \mid x) = \underset{y}{\operatorname{argmax}}\, \frac{P(x \mid y)P(y)}{P(x)}$$

Since P(x) does not depend on y can be ignored leaving the problem as: [10]

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(x \mid y)P(y)$$

k-Nearest Neighbors (k-NN) and Naive Bayes Classifier are Generative algorithms. [10]

Chapter 3

Support Vector Machines

3.1 Theoretical background

Support Vector Machine is a non-probabilistic binary linear classifier: given a training set with only two categories (binary), the SVM training algorithm maps each training example in space and finds an hyperplane which maximises the gap between the two categories (linear). [12] The category prediction for a new example is done by mapping it to a point in the space: its category is chosen according on which side of the gap it falls (non-probabilistic). [12]

In other words, given a training set of n points $(x_1, y_1), \ldots, (x_n, y_n)$, where y_i are either 1 or -1 (one of the two categories) and x_i a m-dimensional vector representing a point; the SVM training algorithm finds the hyperplane that divides the group of x_i that have $y_i = 1$ from the group of x_i that have $y_i = -1$ by maximizing the distance between the hyperplane and the nearest point x_i from either group. [12] These points are called "Support vectors" and the total distance is called "margin". [12] See Figure 3.1.

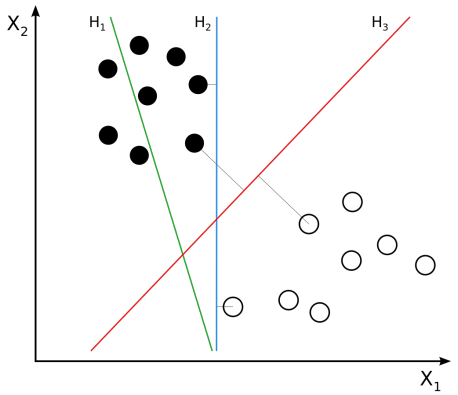


Figure 3.1: H1 does not separate the classes. H2 does, but only with a small margin. H3 separates them with the maximum margin. [12]

The hyperplane can be written as $w^T x + b = 0$ where w is the orthogonal vector to the hyperplane. The parameter $\frac{|b|}{||w||}$ determines the distance of the hyperplane from origin along the vector w. [12]

Note that, if $c \neq 0$ then $w^T x + b = 0$ and $c(w^T x + b) = 0$ define the same hyperplane. Hence w and b can be normalized such that: [13]

- $w^T x + b = +1$ for positive support vectors ($y_i = 1$);
- $w^T x + b = -1$ for negative support vectors ($y_i = -1$).

The hyperplane with such normalization is called the "canonical form hyperplane".

Given a point x, its distance from the hyperplane is given by: $\frac{|w^T x + b|}{||w||}$. Therefore the margin can be written as $\frac{2}{||w||}$. [13] See Figure 3.2.

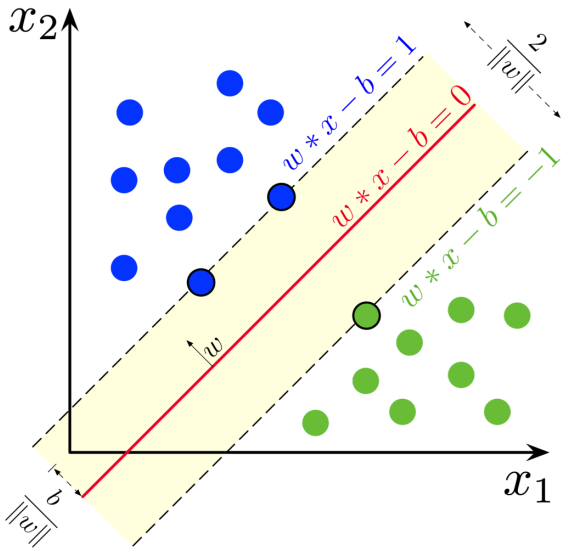


Figure 3.2: Maximum-margin hyperplane and margin for an SVM trained with samples from two classes. [12]

Therefore:

- $w^T x + b \geq +1$ for positive examples ($y_i = 1$);
- $w^T x + b \leq -1$ for negative examples ($y_i = -1$).

The learning phase of the SVM can be formulated as an optimization problem [12]:

$$\max_w \frac{2}{||w||}$$

$$\text{subject to } \begin{cases} w^T x_i + b \geq 1 & \text{if } y_i = +1 \\ w^T x_i + b \leq -1 & \text{if } y_i = -1 \end{cases}$$

$$\text{for } i = 1, \ldots, N$$

In other words, we want to find a w that maximizes the margin given that each training example must fall in the correct region of the space. [12]

The above optimization problem is equivalent to a convex quadratic optimization problem subject to linear constraints: [12] [13]

$$\begin{aligned} &\min_w \frac{1}{2}||w||^2 \\ &\text{subject to } y_i(w^T x_i + b) \geq 1 \\ &\text{for } i = 1, \ldots, N \end{aligned}$$

This consideration guarantees that a unique minimum exists.

The problem can be turned into a unconstrained one by introducing n Lagrange multipliers $\lambda_i \geq 0$ (one for each constraint) giving the Lagrangian function: [13]

$$\begin{aligned} &\min_w L(w, b, \Lambda) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \lambda_i [y_i(w^T x_i + b) - 1] \\ &\text{for } i = 1, \ldots, N \end{aligned}$$

Setting the derivatives to zero we obtain:

$$w = \sum_{i=1}^{n} \lambda_i y_i x_i$$

$$\sum_{i=1}^{n} \lambda_i y_i = 0$$

Plugging these two results in the Lagrangian function above, yields the dual representation of the maximum margin problem: [13]

$$\begin{aligned} &\max_{\Lambda} L(w, b, \Lambda) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j \\ &\text{subject to } \sum_{i=1}^{n} \lambda_i y_i = 0, \\ &\qquad\qquad \lambda_i \geq 0 \\ &\text{for all } i = 1, \ldots, n \end{aligned}$$

where $\Lambda = (\lambda_1, ..., \lambda_n)$ is the vector of Lagrange multipliers. The system has a unique optimal solution that can be found using quadratic programming or gradient ascent. [13]

Given an unknown vector u, predict class (−1,1) as follows [13]:

$$h(u) = sgn(\sum_{i=1}^{k} \lambda_i y_i x_i^T u)$$

The sum is over k support vectors.

3.1.1 Not-linear separable data

SVM can be extended to cases in which the data are not linearly separable by using: [13]

- **Soft margin**, for "nearly" linear separable data (data with outliers);
- **Mercer's Kernels**, for completely not linear separable data.

Soft margin

In some cases is better to ignore extreme points (outliers) to have an hyperplane that better adapt to the data; in particular, we allow the violation of some constraint to obtain a greater margin. [12] [13]

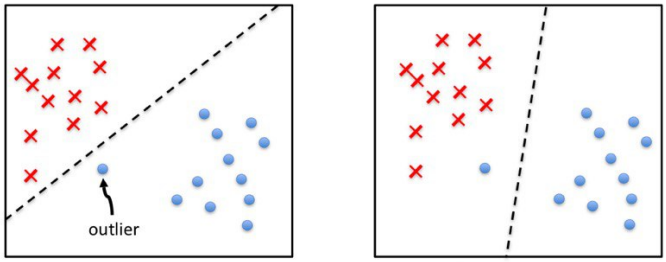


Figure 3.3: On the left a separation with a small margin that satisfies all the constraints. On the right a separation with a bigger margin that violates the outlier's constraint.

This is done by introducing slack variables (ζ_i): [12] [13]

$$\begin{aligned} &\min_w ||w||^2 + C\sum_{i=1}^{N}\zeta_i \\ &\text{subject to } y_i(w^T x_i + b) \geq 1 - \zeta_i, \quad \zeta_i \geq 0 \\ &\text{for } i = 1, \ldots, N \end{aligned}$$

where:

- $\zeta_i = \max(0, 1 - y_i(w^T x_i - b))$. In other words $\zeta_i = 0$ if the i-th constraint is satisfied i.e. if x_i falls in the correct side of the margin, otherwise ζ_i is proportional to the distance of x_i from the margin.
- C is the hyper-parameter that controls the trade-off between the classification accuracy of the training data and the maximization of the margin. This hyper-parameter needs to be fine-tuned to obtain the best SVM classifier:
  - small C: weaker constraints (large margin);
  - large C: strong constraints (narrow margin);
  - $C \to \infty$: all constraints are satisfied (hard margin).

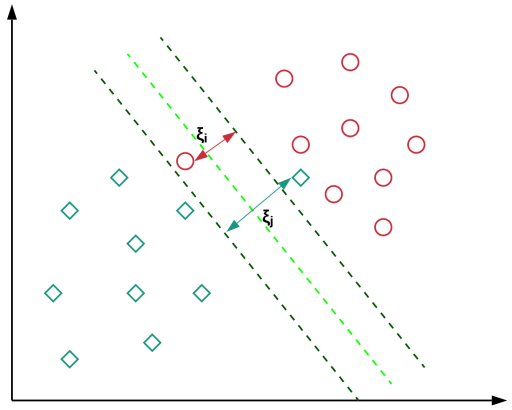


Figure 3.4: A data separation using a soft margin. [14]

In dual representation: [12] [13]

$$\max_{\Lambda} L(w, b, \Lambda) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j$$

$$\text{subject to } \sum_{i=1}^{n} \lambda_i y_i = 0,$$

$$C \geq \lambda_i \geq 0$$

$$\text{for all } i = 1, \ldots, n$$

This approach limits the effect of outliers by limiting λ_i. The soft margin optimization problem is equivalent to that of the maximum margin hyperplane with the additional constraint $C \geq \lambda_i$. [12] [13] Given this equality is always preferable to use the soft margin approach and choosing how "soft" it needs to be by fine-tuning the hyper-parameter C.

Mercer's Kernels

In some cases using a soft margin is not enough to effectively separate data. In many cases, if we transform the feature values in a non-linear way, we can transform a problem that was not linearly separable into one that is. [12] [13] The data are mapped to the new feature space by the function $\phi(.)$. [12] [13] The transformation can be to a higher-dimensional space and can be non-linear. [12] [13] Figure 3.5.

Note that in the dual representation of SVMs the input appears only in a dot-product form: to train the classifier and to use it on new examples the result of the dot-product is needed. [12] [13]

Suppose you have a non-linear function K such that: $\phi(x_i)\phi(x_j) = K(x_i, x_j)$. In this case there would be no need to compute the high-dimensional mapping $\phi(.)$. [12] [13]

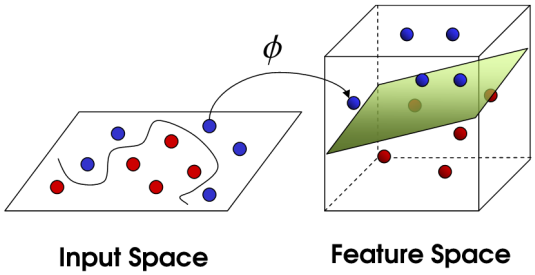


Figure 3.5: Data mapping using $\phi(.)$ function. [15]

Theorem 2 (Mercer's Reproducer Theorem [13]) *Let $K : X \times X \to R$ be a positive-definite function, then there exist a (possibly infinite-dimensional) vector space Y and a function $\phi : X \to Y$ such that $K(x_1, x_2) = \phi(x_i)\phi(x_j)$.*

Theorem 2 states that it is possible to substitute dot products with any positive-definite function K (called kernel) and that the data are implicitly mapped in a not-linear way to a high-dimensional space. [13] Choosing the right kernel is fundamental to fit the data properly. The most common choices for kernels are: [13]

- Linear kernel $K(x_i, x_j) = x_i^T x_j$: simple dot product;
- Polynomial kernel $K(x_i, x_j) = (1+x_i^T x_j)^n$ (for any $n > 0$): represents the dot product over polynomials of the original variables, allowing learning of non-linear models; The most common choice of n is $n = 2$, higher values tend to over-fit the model;
- Radius Basis kernel $K(x_i, x_j) = exp(-\frac{1}{1}\frac{||x_i - x_j||^2}{\sigma^2})$: represent the dot product as Gaussian bumps centered on the input vectors. σ^2 is a free parameter.

3.1.2 Multi-class SVM

SVM can be extended to a non-binary classifier using k linear SVM "one-vs-the-rest" binary classifiers, with k the number of categories. [16]

The classification phase requires that each model predicts a class membership probability or a probability-like score. The argmax of these scores (class index with the largest score) is then used to predict a class. [16]

See Figure 3.6.

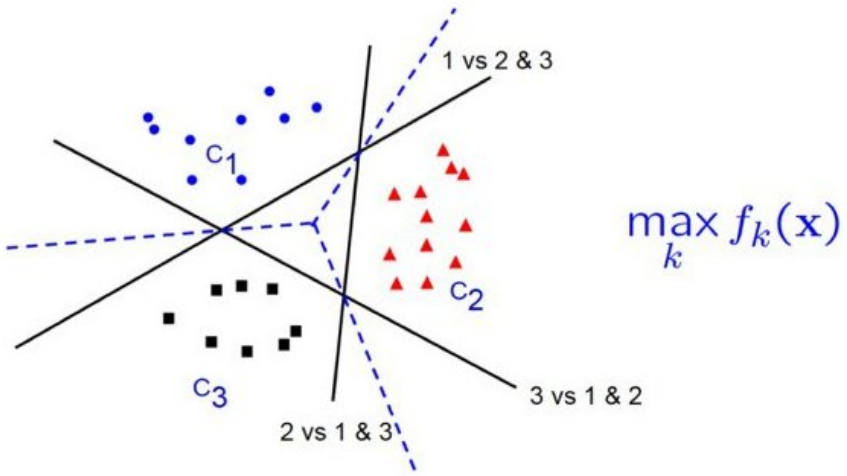


Figure 3.6: SVM multi-class with "one-vs-the-rest" approach.

3.2 Using SVM with MNIST dataset

The scikit-learn library was used for SVM classifiers. In particular "sklearn.svm.SVC" was used.

3.2.1 Results

All the SVM models were trained and tested using "without unique value pixels and split" data. For more details see Section 1.3.

The only hyper-parameter tuned was 'C'. The range of its values was initially very large and generic for all the SVM classifiers, but it was made more specific for each classifier:

- "linear kernel": [0.01,0.05,0.1,0.5,1]
- "polynomial kernel": [4,6,8,10,12]
- "RBF kernel": [9,10,11,12,13]

The difference among the ranges are a direct consequence of the kernel used: the better the data adapt to the kernel transformation, the higher "C" will be.

As explained in Section 1.2, all the models were tuned using " utility .model_selector(...)" with 10 folds cross-validation technique with accuracy score metric.

In Table 3.1 are reported the CV results for the tuning of the SVM model with linear kernel.

C parameter	mean score
0.01	0.93952
0.05	0.94295
0.1	0.94327
0.5	0.93918
1	0.93617

Table 3.1: SVM with linear kernel tuning.

In Table 3.2 are reported the CV results for the tuning of the SVM model with polynomial kernel of degree 2.

C parameter	mean score
4	0.98000
6	0.98037
8	0.98055
10	0.98038
12	0.98033

Table 3.2: SVM with polynomial kernel tuning.

In Table 3.3 are reported the CV results for the tuning of the SVM model with RBF kernel.

C parameter	CV mean score
9	0.98415
10	0.98417
11	0.98417
12	0.98412
13	0.98408

Table 3.3: SVM with RBF kernel tuning.

In Table 3.4 are reported the "C" best parameter along with the mean score for the 10 cross folds validations for each classifier.

SVM Kernel	C parameter	CV mean score
linear kernel	0.1	0.94327
polynomial kernel	8	0.98055
RBF kernel	10	0.98417

Table 3.4: SVM models with their best C parameter and best score.

As we can see, the "C" parameter and the mean score for the linear kernel SVM model are low compared to the other two models: it is clear that the data-set is not linearly separable. The other two models present a pretty decent score along with an higher C parameter, meaning that the kernel transformation captured some of the shape of the real data. Despite this considerations their C value can still be classified as a "small C", implying weak constraints: maybe exploring other kernel types could improve the score on this data-set. The highest score value along with the bigger C value was obtained by the model with RBF kernel.

In Table 3.5 are reported the accuracy scores for each model on the test set. These scores are quite similar to the CV mean scores, implying the absence of overfitting.

SVM Kernel	test score
linear kernel	0.9428
polynomial kernel	0.9793
RBF kernel	0.9816

Table 3.5: Test scores for SVM models.

In Figures 3.7, 3.8 and 3.9 are reported the confusion matrices for each SVM classifier.

The linear kernel SVM model misclassifies a significant number of examples which true and predicted digits are written quite similar. Furthermore, the model misclassifies a significant number of examples which true and predicted digits are not written quite similar: this can be caused by the low accuracy score compared to the other two models.

Similar considerations can be done with the polynomial kernel SVM model and the RBF kernel SVM model, but they misclassify less digits: they have an higher accuracy.

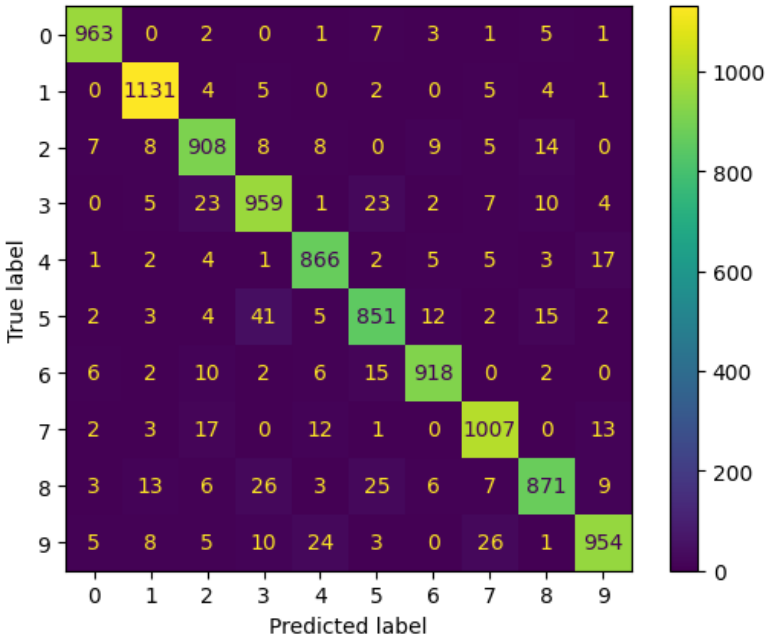


Figure 3.7: Confusion matrix for SVM with linear kernel.

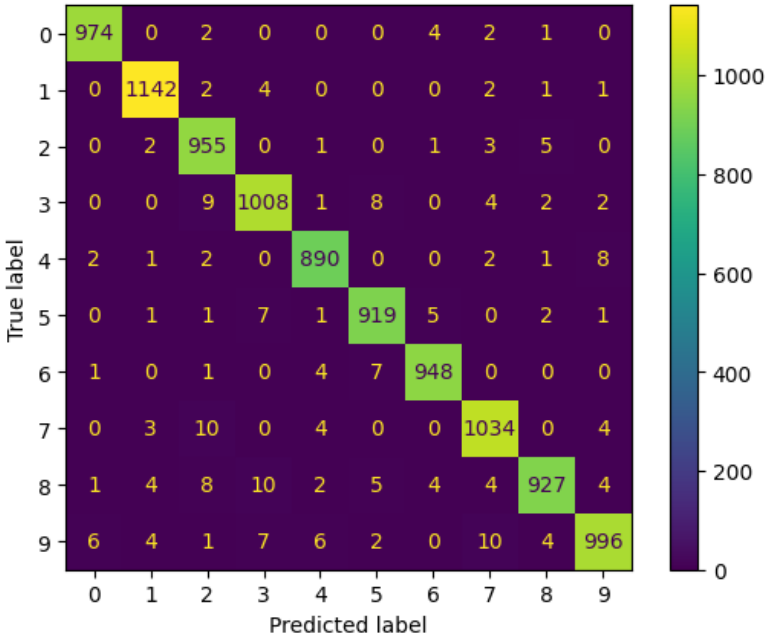


Figure 3.8: Confusion matrix for SVM with polynomial kernel of degree 2.

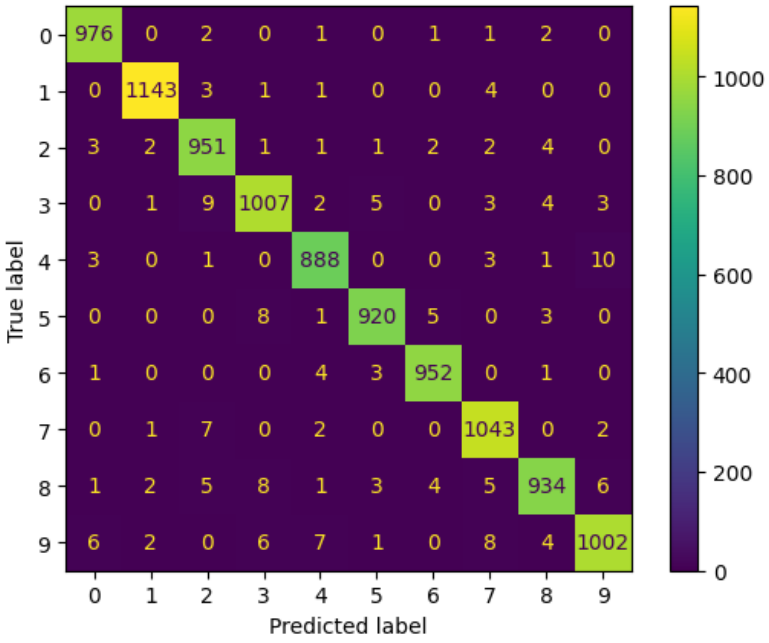


Figure 3.9: Confusion matrix for SVM with RBF kernel.

3.2.2 Performance

The models were tested using a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor along with Python 3.10.6 64bit.

The fitting and predicting time performances for each classifier are reported in Table 3.6.

SVM Kernel	fit time	predict time
linear kernel	2min 19s	40s
polynomial kernel	2min 14s	31.5s
RBF kernel	2min 44s	1min 10s

Table 3.6: Performance of SVM models.

The fitting times are quite high compared to the predicting ones. This is because the SVM algorithm needs to solve the optimization problem mentioned in Section 3.1 at fitting time; instead, at predicting time, it only needs to apply the $h(.)$ function to each test example, that is way less computationally expensive.

Furthermore, the predicting time for the RBF kernel is quite high compared to the other two, this can be explained by an higher number of support vectors present in this model.

Chapter 4

Random Forests

4.1 Theoretical background

Random forests (RFs) or random decision forests (RDFs) are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time. [17] An ensemble method uses multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. [18] Indeed, to understand RF, it is mandatory to explain how a decision tree works.

4.1.1 Decision Trees

Decision Trees (DTs) are a supervised learning method used for classification and regression. [19] The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features (one for each internal node). [19] The leaf nodes of the tree represent the final decision. Figure 4.1.

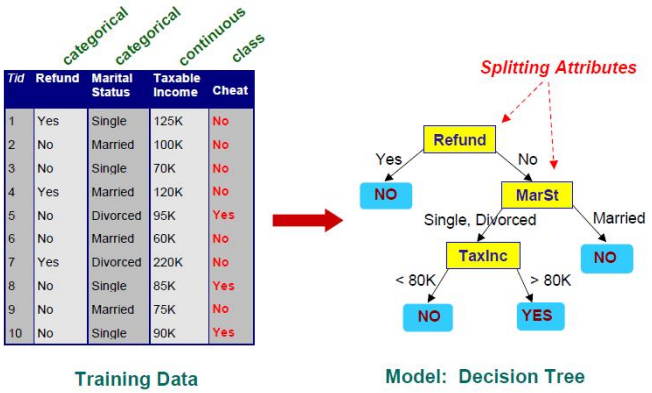


Figure 4.1: Example of a decision tree. [20]

For the aim of this project, only the DT classifier is fully explained.

In classification phase, starting from the root node, the test condition is applied to the given record and it is chosen the appropriate branch based on the outcome of the test. [20] The choice can lead either to another internal node, for which a new test condition is applied, or to a leaf node. When the leaf node is reached, the class label associated with the leaf node is then assigned to the record. [20] Figure 4.2.

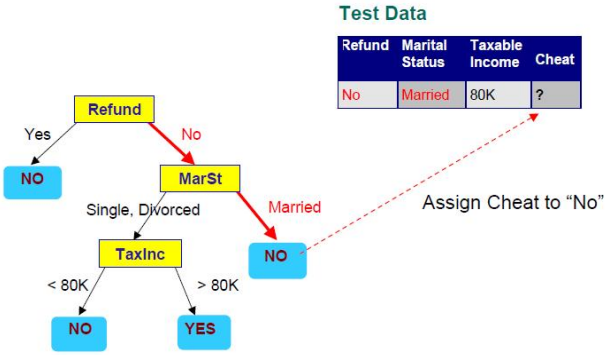


Figure 4.2: Example of classification using a decision tree. [20]

In train phase, the ID3 Algorithm constructs the decision tree; in particular is necessary to choose what test to perform, which variables to consider on each test, the position and the number of each internal node. [21] The ID3 Algorithm uses the Information Gain measure to take these decisions. [21]

Information Gain

The Information Gain measure is based on the notion of entropy. [21] Given a set of examples S labeled in $c_1, \ldots, c_n$, with p_i the proportion of examples with c_i label, the entropy of S is defined as: [21]

$$H(S) = -\sum_{i=1}^{n} p_i log_2(p_i)$$

In other words it is the label-wise average of $-log_2(p_i)$, representing how much information is obtained from label i examples. [21] When p_i is nearly 0, $-log_2(p_i)$ is very large: if the proportion of examples with label c_i is very small, the information obtained from its examples is high. On the other hand, when p_i is nearly 1, $-log_2(p_i)$ is very small: if the proportion of examples with label c_i is very big, the information obtained from its examples is low.

Thus, in the entropy calculation, when p_i is either close to 0 or close to 1 the examples labeled c_i will score low [21]:

- when p_i is close to 0, the information obtained by them is high but less significant, because their number is low;

- when p_i is close to 1, the information obtained by them is low but more significant, because their number is high.

Given a set of examples S and an attribute A, which can take values $v_1, \ldots, v_m$: $Gain(S, A)$ estimates the reduction in entropy when the value of attribute A for the examples in S in known. [21] In mathematical form: [21]

$$Gain(S, A) = H(S) - \sum_{i=1}^{m} \frac{|S_{v_i}|}{|S|} H(S_{v_i})$$

where $S_{v_i} = \{$examples which take value v_i for attribute $A\}$. [21]

In other words, it measures the information gain that is obtained by knowing the value of attribute A. [21]

The ID3 Algorithm

The ID3 algorithm is a recursive algorithm [21]:

1. If S only contains examples in category c, return a leaf node with category c.
2. If S is empty, return a leaf node with the default category (which contains the most examples from the initial S).
3. Choose the root node to be attribute A, such that Gain(S,A) is maximum.
4. For each value v that A can take, draw a branch and label each with corresponding v.
5. For each value of v (for each branch):
   - 5.1 Sv={examples in S with attribute A=v}
   - 5.2 Remove A from attributes that can be chosen
   - 5.3 Return ID3(Sv).

The base cases for recursion are 1 and 2.

4.1.2 Decision tree weakness

Decision trees can be learned to perfectly fit the given data. This will probably lead to overfitting. [21] Therefore, it is important to tune the tree depth to avoid it, but the problem still remains: single pruned trees are poor predictors (low variance, but very high bias) (Figure 4.3) and single deep trees are noisy (low bias, but very high variance) (Figure 4.4). [21]

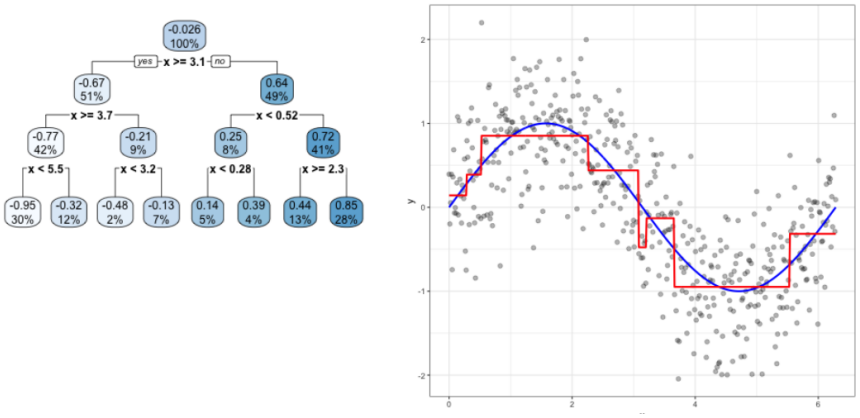


Figure 4.3: Decision tree with Depth=3. [21]

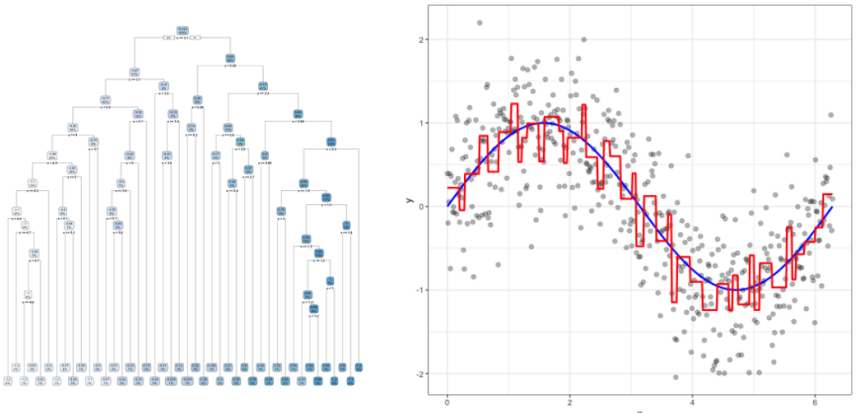


Figure 4.4: Decision tree with Depth=20. [21]

Therefore it is necessary to apply some techniques to reduce either the bias (boosting) or the variance (bagging/random forest). For the aim of this project only the random forest technique is fully explained. [17]

4.1.3 Random Forest Classifier

Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. [17] This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model. [17]

In training phase:

Given a training set $X = x_1, \ldots, x_n$ with responses $_Y = y1, \ldots, y_n$, the algorithm repeatedly (B times) selects a random sample with replacement of the training set and fits trees to these samples: [17]

1. For b = 1, ..., B:

   1.1 Sample, with replacement, n training examples from X, Y; call these X_b, Y_b.

   1.2 Train a classification tree f_b on X_b, Y_b.

RF use a modified tree learning algorithm that selects, at each candidate split in the learning process, a random subset of the features. [17] This process is sometimes called "feature bagging". [17] The reason for doing this is reducing the correlation of the trees: if one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the B trees, causing them to become correlated. [17] Typically for classification trees, $\sqrt{p}$ features are used in each split, with p number of features. [17]

In the prediction phase, for unseen samples x' prediction can be made by taking the mode among all the tree predictions. [17] Figure 4.5.

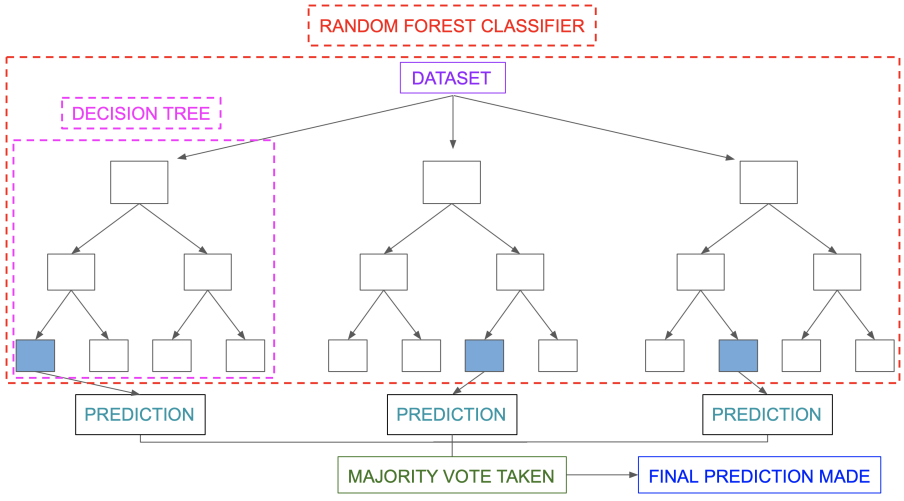


Figure 4.5: Random Forest prediction. [22]

Random forest introduces randomness into the rows and columns of the data, providing a more diverse set of trees that almost always lowers the prediction error. [17]

4.2 Using RF classifier with MNIST dataset

The scikit-learn library was used for RF classifier.

In particular "sklearn.ensemble.RandomForestClassifier" was used.

4.2.1 Results

The RF model was trained and tested using "without unique value pixels and split" data. For more details see Section 1.3.

The only hyper-parameter tuned was 'n_estimators', representing the number of trees in the forest. The range of its values was initially very large and generic, but it was made more specific later: `[x for x in range(100,1100,100)]`.

It was chosen to let every single tree of the RF grow to large depth, to minimize the bias and maximize the variance of each model. Then, the RF algorithm should minimize the total variance with the correct choice of 'n_estimators'.

As explained in Section 1.2, the model was tuned using "`utility.model_selector( ... )`" with 10 folds cross-validation technique with accuracy score metric.

In Table 4.1 are reported the CV result for the tuning of the RF model. The best number of trees found was 700 with an accuracy of 0.970367.

number of trees	CV mean score
100	0.968333
200	0.969183
300	0.969983
400	0.969617
500	0.970050
600	0.970033
700	0.970367
800	0.970017
900	0.970050
1000	0.970367

Table 4.1: RF tuning.

The accuracy score for RF on the test set is 0.9663. This score is quite similar to the CV mean score, implying the absence of overfitting.

In Figure 4.6 is reported the confusion matrix for RF classifier. This classifier misclassifies both a significant number of examples which true and predicted digits are not written quite similar and a significant number of examples which true and predicted digits are written quite similar.

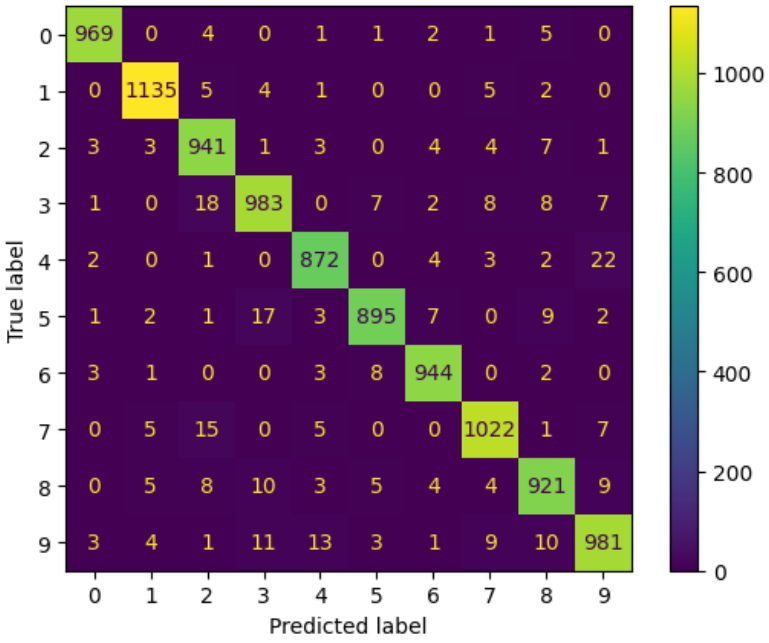


Figure 4.6: Confusion matrix for RF.

4.2.2 Performance

The model was tested using a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor along with Python 3.10.6 64bit.

The fitting and predicting time performances for RF classifier are respectively "4min 18s" and "2.9s". The fitting time is quite high compared to the predicting one. This is because at training time the RF algorithm needs to sample 'n_estimators' times the dataset, a quite expensive operation, and create the same number of trees from these samples; while at predicting time it only needs to take the most common class label by executing each tree on the same test example, a quite fast operation.

Chapter 5

Naive Bayes Classifier

5.1 Theoretical background

Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes theorem (1) with strong (naive) independence assumptions between the features given the class label. [23] [10]

Naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector $X = (x_1, \ldots, x_n)$ representing some n features, it assigns to this instance probabilities: [23] [10]

$$P(C_1|(x_1, \ldots, x_n)), \ldots, P(C_m|(x_1, \ldots, x_n))$$

One probability for each class label.

Using Bayes theorem (1), the conditional probability can be decomposed as: [23] [10]

$$P(C_k \mid X) = \frac{P(X \mid C_k)P(C_k)}{P(X)}$$

In practice, there is interest only in the numerator of that fraction, because the denominator does not depend on C and the values of the features x_i are given, so that the denominator is effectively constant and can be ignored. [23] [10] Furthermore, the numerator is equivalent to the joint probability model: [23] [10]

$$P(C_k, x_1, \ldots, x_n)$$

which can be rewritten as follows by repetitively applying the definition of conditional probability: [23] [10]

$$\begin{aligned} P(C_k, x_1, \ldots, x_n) &= P(x_1, \ldots, x_n, C_k) \\ &= P(x_1 \mid x_2, \ldots, x_n, C_k)P(x_2, \ldots, x_n, C_k) \\ &= P(x_1 \mid x_2, \ldots, x_n, C_k)P(x_2|x_3, \ldots, x_n, C_k)P(x_3, \ldots, x_n, C_k) \\ &= \ldots \\ &= P(x_1 \mid x_2, \ldots, x_n, C_k) \ldots P(x_{n-1}|x_n, C_k)P(x_n, C_k)P(C_k) \end{aligned}$$

With the "naive" conditional independence assumptions: [23] [10]

$$P(x_i|x_{i+1},\dots,x_n,C_k) = P(x_i|C_k)$$

Thus, the joint probability model can be expressed as: [23] [10]

$$\begin{aligned} P(C_k \mid x_1,\dots,x_n) &\propto P(C_k,x_1,\dots,x_n) \\ &\propto P(C_k)P(x_1|C_k)P(x_2|C_k)\dots P(x_n|C_k) \\ &\propto P(C_k)\prod_{i=1}^{n} P(x_i|C_k) \end{aligned}$$

where $\propto$ denotes proportionality (the denominator is ignored).

The most important assumption that Naive Bayes Classifier does is that the distribution of each feature is known. [23] [10] The assumptions on distributions of features are called the "event model" of the naive Bayes classifier. The most common choices are: [23]

- Gaussian distribution: used when the predictors take up a continuous value and are not discrete.
- Multinomial distribution: mostly used for document classification problem. The features/predictors used by the classifier are the frequency of the words present in the document.
- Bernoulli distribution: similar to the multinomial naive bayes but the predictors are boolean variables.

In training phase: [23] [10]

- $P(C_k)$ is calculated estimating the class probability from the training set (number of samples in the class/number of samples).
- The parameters for each feature distribution are estimated from the training set data.

The corresponding classifier is the function that assigns a class label $\hat{y} = C_k$ for some k as follows: [23] [10]

$$\hat{y} = \operatorname*{argmax}_k P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$$

where x_i is a test set example.

5.2 Using Naive Bayes classifier with MNIST dataset

A fundamental assumption given in the project assignment is that each pixel is distributed according to Beta distribution:

$$d(x,\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{(\alpha-1)}(1-x)^{(\beta-1)}$$

with $\alpha > 0$ and $\beta > 0$.
The α and β parameters were estimated using the moments approach:

$$\alpha = KE[X]$$
$$\beta = K(1 - E[X])$$
$$K = \frac{E[X](1 - E[X])}{Var[X]} - 1$$

With these estimates there are two problems:

- The value of K is undefined if $Var[X] = 0$; making the values of both α and β undefined. This can be solved by making some assumptions on $P(X \mid Y)$: when $Var[X] = 0$, it means that X can take only a single value; making it completely deterministic. Thus, $P(X = x \mid Y) = 1$ when x is the single value that X can take and $P(X = x \mid Y) = 0$ otherwise.

- The value of k is negative or zero iff:

  $$\frac{E[X](1 - E[X])}{Var[X]} - 1 \leq 0$$

  This makes $\alpha \leq 0$ and $\beta \leq 0$. To solve this problem the negative α and β values were set to the minimum positive value of α and β respectively of the other pixel distributions for the given class.

5.2.1 Implementation

The implementation consists in a single class "class BetaDistribution_NaiveBayes(BaseEstimator)", inheriting the "BaseEstimator" template, which is the base class for all estimators in scikit-learn. An external library for the calculation of the Beta distribution probability was used called "scipy".

The Naive Bayes class implements four methods:

- fit (self , X_train:pd.DataFrame, y_train:pd.DataFrame) –> BetaDistribution_NaiveBayes: function for fitting this estimator. It takes two parameters:

  - X_train:pd.DataFrame: The features data as a pandas data-frame.
  - y_train:pd.DataFrame | np.ndarray: The class labels as a pandas data-frame or a numpy array.

  Both X_train and y_train are used for setting the corresponding class parameter. An additional private method __param_estimation(self) –> None is then executed. Additionally the fit method returns the fitted estimator, allowing quick one liners. The code can be found in Listing 5.1.

```
def fit(self,train_X:pd.DataFrame,train_y:pd.DataFrame):
    self.train_X=train_X
    self.train_y=train_y
    self.__param_estimation()

    return self

```

Listing 5.1: NaiveBayes fit function.

- __param_estimation(self) -> None: function that estimates α and β parameters for each Beta distribution and calculates all class label frequencies ($P(C_k)$). Listing 5.2.

```
n=#class label

#images of class n
images_class_n=self.train_X[self.train_y["class"]==n]

#mean and variance for each pixel of class n
means_pixels_class_n=images_class_n.mean(axis=0)
variances_pixels_class_n=images_class_n.var(axis=0)

#alpha and beta estimation
ks_pixels_class_n=((means_pixels_class_n*(1-means_pixels_class_n))
                   /variances_pixels_class_n)-1

alphas_pixels_class_n=ks_pixels_class_n*means_pixels_class_n
betas_pixels_class_n=ks_pixels_class_n*(1-means_pixels_class_n)

#class frequency
frequency=self.train_y[self.train_y["class"]==n].size
          /self.train_y["class"].size

```

Listing 5.2: Calculation of alpha and beta parameters and class frequency.

The negative value α and β parameters are substituted by the minimum α and β value respectively for the pixel distributions of the given class. Listing 5.3.

```
#negative alpha and beta
alphas_pixels_class_n[alphas_pixels_class_n<=0]=
        alphas_pixels_class_n[alphas_pixels_class_n>0].min()

betas_pixels_class_n[betas_pixels_class_n<=0]=
        betas_pixels_class_n[betas_pixels_class_n>0].min()

```

Listing 5.3: Negative value substitution for alphas and betas.

Additionally, it calculates another variable called "unique_counts" which is equal to -1 for pixels that have more than one possible value and it is equal to the only possible

value of the pixel for each pixel that has only one possible value. This last parameter is then used to avoid the zero variance problem explained above. Listing 5.4.

```
#unique value pixel (for nan value alphas and betas)
unique_counts=images_class_n.nunique(axis=0, dropna=True)

unique_counts[unique_counts > 1] = -1

unique_counts[unique_counts == 1] =
    means_pixels_class_n[unique_counts == 1]

```

Listing 5.4: Calculation of "unique_counts." variable

Another variable is calculated being the mean of each distribution. The 'nan' values are substituted with the mean of the pixel distribution. Listing 5.5.

```
#Beta means
beta_means_class_n=(alphas_pixels_class_n)
            /(alphas_pixels_class_n+betas_pixels_class_n)
beta_means_class_n[unique_counts != -1]=
            unique_counts[unique_counts != -1]

```

Listing 5.5: Calculation of the mean for each pixel distribution.

These parameters and variables are stored in a dict of dicts (one for each class label). Listing 5.6.

```
self.parameter_per_class[n]={
                'alphas':alphas_pixels_class_n.to_numpy(),
                'betas':betas_pixels_class_n.to_numpy(),
                'unique':unique_counts.to_numpy(),
                'Beta_means':beta_means_class_n.to_numpy(),
                'frequency':frequency}

```

Listing 5.6: Dic for class n.

- predict(self , test_X:pd.DataFrame) -> pd.Series: function for predicting class labels for new examples using this estimator. It takes only a parameter (test_X:pd.DataFrame): the features data as a pandas data-frame. For each image in test_X, the class which maximizes $P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$ needs to be found; for each class:

  1. α and β parameters retrieval, along with "unique_counts".
  2. calculation of $\prod_{i=1}^{n} P(x_i|C_k)$ by using the Beta cumulative density function (CDF); to evaluate a single point (r) probability the following integral was computed:

$$\int_{r-\epsilon}^{r+\epsilon} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{(\alpha-1)}(1-x)^{(\beta-1)}\,dx$$

  The ϵ chosen was 0.1.

3. substitution of probabilities for distributions which α and β parameters are 'nan' as explained in Section 5.2.
4. calculation of $P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$. If it is the bigger value so far, set the current class label C_k to be the value which maximizes the probability.

The code can be found in Listing 5.7. All the operation were vectorized for performance reasons.

```
def predict(self,test_X:pd.DataFrame) -> pd.Series:

    epsilon=0.1

    output=[]
    indexes=[]

    for row_i,row in test_X.iterrows():

        _max=0
        _max_class=-1

        row=row.to_numpy()

        for n in range(10):

            #parameters
            class_parameters=self.parameter_per_class[n]

            _alpha=class_parameters['alphas']
            _beta=class_parameters['betas']
            _unique=class_parameters['unique']

            #integral
            beta_probabilities=beta.cdf(row+epsilon,_alpha,_beta)
                               -beta.cdf(row-epsilon,_alpha,_beta)

            #unique values
            beta_probabilities[ np.logical_and(_unique!=-1,
                                _unique != row) ] = 0

            beta_probabilities[ np.logical_and(_unique!=-1,
                                _unique == row) ] = 1

            probability=class_parameters['frequency']
                        *np.product(beta_probabilities)

            if probability>_max:
                _max=probability
                _max_class=n

        output.append(_max_class)
```

```
43          indexes.append(row_i)
44
45      return pd.Series(data=output,index=indexes)
46
```

Listing 5.7: Naive Bayes predict function.

- mean_plot(self)->None: function that plots for each class the mean of each pixel distribution previously calculated in ___param_estimation(self) -> None. Listing 5.8.

```
1  def mean_plot(self)->None:
2
3      _,axs=plt.subplots(3,4)
4
5      axs = [item for sublist in axs for item in sublist]
6
7      axs[10].axis('off')
8      axs[11].axis('off')
9
10     for digit in range(10):
11
12         means_class_n=self.parameter_per_class[digit]["Beta_means"]
13
14         axs[digit].imshow(means_class_n.reshape(28, 28))
15         axs[digit].axis('off')
16
```

Listing 5.8: Mean plot for each class.

5.2.2 Results

For this model there was no hyper-parameter to tune. It was trained and tested using the "without unique value pixels and split" data. For more details see Section 1.3.

The accuracy score on the test set is 0.8431. In Figure 5.1 is reported the confusion matrix for this model. The Naive Bayes misclassifies a significant number of examples which true and predicted digits are written quite similar, while the number of examples which true and predicted digits, that are written quite different, misclassified is low, compared to the previous one.

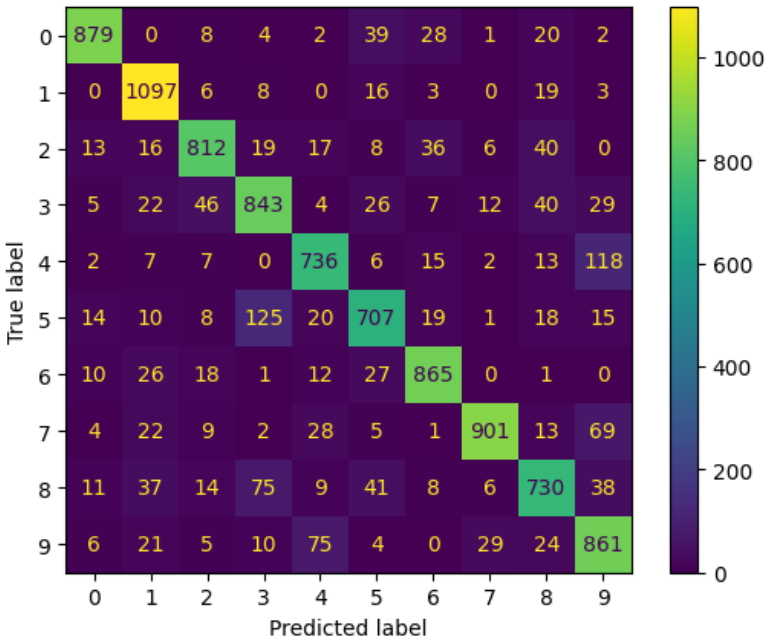


Figure 5.1: Confusion matrix for Naive Bayes Classifier.

Furthermore for this model, as suggested in the assignment, for each class the mean of each pixel distribution was plotted; to do so, the model was retrained using the "as fetched from the net and not split" data, because it was needed to have 784 pixels (distributions) to reshape them into 28x28 images. Figure 5.2. This is a visual indication of what the model is learning: each digit is clearly visible.

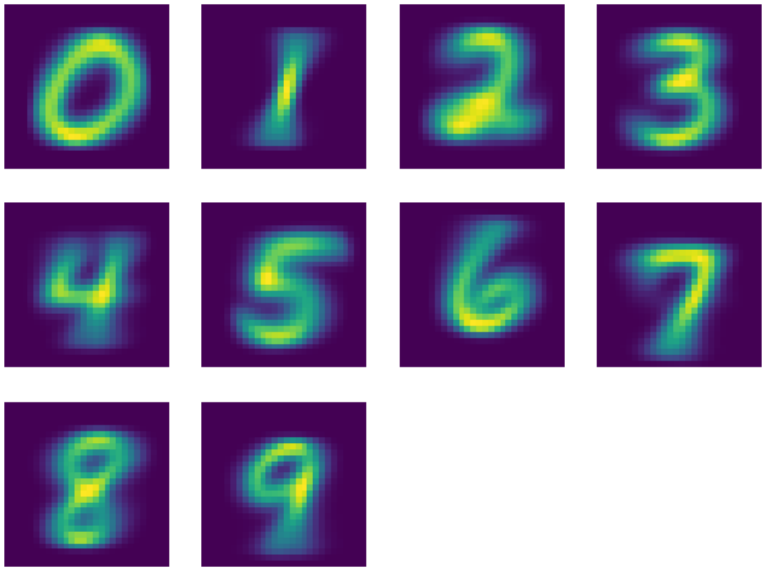


Figure 5.2: For each class, plot of mean values for each pixel distribution.

5.2.3 Performance

The model was tested using a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor along with Python 3.10.6 64bit.

The fitting and predicting time performances for Naive Bayes classifier are respectively "1s" and "1min 33s". The fitting time is quite low compared to the predicting one. This is because at training time the Naive Bayes algorithm only estimates the parameters for each class and for each pixel distribution; while at predicting time for each test example, it computes an integral for each pixel distribution and for each class, a quite expensive operation.

Chapter 6

k-Nearest Neighbors

6.1 Theoretical background

The k-nearest neighbors algorithm (k-NN) is a supervised learning method. It is used for both classification and regression. [24] For the aim of understanding this project, only the k-NN classifier is fully explained.

The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples. [24]

In the classification phase, an unlabeled vector (test point) is classified by assigning the label which is most frequent among the k (neighbors) training samples nearest to that test point. [24]

In Figure 6.1, the test sample (green dot) should be classified either to blue squares or to red triangles. If k = 3 (solid line circle) it is assigned to the red triangles (2 triangles vs 1 square). If k = 5 (dashed line circle) it is assigned to the blue squares (3 squares vs. 2 triangles). [24]

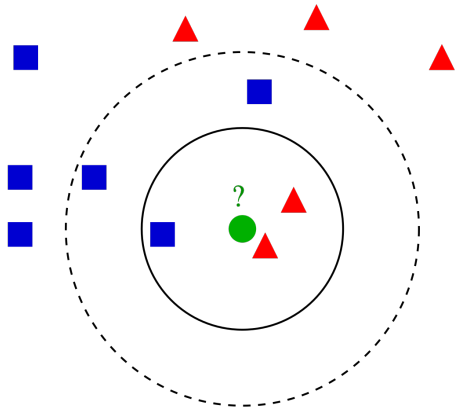


Figure 6.1: Example of k-NN classification. [24]

In general, for points given by Cartesian coordinates in n-dimensional Euclidean space, the distance (Euclidean distance) is [25]:

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2}$$

where a and b are n-dimensional vectors.

The best choice of k depends upon the data; generally, larger values of k reduce effect of the noise on the classification, but make boundaries between classes less distinct. [24] The special case where the class is predicted to be the class of the closest training sample (i.e. when k = 1) is called the nearest neighbor algorithm. [24] k acts as a smoothing parameter in the decision boundary. [10]

As n and k increase, k more slowly than n, k-NN converges to the optimal classifier. [10]

Nearest neighbor is great in low dimensions, but as it increases almost all points are far away from one another and almost all near the boundary. [10]

6.2 Using k-NN classifier with MNIST dataset

The implementation follows the scikit-learn guideline, this guarantees that it is fully compatible with "GridSearchCV" and other sklearn functions.

6.2.1 Implementation

The implementation consists in a single class "class KNN(BaseEstimator)", inheriting the "BaseEstimator" template, which is the base class for all estimators in scikit-learn. The class constructor initializes the only hyper-parameter "k", representing the number of neighbours. Listing 6.1.

```
1 def __init__( self, k:int = 5 ) -> None:
2     self.k=k
3
```

Listing 6.1: KNN class constructor.

The KNN class implements two methods:

- fit (self , X_train:pd.DataFrame, y_train:pd.DataFrame | np.ndarray) -> KNN: function for fitting this estimator. It takes two parameters:
  - X_train:pd.DataFrame: The features data as a pandas data-frame.
  - y_train:pd.DataFrame | np.ndarray: The class labels as a pandas data-frame or a numpy array.

  Both X_train and y_train are converted to numpy arrays for efficiency reasons and are used for setting the corresponding class parameter. As mentioned in section 6.1, the fit method doesn't do additional elaboration on the training data. Moreover, the

fit method returns the fitted estimator, allowing quick one liners. The code can be found in Listing 6.2.

```
def fit( ... ) -> KNN:

    if X_train.shape[0]!=y_train.shape[0]:
        raise ValueError("The number of rows should be the same")

    self.X_train=X_train.to_numpy()

    if isinstance(y_train,pd.DataFrame):
        self.y_train=y_train.to_numpy()
    else:
        self.y_train=y_train

    return self

```

Listing 6.2: KNN fit function.

- predict(self, test_X:pd.DataFrame) -> pd.Series: function for predicting class labels for new examples using this estimator. It takes only a parameter (test_X:pd.DataFrame): the features data as a pandas data-frame.

  The method returns the predicted class labels as a pandas Series. Before the actual computation the predict method converts test_X to a numpy array for efficiency reasons after saving the pandas index for each row.

  For each new example:

  1. compute the euclidean distances to each training example;
  2. label the new example with most frequent label among the first k nearest training examples.

  The code can be found in Listing 6.3. All the operation were vectorized for performance reasons.

```
def predict( self, test_X:pd.DataFrame ) -> pd.Series:

    indexes=test_X.index.to_list()
    test_X=test_X.to_numpy()

    output=[]

    for row in test_X:

        distances=np.linalg.norm(self.X_train - row, axis=1)

        mode=st.mode((self.y_train[np.argsort(distances)]
                      [:self.k]).flatten())
```

```
14
15          output.append(mode)
16
17      return pd.Series(data=output,index=indexes)
18
```

Listing 6.3: KNN predict function.

6.2.2 Results

The k-NN model was trained and tested using half size of the "without unique value pixels and not split" data, for performance reasons. The used examples were chosen at random. For more details see Section 1.3.

The only hyper-parameter tuned was k (number of neighbors). The range of its values was initially very large and generic, but it was made more specific later: [2,3,4,5,6,7] .

As explained in Section 1.2, the model was tuned using "utility.model_selector(...)" with 10 folds cross-validation technique with accuracy score metric.

In Table 6.1 are reported the CV result for the tuning of the k-NN model. The best k found was 4 with an accuracy of 0.967133.

k parameter	CV mean score
2	0.965367
3	0.965667
4	0.967133
5	0.965533
6	0.965733
7	0.963433

Table 6.1: k-NN tuning.

The accuracy score on the test set is 0.9638. This is a good result given that the data provided are half of the total; augmenting the training data-set could improve even further this result. Furthermore, this score is quite similar to the CV mean score, implying the absence of overfitting.

In Figure 6.2 is reported the confusion matrix for this model. The k-NN misclassifies a significant number of examples which true and predicted digits are written quite similar, but even examples which true and predicted digits are written quite different. This could be caused from the limited number of examples used for fitting this model.

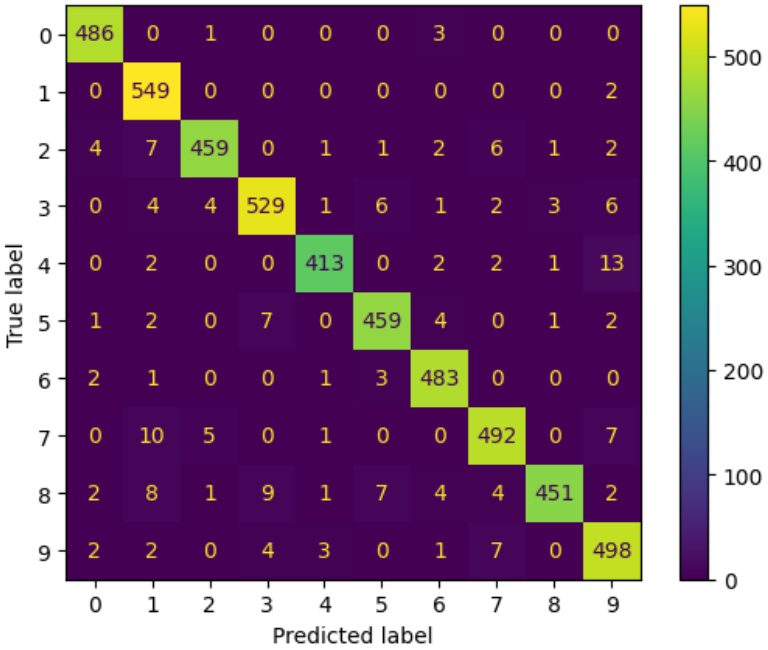


Figure 6.2: Confusion matrix for k-NN.

6.2.3 Performance

The model was tested using a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor along with Python 3.10.6 64bit.

The fitting and predicting time performances for k-NN classifier are respectively "1s" and "7min 28s". The fitting time is quite low compared to the predicting one. This is because at training time the k-NN algorithm only stores the training data, without further elaborations; while at predicting time for each test example, it computes the distances to each point, a quite expensive operation.

Chapter 7

Conclusions

In this project it was required to analyse different classifiers both in terms of classification performance and in terms of computational requirements (timings) in training and in prediction. A summary of this analysis is provided in Table 7.1.

model	accuracy	fit time	predict time	total time
SVM linear kernel	0.9428	2min 19s	40s	2min 59s
SVM polynomial kernel	0.9793	2min 14s	31.5s	2min 45s
SVM RBF kernel	0.9816	2min 44s	1min 10s	3min 54s
Random Forest	0.9663	4min 18s	3s	4min 21s
Naive Bayes	0.8431	1s	1min 33s	1min 34s
k-NN	0.9638	1s	7min 28s	7min 29s

Table 7.1: Accuracy score and fit/predict time performances for each model.

The best model for accuracy is 'SVM RBF kernel', while the worst one is 'Naive Bayes'. The best models for fit time are 'Naive Bayes' and 'k-NN', while the worst one is 'Random Forest'. The best model for predict time is 'Random Forest', while the worst one is 'k-NN'. The best model for total time is 'Naive Bayes', while the worst one is 'k-NN'.

The 'k-NN' classifier is clearly the overall worst model, the predict time is too large and the accuracy obtained is mediocre. Its accuracy could be surely improved using all the possible data, but in this case the predicting time will be even larger.

On the other hand, the 'Naive Bayes' reaches almost the 0.85 of accuracy in one and a half minute.

A good compromise could be the 'SVM polynomial kernel' classifier: it almost reaches the best accuracy in a minute less than 'SVM RBF kernel'.

Bibliography

[1] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] The pandas development team. *pandas-dev/pandas: Pandas*. Version 1.5.2. Nov. 2022. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[3] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[4] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[5] Yann LeCun et al. *"THE MNIST DATABASE of handwritten digits"*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 12/26/2022).

[6] Vision Systems Design. *"Support vector machines speed pattern recognition - Vision Systems Design"*. URL: https://www.vision-systems.com/home/article/16737424/support-vector-machines-speed-pattern-recognition (visited on 12/26/2022).

[7] Patrick J. Grother. *"NIST Special Database 19 - Handprinted Forms and Characters Database"*. National Institute of Standards and Technology. URL: https://www.nist.gov/system/files/documents/srd/nistsd19.pdf (visited on 12/26/2022).

[8] *"Supervised learning"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Supervised_learning (visited on 12/26/2022).

[9] *"Statistical classification"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Statistical_classification (visited on 12/26/2022).

[10] Andrea Torsello - Ca’ Foscari University. “Generative Models slides”.

[11] *"Bayes’ theorem"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Bayes%5C%27_theorem (visited on 12/26/2022).

[12] *"Support vector machine"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Support_vector_machine (visited on 12/26/2022).

[13] Andrea Torsello - Ca’ Foscari University. “Linear Classifiers slides”.

[14] Rishabh Misra. *"Support Vector Machines — Soft Margin Formulation and Kernel Trick"*. Towards Data Science. URL: https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe (visited on 12/26/2022).

[15] Drew Wilimitis. *"The Kernel Trick in Support Vector Classification"*. Towards Data Science. URL: https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f (visited on 12/26/2022).

[16] *"Multiclass Classification Using Support Vector Machines"*. baeldung. URL: https://www.baeldung.com/cs/svm-multiclass-classification (visited on 12/26/2022).

[17] *"Random forest"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Random_forest (visited on 12/26/2022).

[18] *"Ensemble learning"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Ensemble_learning (visited on 12/26/2022).

[19] *"Decision tree"*. scikit-learn. URL: https://scikit-learn.org/stable/modules/tree.html (visited on 12/26/2022).

[20] Yong Joseph Bakos. *"Decision Tree Classifier"*. Colorado School of Mines. URL: http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/lguo/decisionTree.html (visited on 12/26/2022).

[21] Andrea Torsello - Ca' Foscari University. "Decision trees slides".

[22] Karan Kashyap. *"Machine Learning- Decision Trees and Random Forest Classifiers"*. Analytics Vidhya. URL: https://medium.com/analytics-vidhya/machine-learning-decision-trees-and-random-forest-classifiers-81422887a544 (visited on 12/26/2022).

[23] *"Naive Bayes classifier"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Naive_Bayes_classifier (visited on 12/26/2022).

[24] *"k-nearest neighbors algorithm"*. Wikipedia. URL: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm (visited on 12/26/2022).

[25] *"Euclidean distance"*. Wikipedia. URL: https://en.wikipedia.org/wiki/Euclidean_distance (visited on 12/26/2022).