Università Ca'Foscari Venezia

Artificial Intelligence and Data Engineering

Foundation of Artificial Intelligence

Assignment 2: **Discriminative and Generative Classifiers**

Professor
Andrea Torsello

Autor
Davide Tonetto
Student ID 884585

Academic year
2023/2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Discriminative and Generative Classifiers 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 a 60000 training set and a 10000 test set.

Train the following classifiers on the dataset:

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

$$p(x_i|y) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x_i^{\alpha-1}(1-x_i)^{\beta-1}$$

4. 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 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 prediction.

1.2 Dataset

The MNIST dataset is composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into a 60000 training set and a 10000 test set. The images are labeled with the corresponding digit.

Here are some examples of images in the dataset with their labels:

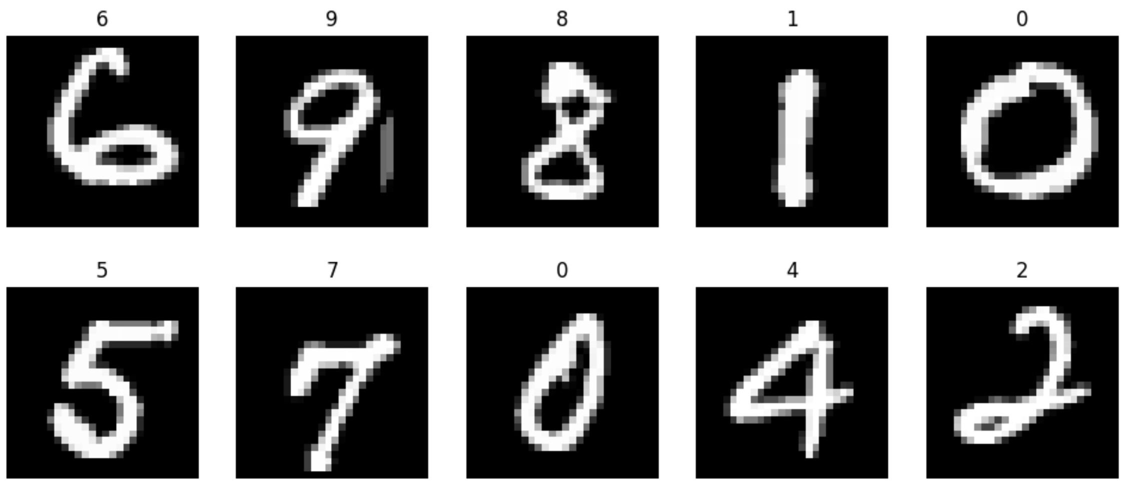


Figure 1.1: MNIST dataset examples.

1.3 Project structure

The project contains the following files:

1. **data_engineering.ipynb**: notebook used to explore the dataset and to construct the training and test sets;
2. **SVM_linear_classifier.ipynb**: notebook used to train and test the SVM classifier with the linear kernel;
3. **SVM_poly_classifier.ipynb**: notebook used to train and test the SVM classifier with the polynomial kernel;
4. **SVM_RBF_classifier.ipynb**: notebook used to train and test the SVM classifier with RBF kernel;
5. **RandomForest_classifier.ipynb**: notebook used to train and test the Random Forest classifier;
6. **naive_bayes_classifier.ipynb**: notebook used to train and test the Naive Bayes classifier with Beta distribution;
7. **KNN_classifier.ipynb**: notebook used to train and test the k-NN classifier;
8. **KNN.py**: python file containing the implementation of the k-NN classifier;
9. **naive_bayes.py**: python file containing the implementation of the Naive Bayes classifier with the Beta distribution.

Every notebook contains the code used to train and test the classifier, the results obtained and the comments on the results. The notebooks are structured in the following way:

1. **Imports**: import the libraries used;
2. **Data retrieving**: retrieve the data from the CSV files;
3. **Cross validation**: perform 10-fold cross-validation to find the best parameters for the classifier using a reduced training set;

4. **Training**: train the classifier on the whole training set with the best parameters found;
5. **Testing**: test the classifier on the test set;
6. **Results**: show the results obtained and the execution times;

1.4 Data engineering

In the **data_engineering.ipynb** notebook the dataset is explored and the training and test sets are constructed. Using the following code the dataset is loaded from the CSV files:

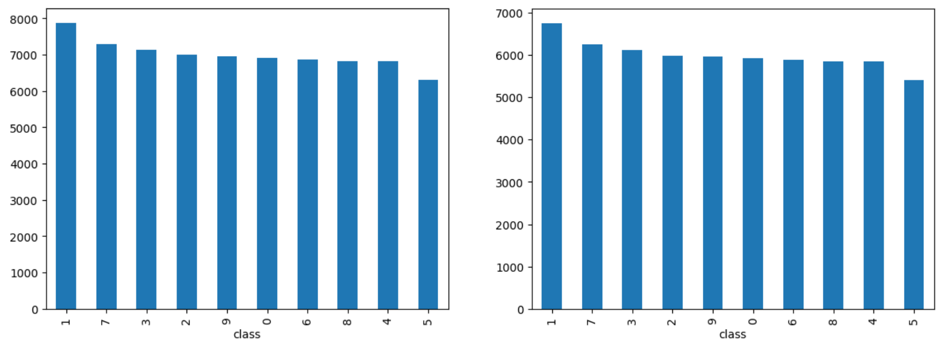
```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True,
                                                  parser='auto')
y = y.astype(int)
X = X / 255

# transform y to DataFrame with only one column
y = y.to_frame()
```

Then in the following code, the training and test sets are constructed using the **train_test_split** function from the **sklearn.model_selection** library:

```
# construct training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, y,
            test_size=10000, random_state=42, stratify=y)
```

As said before, the test set is composed of 10000 samples, while the training set is composed of 60000 samples.

The **stratify** parameter is used to ensure that the training and test sets have the same distribution of the labels. In the notebook, it is shown that the training and test sets have the same distribution of the labels using the following two histograms in Figure 1.2.



(a) Y distribution (b) Y distribution of train set

Figure 1.2: Y distribution comparison for full dataset and train set.

After that, the training set is split again in order to have a reduced training set to use for the cross-validation phase:

```
X_train_small, X_test_small, Y_train_small, Y_test_small =
train_test_split(X_train, Y_train, train_size=20000, test_size=6000,
random_state=42, stratify=Y_train)
```

The reduced training set is composed of 20000 samples, while the reduced test set is composed of 6000 samples. We will use the reduced training set to perform the cross-validation and find the best parameters for the classifiers because using the entire dataset for parameter tuning requires too much time to execute on the computer.

Finally, the training and test sets are saved in CSV files using the following code:

```
X_train.to_csv('./data/X_train.csv', index=False)
X_test.to_csv('./data/X_test.csv', index=False)
Y_train.to_csv('./data/Y_train.csv', index=False)
Y_test.to_csv('./data/Y_test.csv', index=False)

X_train_small.to_csv('./data/X_train_small.csv', index=False)
X_test_small.to_csv('./data/X_test_small.csv', index=False)
Y_train_small.to_csv('./data/Y_train_small.csv', index=False)
Y_test_small.to_csv('./data/Y_test_small.csv', index=False)
```

This is to use the same training and test sets for all the classifiers in the other notebooks.

Chapter 2

Supervised learning

2.1 Defining Supervised Learning

The task of supervised learning is the following one:

Definition 1. *Given a training set of N examples, each one consisting of a pair (x_i, y_i), where x_i is the input and y_i is the output, we want to find a function h that maps each input x_i to the correct output y_i.*

In other words, where each y_j was generated by an unknown function $y = f(x)$, we want to discover a function h that approximates the true function f as well as possible. The function h is called a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we give it a test set of examples that are distinct from the training set

We say that a learning algorithm **generalizes** if it performs accurately on examples that were not in the training set. The goal of learning is to find a hypothesis that generalizes well. Sometimes the function f is stochastic: it is not strictly a function of x, and what we have to learn is a conditional probability distribution, $P(Y|x)$.

We can distinguish two types of supervised learning problems (see Figure 2.1):

- **Regression**: the output y is a real number or a vector of real numbers. The goal is to predict a continuous-valued output.
- **Classification**: the output y is one of a discrete set of labels. The goal is to predict a discrete-valued output.

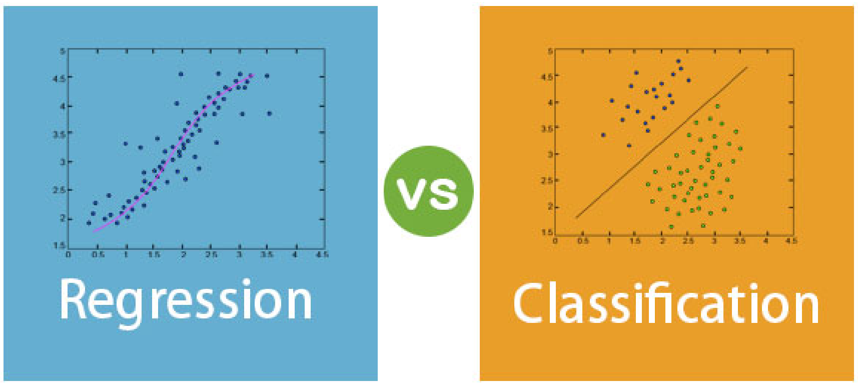


Figure 2.1: Regression vs Classification.

The problem of digit recognition is a classification problem because the output is a discrete set of labels (the digits from 0 to 9). So, we will use classification algorithms to solve this problem.

2.2 Discriminative vs Generative Models

Two main approaches can be used for classification: **discriminative** and **generative** models. Discriminative models learn the (hard or soft) boundary between classes, while generative models model the distribution of individual classes. Discriminative models are usually more powerful, but generative models are easier to train and more interpretable. Let's see the two approaches in detail.

2.2.1 Discriminative Models

Discriminative models directly learn the conditional probability $P(y|x)$, where y is the class label and x is the input.

In the case of classification, **discriminative models** learn the boundary between classes. Then they use this boundary to classify new samples, in other words, we aim to find output that maximizes the probability for the given input $y = \arg\max_y p(y|x)$.

The most common discriminative models are:

- **Logistic Regression**: it is a linear model that learns the probability of a sample belonging to a class. It is a discriminative model because it learns the boundary between classes.
- **Support Vector Machines**: they are discriminative models that learn the boundary between classes by maximizing the margin between the closest samples of different classes.
- **Neural Networks**: they are discriminative models that learn the boundary between classes by using a non-linear function to map the input to the output.
- **Decision Trees**: they are discriminative models that learn the boundary between classes by using a tree structure to split the input space.
- **Random Forests**: they are discriminative models that learn the boundary between classes by using an ensemble of decision trees.
- **K-Nearest Neighbors**: it is a discriminative model that learns the boundary between classes by using the majority vote of the k closest samples.

2.2.2 Generative Models

Generative models learn the joint probability $P(x|y)$, where y is the class label and x is the input. Then they use the Bayes rule to compute the conditional probability $P(y|x)$.

Definition 2 (Bayes rule)**.**

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{2.1}$$

In the case of classification, **generative models** learn the distribution of individual classes. Then they use this distribution to classify new samples, in other words, we aim to find output that maximizes the probability for the given input $y = \arg\max_y p(x|y)p(y)$.

The most common generative models are:

- **Naive Bayes**: it is a generative model that learns the distribution of individual classes by assuming that the features are independent.
- **Gaussian Mixture Models**: they are generative models that learn the distribution of individual classes by assuming that the features are normally distributed.

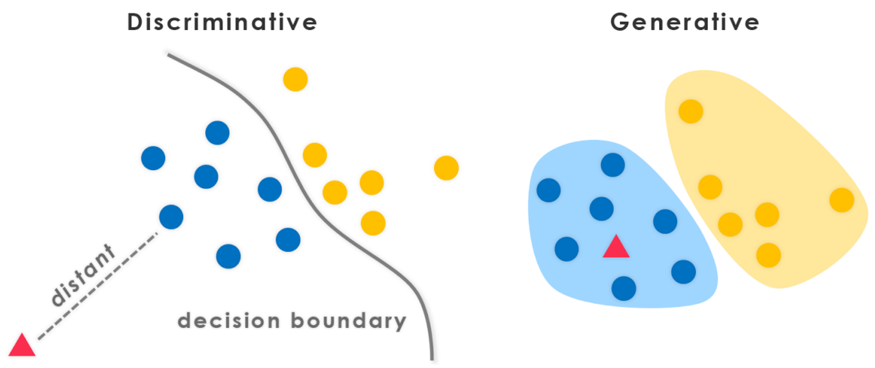


Figure 2.2: Generative vs Discriminative models.

Chapter 3

Random forest Classifier

3.1 Defining Random Forest

Random Forest is an **ensemble learning method** for classification and regression that operates by constructing a multitude of **decision trees** at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

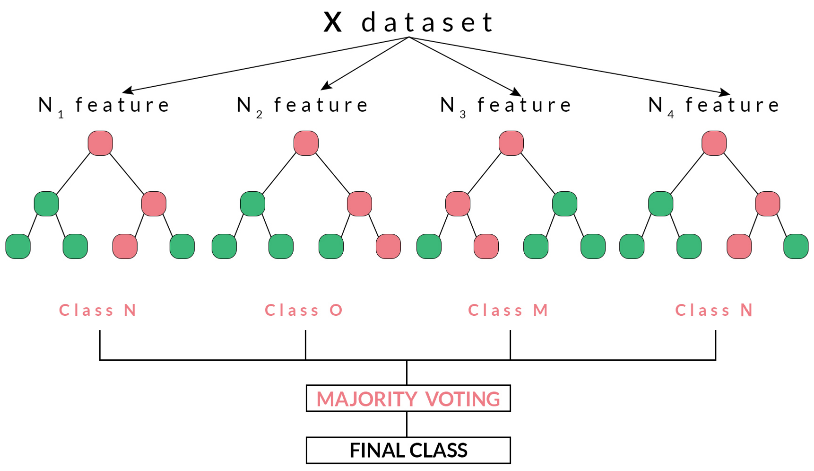


Figure 3.1: Random Forest Algorithm

In the next sections, we will see how this algorithm works starting from the definition of ensemble learning method and decision tree.

3.1.1 Ensemble learning method

Definition 3 (Ensemble learning method)**.** *The ensemble learning method is a technique that combines the predictions from multiple machine learning algorithms to make more accurate predictions than any individual model.*

An ensemble provides greater accuracy even if its weak learners are not very accurate. It's important to say that we are assuming models are independent but this is not always true. In fact, this is difficult because we train the models on the same/similar data. To solve this problem we can use **sampling**.

Finally, to combine the predictions of the weak learners, we can use **voting** or **averaging** for respectively classification and regression tasks.

3.1.2 Decision tree

Definition 4 (Decision tree). *A* ***decision tree*** *represents a function that takes as input a vector of attribute values and returns a "decision" (a single output value). The input vector can be represented as a set of feature-value pairs. The output of a decision tree is a single value (or a class label in the case of classification trees). The output value depends on the values of the input features and the particular structure of the tree.*

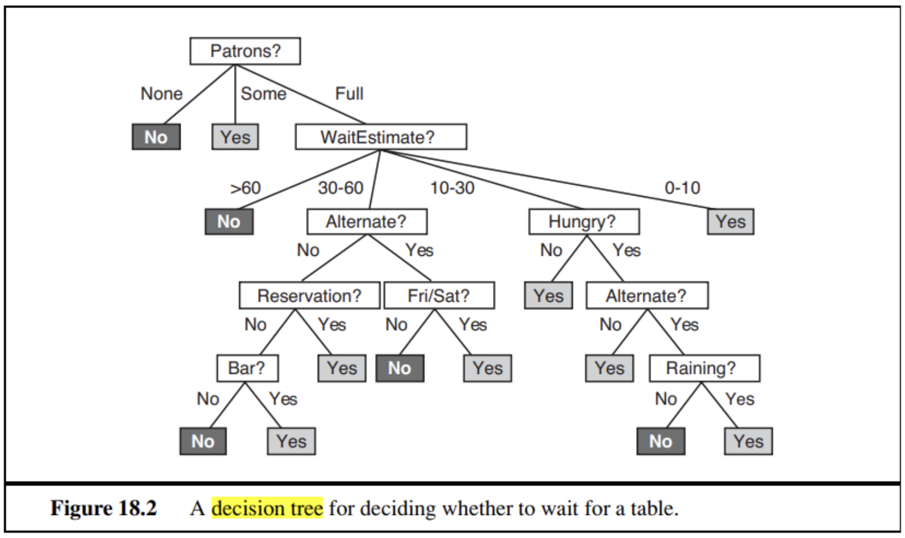


Figure 3.2: Decision tree example

As we can see in Figure 3.2, the decision tree is composed of:

- **Root node**: it represents the entire population or sample and this further gets divided into two or more homogeneous sets.
- **Splitting**: it is a process of dividing a node into two or more sub-nodes.
- **Decision node**: when a sub-node splits into further sub-nodes, then it is called the decision node.
- **Leaf/Terminal node**: nodes that do not split are called Leaf or Terminal nodes and represent the final decision.

Train a decision tree

In the training phase, we have to choose the best attribute to split the data. To do this, we can use different metrics such as **Gini index** and **Information gain**.

Definition 5 (Gini index). *GINI is a measure of statistical dispersion developed by the Italian statistician and sociologist Corrado Gini in 1912.*

$$Error(\mathcal{D}) = Gini(\mathcal{D}) = 1 - \sum_i p_i^2$$

- *Maximum:* $(1 - 1/m)$ *when records are equally distributed among all classes, implying the least interesting information*

- *Minimum:* 0.0 *when all records belong to one class, implying the most interesting information*

Definition 6 (Information gain)**.** *The error of a dataset is measured as the entropy of its label distributions*

$$Error(\mathcal{D}) = Info(\mathcal{D}) = -\sum_i p_i \log_2(p_i)$$

Where p_i is the probability/frequency of label i

- *Maximum:* $\log m$*, where m is the number of classes when records are equally distributed among all classes implying the least information*
- *Minimum:* 0.0 *when all records belong to one class, implying most information (assume* $0 \log 0 = 0$*)*

In general, assuming *Error* is an average measure, we denote the gain of a split as the error reduction concerning not splitting the node. Given a dataset $\mathcal{D}$ and an attribute $\mathcal{A}$, $\mathcal{A}$ can take values in $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$. Let $\mathcal{D}_i$ be the subset of $\mathcal{D}$ where $\mathcal{A} = v_i$. Then the gain of $\mathcal{A}$ is defined as:

$$Gain(\mathcal{D}, \mathcal{A}) = Error(\mathcal{D}) - \sum_{i=1}^{m} \frac{|\mathcal{D}_i|}{|\mathcal{D}|} Error(\mathcal{D}_i)$$

We would like $Gain > 0$, note that *Gain* cannot decrease.

Then we can use the **ID3 algorithm** to build a decision tree. The algorithm is a recursive algorithm that works as follows:

1. If all records in $\mathcal{D}$ belong to the same class c, then create a leaf node labeled with c and return it.
2. If $\mathcal{A} = \emptyset$, then create a leaf node labeled with the most frequent class in $\mathcal{D}$ and return it.
3. Otherwise, find the attribute $\mathcal{A}^*$ with the highest gain and create a decision node labeled with $\mathcal{A}^*$.
4. For each value v_i of $\mathcal{A}^*$, create a branch from the decision node and let $\mathcal{D}_i$ be the subset of $\mathcal{D}$ where $\mathcal{A}^* = v_i$. Recursively call the algorithm on $\mathcal{D}_i$ to create a subtree.
5. Return the decision node.

Prediction with a decision tree

In the prediction phase, we have to follow the path from the root node to a leaf node. The label of the leaf node is the prediction.

Depth of a decision tree

Decision trees can be very deep, but this is not always a good thing. In fact, a deep tree can overfit the training data. To avoid this problem, we can limit the depth of the tree. In this way, we can control the complexity of the model.

3.1.3 Random forest algorithm

As said before, Random forest is an ensemble learning method that combines multiple decision trees to make a more accurate prediction. The basic idea behind this is to combine multiple **unique** decision trees in determining the final output rather than relying on individual decision trees.

In the training phase, the algorithm for a predefined number of times (the number of trees) does the following:

1. Randomly selects k features from total m features where $k < m$.
2. Sample n training examples from the given train set with replacement.
3. Build a decision tree based on the k features and n training examples.

In the prediction phase, the algorithm does the following:

1. Takes the test features and uses the rules of each randomly created decision tree to predict the outcome and stores the predicted outcome (target).
2. Calculate the votes for each predicted target.
3. Consider the high-voted predicted target as the final prediction from the random forest algorithm.

The main aspect of the random forest algorithm is that a large number of relatively uncorrelated models (trees). The low correlation between models is the key.

3.2 RF with MNIST dataset

In the **RandomForest_classifier.ipynb** notebook the Random Forest classifier is trained and tested on the MNIST dataset.

We used the **scikit-learn** library to train and test the classifier. The library provides the **RandomForestClassifier** class that implements the Random Forest algorithm.

3.2.1 Parameter tuning

The Random Forest classifier has many parameters that can be tuned. Using the **GridSearchCV** class of the **scikit-learn** library we performed a 10-fold cross-validation to find the best parameters for the classifier. The parameters that we tuned are:

- **max_features**: the number of trees in the forest;
- **max_depth**: the maximum depth of the tree;

- **criterion**: the function to measure the quality of a split;
- **max_features**: the number of features to consider when looking for the best split.

with the following values:

```
{
    'n_estimators': [100, 200, 500, 700, 1000],
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 20, 60],
    'max_features': [None, 'sqrt', 'log2']
}
```

The best parameters found are:

```
{
    'criterion': 'entropy',
    'max_depth': 60,
    'max_features': 'sqrt',
    'n_estimators': 500
}
```

3.2.2 Results

The Random Forest classifier has achieved an accuracy of 96.8% on the test set. The confusion matrix is shown in Figure 3.3. The confusion matrix shows that the classifier has made a few mistakes. In fact, most of the samples are classified correctly.

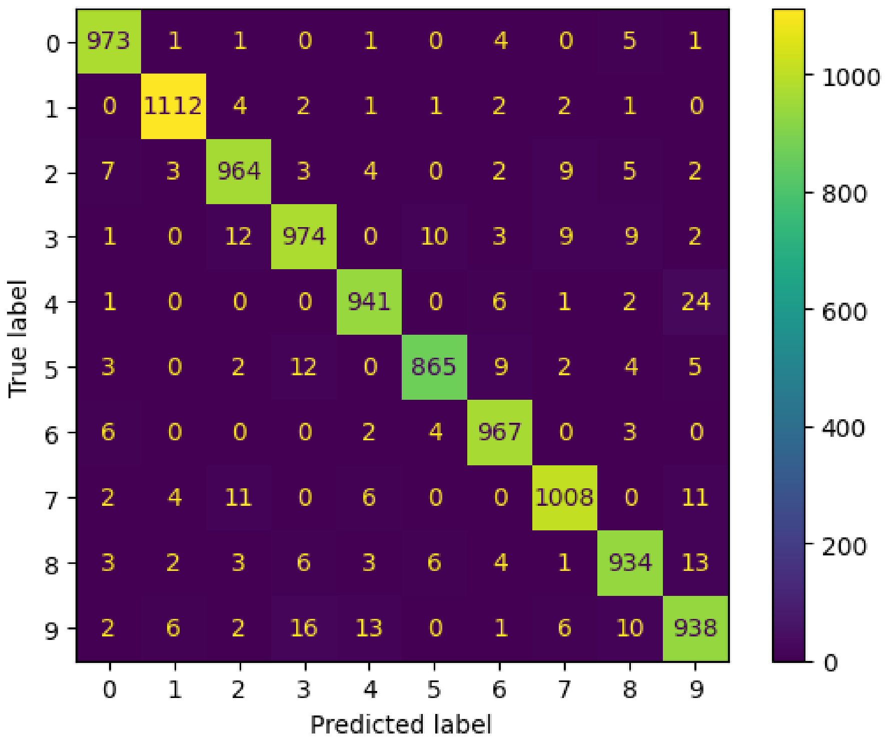


Figure 3.3: Random Forest confusion matrix.

3.2.3 Execution time

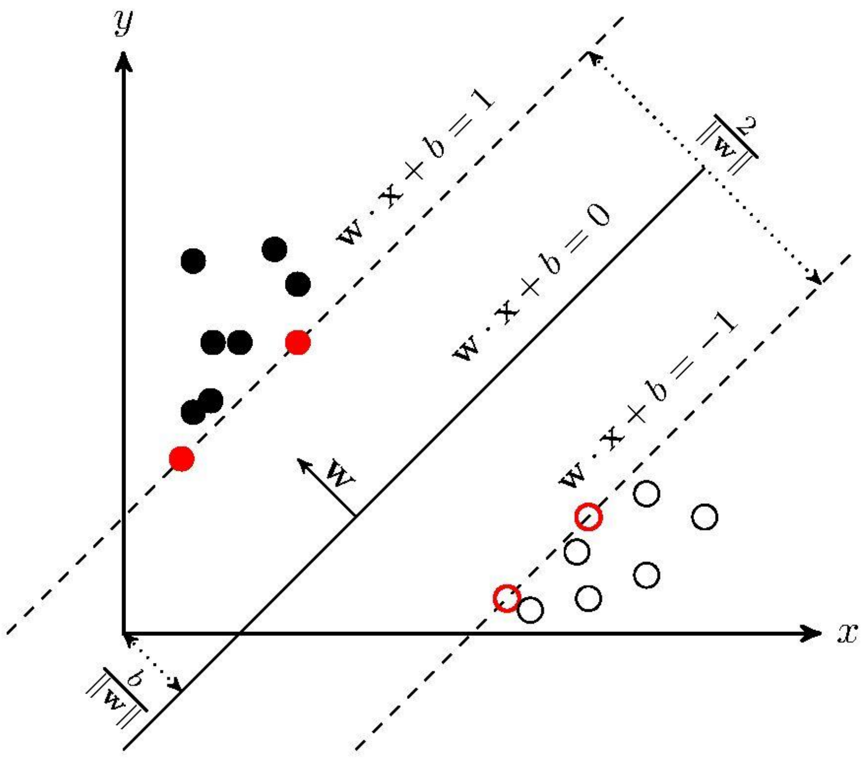
To train the Random Forest classifier on the entire training set with the best parameters found, it takes about 4 minutes. To test the classifier on the test set, the model takes about 1.4 seconds. The training time is much higher than the testing time because the training phase is more computationally expensive than the prediction phase. After all, the algorithm must create "n_estimators" decision trees.

Chapter 4

Support Vector Machine

4.1 Defining Support Vector Machine

The **Support Vector Machine** (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems. In this chapter, we will focus on the classification task.



y
x
$\mathbf{w}\cdot\mathbf{x}+b=1$
$\mathbf{w}\cdot\mathbf{x}+b=0$
$\mathbf{w}\cdot\mathbf{x}+b=-1$
$\frac{2}{\|\mathbf{w}\|}$
$\mathbf{w}$
$\frac{b}{\|\mathbf{w}\|}$

Figure 4.1: Support Vector Machine

There are three important properties of SVMs:

- **Maximal margin separator**: SVMs construct a maximum margin separator. This means that the algorithm creates a decision boundary that is as far as possible from example points. This helps them generalize well.
- **Kernel trick**: SVMs create a linear separating hyperplane in the input space. However, in many cases, the data is not linearly separable in the input space. In such cases, SVMs use the kernel trick to transform the input space to a higher-dimensional feature space where the data is linearly separable.

- **Nonparametric model**: SVMs are nonparametric models. In practice, they often end up retaining only a small fraction of the number of examples, sometimes as few as a small constant times the number of dimensions.

4.1.1 How does SVM work?

In the previous section, we said that SVMs create a linear separating hyperplane in the input space. In this section, we will see how this is possible.

Let's consider a dataset with two classes, as shown in Figure 4.2. The goal of SVM is to find the best-separating hyperplane that maximizes the margin between the two classes. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane (which are called **support vectors**).

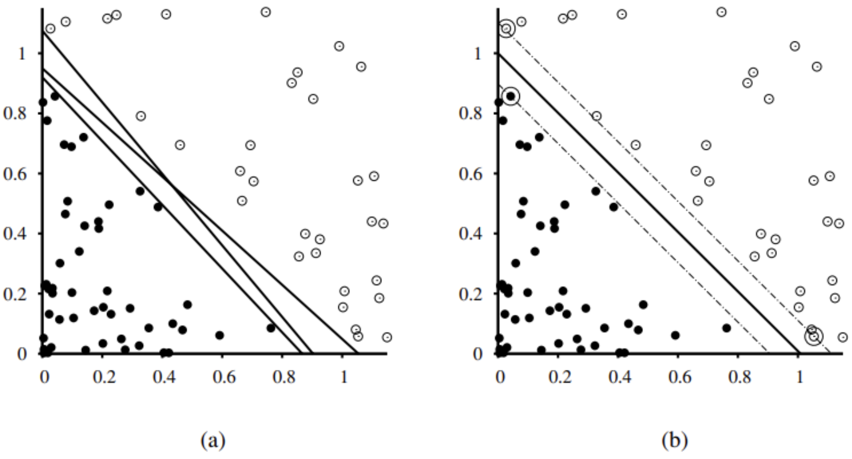


Figure 4.2: Support Vector Machine classification. (a) Three candidate linear separators for the two classes. (b) The maximum margin separator and support vectors are highlighted.

In Figure 4.2-(a), we can see three candidate linear separators for the two classes and all of them correctly classify the training samples. However, the maximum margin separator (shown in 4.2-(b)) is the best choice because it will generalize better on the test data. This is because the maximum margin separator is the one that is farthest from the training samples.

Before going into mathematical details, it is necessary to introduce the concepts of **functional margin** and **geometric margin**.

Definition 7 (Functional margin). *The functional margin of a hyperplane concerning a labeled training example $(x^{(i)}, y^{(i)})$ is defined as:*

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

When the functional margin is positive, the example is correctly classified. The functional margin is negative when the example is misclassified. The magnitude of the functional margin is inversely proportional to the distance of the example from the hyperplane.

Definition 8 (Geometric margin). *The geometric margin of a hyperplane concerning a labeled training example $(x^{(i)}, y^{(i)})$ is defined as:*

$$\gamma^{(i)} = \frac{\hat{\gamma}^{(i)}}{||w||}$$

The geometric margin is the distance of the example from the hyperplane. The geometric margin is always positive, even when the example is misclassified.

The idea behind SVMs is to find a hyperplane with the **maximum geometric margin**. This is because, the larger the margin, the lower the generalization error of the classifier. So, an SVM solves the following optimization problem:

$$\begin{aligned} \underset{\gamma,w,b}{\text{minimize}} \quad & \frac{1}{2}||w||^2 \\ \text{subject to} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \dots, m \end{aligned}$$

The constraints in the above optimization problem ensure that all the training examples are correctly classified by the hyperplane. The optimization problem is solved using the Lagrange multipliers. The Lagrangian of the above optimization problem is:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]$$

Setting the derivatives of $L(w, b, \alpha)$ with respect to w and b to zero, we get:

$$\begin{aligned} w &= \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \\ 0 &= \sum_{i=1}^{m} \alpha_i y^{(i)} \end{aligned}$$

Substituting the above equations in the Lagrangian, we get the dual form of the optimization problem:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} x^{(i)T} x^{(j)} \\ \text{subject to} \quad & \alpha_i \geq 0, \quad i = 1, \dots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \end{aligned}$$

The advantage of using this formulation is that only support vectors will have non-zero α_i values. So the SVM will only depend on a few training examples (support vectors). This is why SVMs are called nonparametric models.

Finally, given an unknown vector u, the prediction of class $(-1, 1)$ is made by:

$$h(u) = sign\left(\sum_{i=1}^{k} \alpha_i y^{(i)}(x^{(i)}u)\right)$$

The sum over k is over all the support vectors. The system has a unique optimal solution that can be found using quadratic programming or gradient ascent.

4.1.2 How does SVM handle non-linearly separable data?

SVM can handle also non-linearly separable data. We can identify two cases:

- **Nearly separable data**: In this case, we can use a soft margin classifier. This means that we allow some examples to be misclassified.
- **Non-separable data**: In this case, we can use the kernel trick. This means that we transform the input space to a higher-dimensional feature space where the data is linearly separable.

Soft margin

In the previous section, we saw that SVMs create a maximum margin separator. However, in many cases, the data is not linearly separable. In such cases, SVMs use a soft margin classifier. This means that we allow some examples to be misclassified.

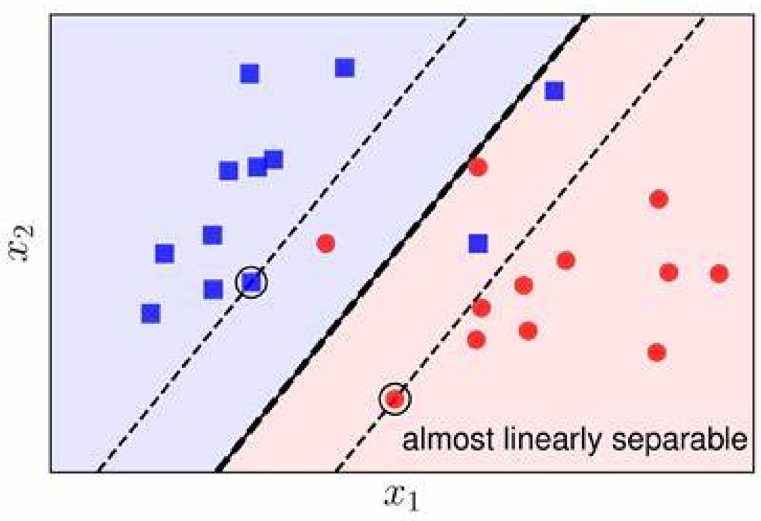


Figure 4.3: Soft margin SVM. A linearly separable dataset with outliers and a soft margin.

In Figure 4.3, we can see a linearly separable dataset with outliers and a soft margin. The soft margin classifier allows some examples to be misclassified. The goal of the soft margin classifier is to find a hyperplane that maximizes the margin and minimizes the number of misclassified examples. This is done by introducing a slack variable $\xi^{(i)}$ for each example $(x^{(i)}, y^{(i)})$. The slack variable $\xi^{(i)}$ measures the degree of misclassification of the example $(x^{(i)}, y^{(i)})$. The optimization problem of the soft margin classifier is:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi^{(i)} \\ \text{subject to} \quad & y^{(i)}(w \cdot x^{(i)}) \geq 1 - \xi^{(i)}, \quad i = 1, \ldots, m \\ & \xi^{(i)} \geq 0, \quad i = 1, \ldots, m \end{aligned}$$

Where C is a hyperparameter that controls how strictly the margin must be enforced. The larger the value of C, the more strictly the margin is enforced. The smaller the value of C, the more misclassified examples are allowed.

Kernel trick

In the previous section, we saw that SVMs use a soft margin classifier when the data is not linearly separable. However, in many cases, the data is not linearly separable even with a soft margin classifier. In such cases, SVMs use the kernel trick. This means that we transform the input space to a higher-dimensional feature space where the data is linearly separable.

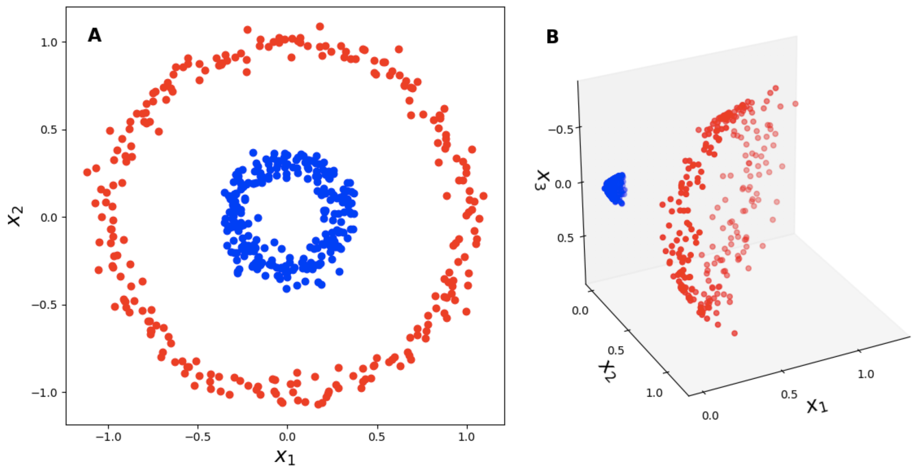


Figure 4.4: Kernel trick. A non-linearly separable dataset and a linearly separable dataset in a higher-dimensional feature space.

In Figure 4.4, we can see a non-linearly separable dataset and a linearly separable dataset in a higher-dimensional feature space. The goal of the kernel trick is to find a function ϕ that maps the input space to a higher-dimensional feature space. The SVM then works in this higher-dimensional feature space to find a linear separating hyperplane by using a function called **kernel function** that satisfy the following property:

$$K(x, z) = \phi(x) \cdot \phi(z)$$

The kernel function is chosen based on the dataset. Some commonly used kernel functions are:

- **Linear kernel**: $K(x, z) = x \cdot z$
- **Polynomial kernel**: $K(x, z) = (x \cdot z + 1)^d$

- **Gaussian Radial Basis Function (RBF) kernel**: $K(x,z) = \exp\left(-\frac{||x-z||^2}{2\sigma^2}\right)$

4.2 SVM with MNIST dataset

We tried to use SVM with the MNIST dataset. We used the `sklearn.svm.SVC` class. We used the linear kernel, the polynomial kernel of degree 2 and the Gaussian RBF kernel, respectively in the following Jupyter notebooks:

- `SVM_linear_classifier.ipynb`
- `SVM_poly_classifier.ipynb`
- `SVM_RBF_classifier.ipynb`

4.2.1 Parameter tuning

Using the **GridSearchCV** class of the **scikit-learn** library we performed a 10-fold cross-validation to find the best parameters for each classifier. The parameters that we tuned are:

- **C**: hyperparameter that controls how strictly the margin must be enforced. The larger the value of C, the more strictly the margin is enforced. The smaller the value of C, the more misclassified examples are allowed.
- **gamma**: the kernel coefficient for the RBF and polynomial kernels.

with the following values:

```
{
    'C': [0.01, 0.05, 0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.01, 0.1, 1],
}
```

The best parameters found are:

- **Linear kernel**: C = 0.1.
- **Polynomial kernel of degree 2**: C = 0.1, gamma = 0.1.
- **Gaussian RBF kernel**: C = 100, gamma = 'scale'.

Note that if gamma = 'scale' (default) is passed then it uses $\frac{1}{n\_features \cdot X.var()}$ as the value of gamma.

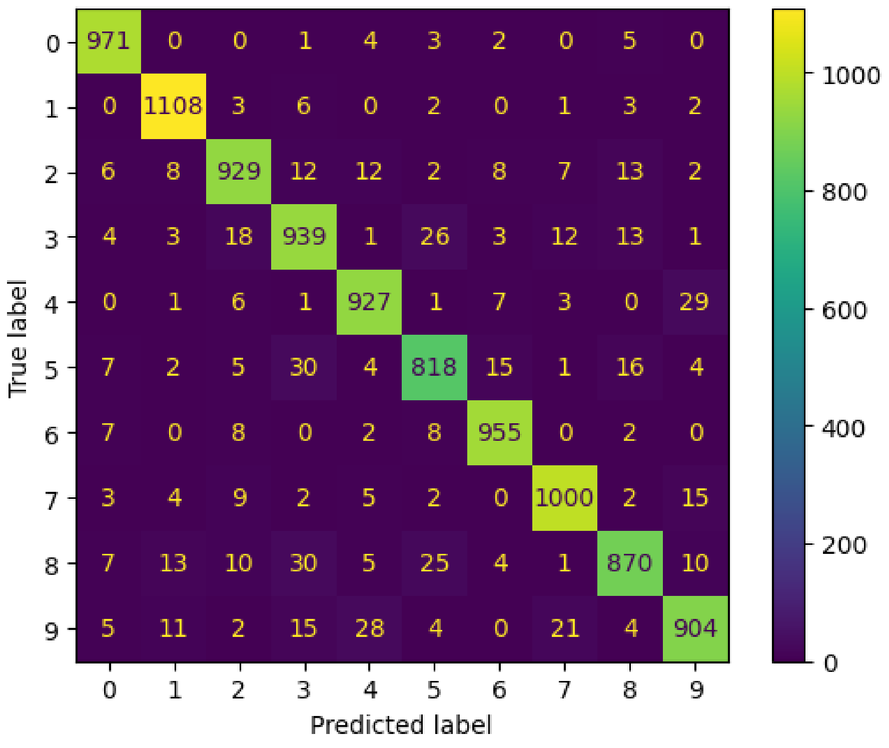
4.2.2 Results

We achieved the following results:

Linear kernel

The SVM with a linear kernel has achieved an accuracy of 94.2% on the test set in 26.47 seconds. It takes 111.37 seconds to train the model on the entire dataset.

The confusion matrix is shown in Figure 4.5. The confusion matrix shows that the classifier has made a few mistakes. In fact, most of the samples are classified correctly.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	971	0	0	1	4	3	2	0	5	0
1	0	1108	3	6	0	2	0	1	3	2
2	6	8	929	12	12	2	8	7	13	2
3	4	3	18	939	1	26	3	12	13	1
4	0	1	6	1	927	1	7	3	0	29
5	7	2	5	30	4	818	15	1	16	4
6	7	0	8	0	2	8	955	0	2	0
7	3	4	9	2	5	2	0	1000	2	15
8	7	13	10	30	5	25	4	1	870	10
9	5	11	2	15	28	4	0	21	4	904

Figure 4.5: SVM with linear kernel confusion matrix.

Polynomial kernel

The SVM with a polynomial kernel of degree 2 has achieved an accuracy of 98% on the test set in 19.17 seconds. It takes 94.48 seconds to train the model on the entire dataset.

The confusion matrix is shown in Figure 4.6. The confusion matrix shows that the classifier has made a few mistakes. In fact, most of the samples are classified correctly.

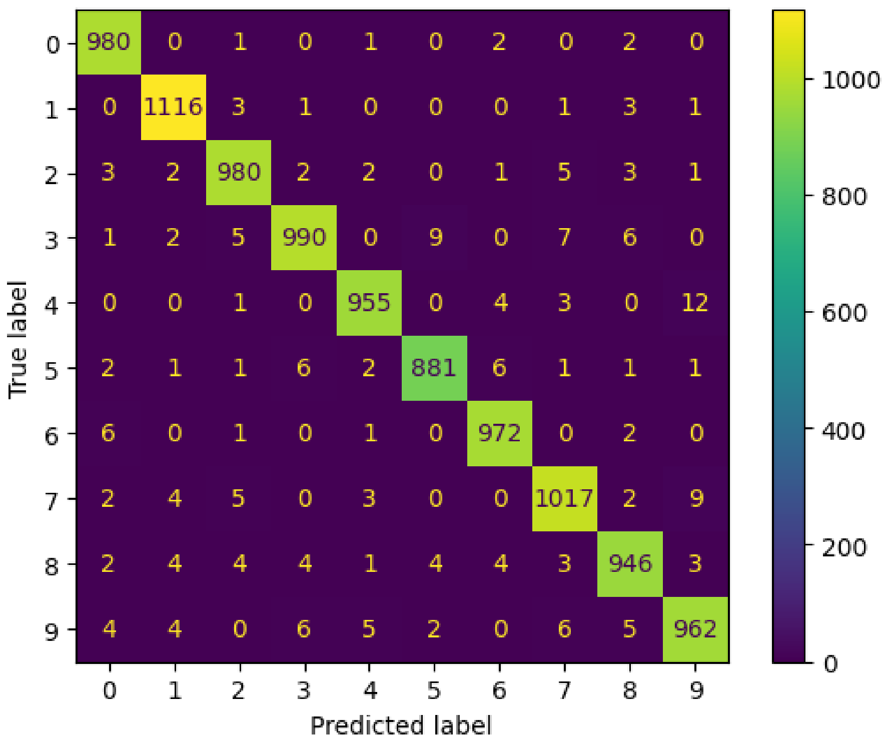


Figure 4.6: SVM with polynomial kernel confusion matrix.

Gaussian RBF kernel

The SVM with an RBF kernel has achieved an accuracy of 98.4% on the test set in 45.62 seconds. It takes 161.53 seconds to train the model on the entire dataset.

The confusion matrix is shown in Figure 4.7. The confusion matrix shows that the classifier has made a few mistakes. In fact, most of the samples are classified correctly.

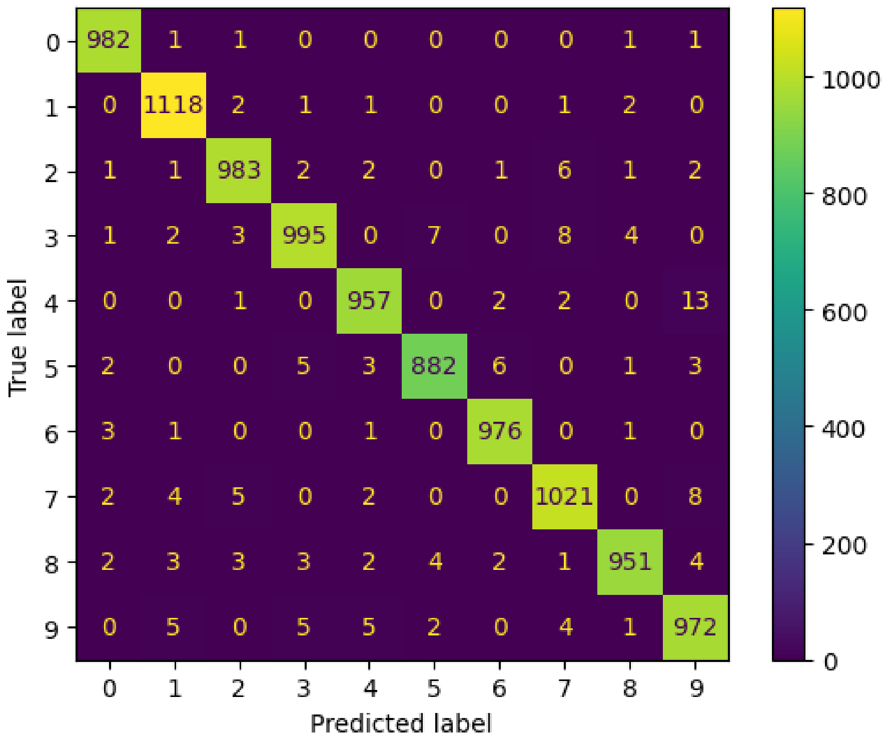


Figure 4.7: SVM with RBF kernel confusion matrix.

Chapter 5

K-Nearest Neighbors

5.1 Defining K-Nearest Neighbors

The K-Nearest Neighbors algorithm is a supervised learning algorithm that can be used to solve both classification and regression problems. It is a non-parametric algorithm, which means that it does not make any assumption on the underlying data distribution.

5.1.1 How does it work?

The algorithm is based on the idea that similar data points are close to each other. In other words, data points that have similar features are close to each other.

Given a new point to be classified the algorithm looks for the K closest points to it and assigns the class that is most frequent among them. In Figure 5.1 the new point star would be classified as B for k=3 and as A for k=6.

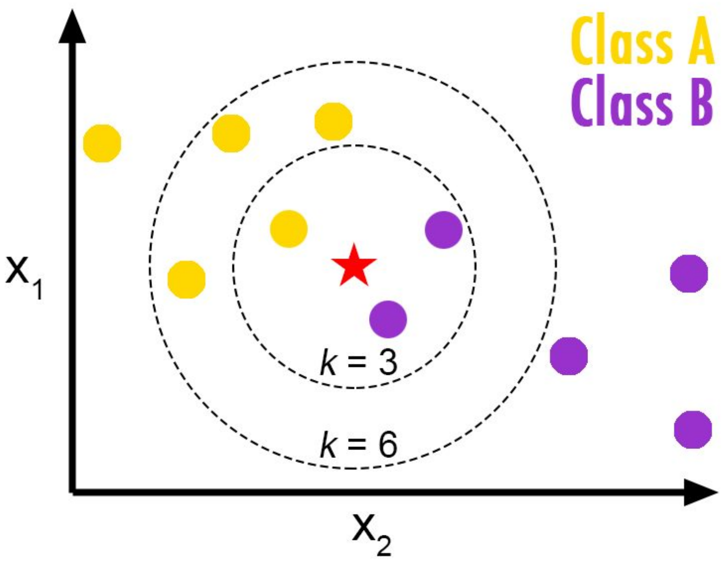


Figure 5.1: K-NN concept.

To compute the distance between two points we can use the Euclidean distance, which is defined as:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

where x and y are two points in the n-dimensional space. But we can also use other distance metrics such as the Manhattan distance or the Minkowski distance.

5.1.2 Python implementation

In the **KNN.py** file we can find the implementation of the K-NN algorithm. The class **KNN** has two methods: **fit** and **predict**. Let's see how they work.

fit

```python
def fit(self, X, y):
    # Check that X and y have the correct shape
    X, y = check_X_y(X, y)
    check_classification_targets(y)

    # Store the classes seen during fit
    self.classes_ = unique_labels(y)

    self.X_ = X.to_numpy() if isinstance(X, pd.DataFrame) else X
    self.y_ = y.to_numpy() if isinstance(y, pd.DataFrame) else y

    # n features in
    self.n_features_in_ = self.X_.shape[1]

    return self
```

The **fit** method takes as input the training data and the labels. It checks that the data has the correct shape and stores the classes seen. Then it stores the training data and the labels in the **X_** and **y_** variables. Finally, it stores the number of features in the **n_features_in_** variable and returns the object itself.

predict

```python
def predict(self, X):
    # check is fit had been called
    check_is_fitted(self)

    # Input validation
    check_array(X)

    # transform X into numpy array
    X = X.to_numpy() if isinstance(X, pd.DataFrame) else X

    closest = np.argpartition(
        euclidean_distances(X, self.X_),
        axis=1,
        kth=self.k
    )[:, :self.k]

    # foreach k-ple of nearest neighbors, get the most frequent label
    y_pred = np.apply_along_axis(
        lambda x: np.bincount(self.y_[x]).argmax(),
        axis=1,
        arr=closest
    )

    return y_pred
```

The **predict** method takes as input the data to be classified and returns the predicted labels. It checks that the **fit** method has been called and that the input data has the correct shape. With the **euclidean_distances** function it computes the distance between each point in the input data and each point in the training data. Then, using the **argpartition** function, it finds the k closest points for each point in the input data. Finally, for each k-ple of nearest neighbors, it gets the most frequent label and returns the predicted labels.

5.2 K-NN with MNIST dataset

In the **KNN_classifier.ipynb** notebook the K-NN classifier is trained and tested on the MNIST dataset using the **KNN** class defined in the **KNN.py** file.

5.2.1 Parameter tuning

The K-NN classifier has only one parameter that can be tuned: the number of neighbors **k**. Using the **GridSearchCV** class of the **scikit-learn** library we performed a 10-fold cross-validation to find the best value for **k**.

The best value for **k** is 3.

5.2.2 Results

The accuracy of the classifier on the test set is 97.4%. Follows the confusion matrix.

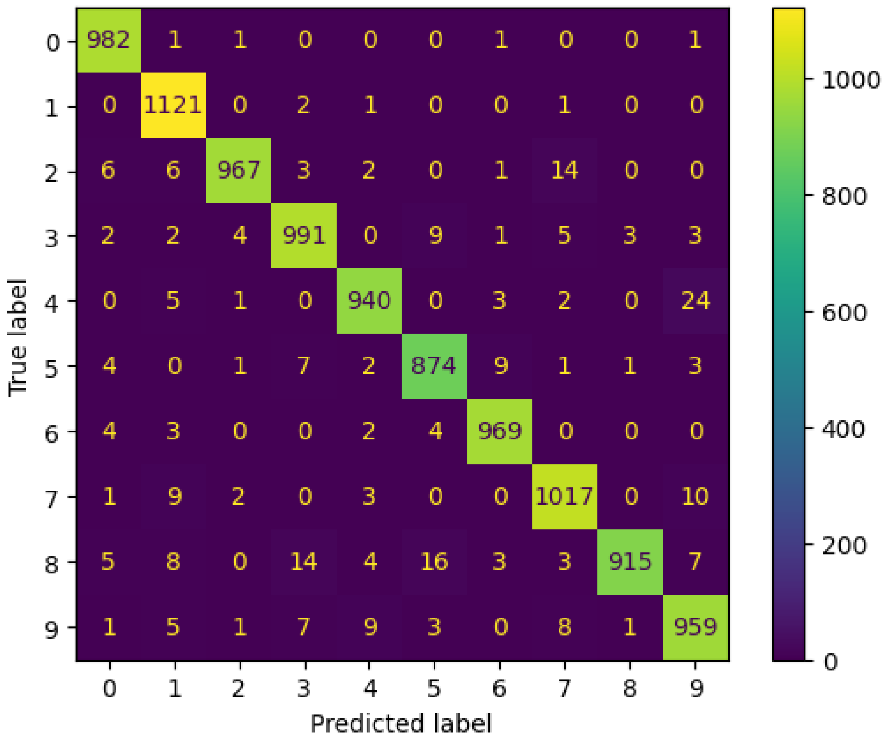


Figure 5.2: K-NN confusion matrix.

5.2.3 Execution time

To train the K-NN classifier on the entire training set it takes about 0.19 seconds. On the other hand, to test the classifier on the test set, the model takes about 17.55 seconds. The training time is much lower than the testing time because the training phase is less computationally expensive than the prediction phase. After all, the algorithm must only store the training data. Otherwise, during the prediction phase, the algorithm must compute the distance between each point in the test set and each point in the training set.

Chapter 6

Naive Bayes

6.1 Defining Naive Bayes

Naive Bayes is a generative model that uses Bayes' theorem to classify data. It is called "**naive**" because it is based on the assumption that the features are independent of each other, given the class.

Naive Bayes is a probabilistic classifier, which means that it predicts the probability of each class for a given input data. Then, the class with the highest probability is assigned to the input.

6.1.1 Bayes' theorem

Bayes' theorem is a formula that allows to compute the conditional probability of an event, given the occurrence of another event. It is defined as:

Definition 9 (Bayes' theorem)**.** *Let $A_1, \dots, A_k$ be a collection of mutually exclusive and exhaustive events with $P(A_i) > 0$ for $i = 1, \dots, k$. Then for any other event B for which $P(B) > 0$ we have:*

$$P(A_j|B) = \frac{P(A_j \cap B)}{P(B)} = \frac{P(A_j)P(B|A_j)}{\sum_{i=1}^{k} P(A_i)P(B|A_i)} \quad \text{for } j = 1, \dots, k$$

6.1.2 How does it work?

In the training phase, the algorithm computes the prior probability of each class and the conditional probability of each feature given the class. Therefore, in the presence of multiple classes $C_1, \dots, C_m$ and multiple features $X_1, \dots, X_n$, we have that the prediction of our model is:

$$\arg\max_{C_i} P(C_i|X) = P(x_1|C_i) \cdot P(x_2|C_i) \cdot \ldots P(x_f|C_i) \frac{P(C_i)}{P(X)}$$

Note that $P(X)$ does not change the ranking of the different classes C_i, and therefore it can be removed for classification purposes.

Nicely enough, $P(x_j|C_i)$ can be pre-computed from the dataset by assuming the distribution of the features. Some common distributions are:

- **Multinomial Naive Byes**: if feature f_j is categorical then $P(x_j|C_i)$ is the number of instances in the dataset of class C_i whose j-the features has the same values as X, divided by the cardinality of C_i.

- **Gaussian Naive Byes**: if feature f_j is numerical then we assume the feature has Gaussian distribution and, after computing mean μ and standard deviation σ from the dataset, we have:

$$P(x_j|C_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}}$$

6.1.3 Python implementation

In the **naive_bayes.py** file we can find the implementation of the Naive Bayes algorithm. The class **BetaNaiveBayes** has two methods: **fit** and **predict**. Let's see how they work.

It's important to denote that the implementation of the algorithm is based on the assumption that each pixel follows a **Beta distribution** of parameters α and β.

Definition 10 (Beta distribution)**.** *In probability theory and statistics, the beta distribution is a family of continuous probability distributions defined on the interval* $[0, 1]$ *or* $(0, 1)$ *in terms of two positive parameters, denoted by alpha (α) and beta (β), that appear as exponents of the variable and its complement to 1, respectively, and control the shape of the distribution.*[1]

$$p(x_i|y) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x_i^{\alpha-1}(1-x_i)^{\beta-1} \quad \textit{with } \alpha, \beta > 0$$

The parameters α and β are estimated using the method of moments giving the following formulas:

$$\alpha = KE(X)$$

$$\beta = K(1 - E(X))$$

$$K = \frac{E(X)(1 - E(X))}{Var(X)} - 1$$

As we can notice, we may have a division by zero if $Var(X) = 0$. To avoid this problem we transform the variance to $1e-6$ in case $Var(X) = 0$. Moreover, in some cases, we may have $\alpha \leq 0$ or $\beta \leq 0$. To avoid this problem we transform α and β to $1e-6$ in case $\alpha \leq 0$ or $\beta \leq 0$.

[1]`https://en.wikipedia.org/wiki/Beta_distribution`

fit

```
def fit(self, X, y):
    # Check that X and y have the correct shape
    X, y = check_X_y(X, y)
    check_classification_targets(y)

    self.X_ = X.to_numpy() if isinstance(X, pd.DataFrame) else X
    self.y_ = y.to_numpy() if isinstance(y, pd.DataFrame) else y
    self.classes_ = unique_labels(y)
    self.n_classes_ = len(self.classes_)
    self.n_features_ = self.X_.shape[1]
    self.n_samples_ = self.X_.shape[0]
    self.prior_ = np.bincount(self.y_) / self.n_samples_

    # compute alpha and beta for each class
    self.alpha_ = []
    self.beta_ = []
    for i, c in enumerate(self.classes_):
        X_c = self.X_[self.y_ == c]
        mean = np.mean(X_c, axis=0)
        var = np.var(X_c, axis=0)

        # compute alpha and beta, avoid division by zero
        var[var == 0] = 1e-6

        k = ((mean * (1 - mean)) / var) - 1
        alpha = mean * k
        beta = (1 - mean) * k

        # store alpha and beta
        self.alpha_.append(alpha)
        self.beta_.append(beta)

    return self
```

The **fit** method takes as input the training data and the labels. It checks that the data has the correct shape and stores the classes seen. Then it stores the training data and the labels in the **X_** and **y_** variables. Finally, it stores the number of classes, features and samples in the **n_classes_**, **n_features_** and **n_samples_** variables. It also computes the prior probability ($\frac{P(C_i)}{P(X)}$) of each class and the parameters α and β for each class.

predict

```
def predict(self, X):
    # check is fit had been called
    check_is_fitted(self)

    # Input validation
    check_array(X)

    # transform X into numpy array
    X = X.to_numpy() if isinstance(X, pd.DataFrame) else X

    # compute the probability of each class for each sample
    y_pred = []
    for x in X:
        prob = []
        for i, c in enumerate(self.classes_):
            probs =
            beta.cdf(x + self.epsilon, self.alpha_[i], self.beta_[i])
            - beta.cdf(x - self.epsilon, self.alpha_[i], self.beta_[i])

            np.nan_to_num(probs, copy=False, nan=1.0)
            prob.append(np.prod(probs) * self.prior_[i])

        # get the class with the highest probability
        y_pred.append(np.argmax(prob, axis=0))

    return y_pred
```

The **predict** method takes as input the test data returns the predicted labels and checks that the data has the correct shape. Then, for each sample, it computes the probability of each class and stores the class with the highest probability in the **y_pred** variable. Finally, it returns the **y_pred** variable.

To compute the probability of each class for each sample ($\prod_{j=1}^{n} P(x_j|C_i)$) we use the following integral:

$$\int_{x_j-\epsilon}^{x_j+\epsilon} \frac{\Gamma(\alpha_i+\beta_i)}{\Gamma(\alpha_i)\Gamma(\beta_i)} x_j^{\alpha_i-1}(1-x_j)^{\beta_i-1}dx \quad \text{for } j = 1,\ldots,n$$

That corresponds to the following formula:

$$P(x_j|C_i) = \text{Beta.CDF}(x_j+\epsilon,\alpha_i,\beta_i) - \text{Beta.CDF}(x_j-\epsilon,\alpha_i,\beta_i) \quad \text{for } j = 1,\ldots,n$$

Finally, we compute the product of the probabilities of each feature for each class and multiply it by the prior probability of the class obtaining the probability of each class for each sample.

$$\prod_{j=1}^{n} P(x_j|C_i) \cdot \frac{P(C_i)}{P(X)} \quad \text{for } i = 1,\ldots,m$$

6.2 Naive Bayes with MNIST dataset

In the **naive_bayes_classifier.ipynb** notebook the Naive Bayes classifier is trained and tested on the MNIST dataset using the **BetaNaiveBayes** class defined in the **naive_bayes.py** file.

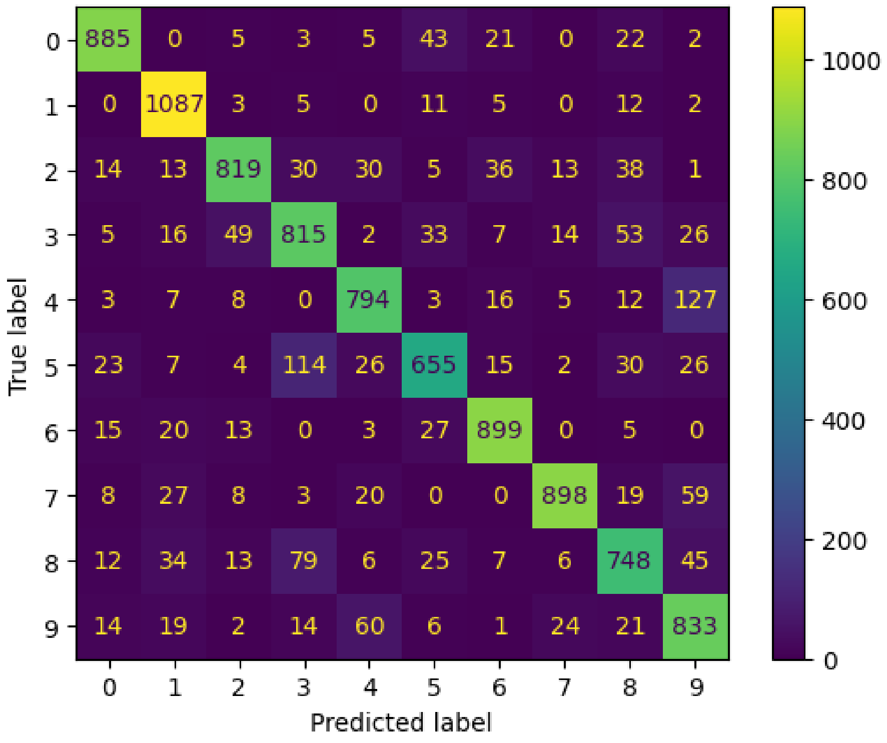
6.2.1 parameter tuning

The Naive Bayes classifier has only one parameter that can be tuned: the epsilon ϵ. Using the **GridSearchCV** class of the **scikit-learn** library we performed a 10-fold cross-validation to find the best value for ϵ.

The best value for ϵ is 0.1.

6.3 Results

The accuracy of the classifiers on the test set is 84.3%. Follows the confusion matrix.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	885	0	5	3	5	43	21	0	22	2
1	0	1087	3	5	0	11	5	0	12	2
2	14	13	819	30	30	5	36	13	38	1
3	5	16	49	815	2	33	7	14	53	26
4	3	7	8	0	794	3	16	5	12	127
5	23	7	4	114	26	655	15	2	30	26
6	15	20	13	0	3	27	899	0	5	0
7	8	27	8	3	20	0	0	898	19	59
8	12	34	13	79	6	25	7	6	748	45
9	14	19	2	14	60	6	1	24	21	833

Figure 6.1: Naive Bayes confusion matrix.

Furthermore, in Figure 6.2 we can see the mean of each pixel distribution for each class. This allows us to understand what the model is learning from the data.

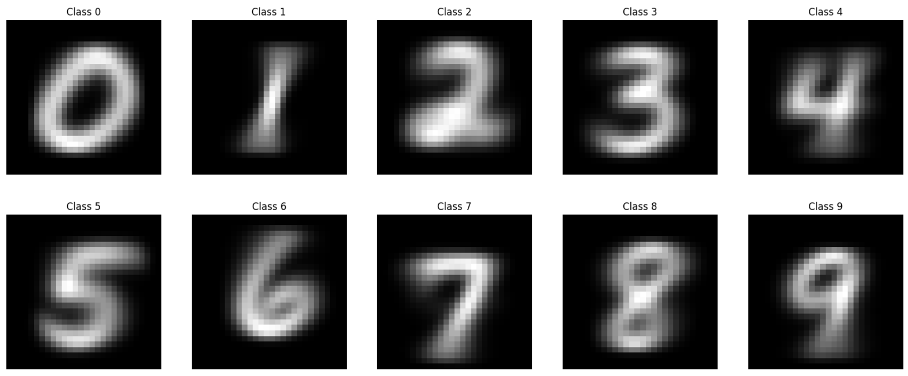


Figure 6.2: Mean of each pixel distribution for each class.

This visual representation is obtained with the following code:

```
def get_mean_images(self):
    means = []
    for i, c in enumerate(self.classes_):
        mean = self.alpha_[i] / (self.alpha_[i] + self.beta_[i])
        means.append(mean.reshape(28, 28))

    return means
```

6.3.1 Execution time

To train the classifier on the entire training set it takes about 0.69 seconds. To test the classifier on the entire test set it takes about 77.56 seconds. Also in this case, the training time is much lower than the test time. This is because the training phase is less computationally expensive than the prediction phase. After all, the algorithm must only compute prior probabilities and α and β for each class in the training data. Otherwise, during the prediction phase, the algorithm must compute the probability of each class for each sample.

Chapter 7

Conclusions

7.1 Models comparison

Table 7.1 summarizes the results obtained with the different models on the MNIST dataset.

Model Name	**Accuracy (%)**	**Train Time (s)**	**Test Time (s)**
Random forest	96.8	242.31	1.44
linear kernel SVM	94.2	111.37	26.47
2-poly kernel SVM	98	94.48	19.17
RBF kernel SVM	98.4	161.53	45.62
K-NN	97.4	0.19	17.55
Beta Naive Bayes	84.3	0.69	77.56

Table 7.1: Model Performance Metrics

Overall, the best model is the RBF kernel SVM, with an accuracy of 98.4%. However, it is also one of the slowest models, with a training time of 161.53 seconds and a test time of 45.62 seconds. The second best model is the 2-poly kernel SVM, with an accuracy of 98%, a training time of 94.48 seconds and a test time of 19.17 seconds. This model is faster than the RBF kernel SVM, but it is still quite slow.

K-NN overall performed very well, with an accuracy of 97.4%, a training time of 0.19 seconds and a test time of 17.55 seconds.

Linear kernel SVM and Naive Bayes did not perform as well as the other models, with an accuracy of 94.2% and 84.3% respectively.

Finally, the Random Forest model performed quite well, with an accuracy of 96.8%, a training time of 242.31 seconds and a test time of 1.44 seconds. This is the model with the fastest test time, but it is also the slowest to train.