Università
Ca'Foscari
Venezia

Master's Degree Programme in Computer Science and Information Technology
Software Development and Engineering

2nd assignment

Artificial Intelligence: Discriminative and Generative Classifier

Student
Andrea Munarin
Matriculation Number 879607

Academic Year
2022-2023

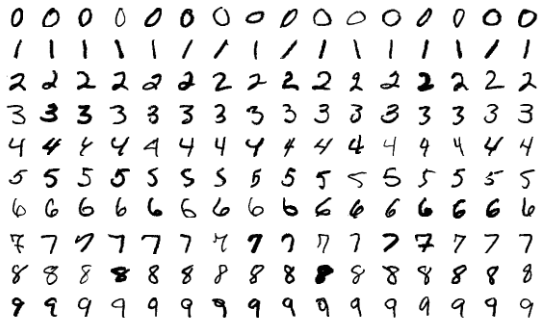
Index

Chapter 1

Introduction

The purpose of this assignment is to learn how to train a model using different type of classifiers. The problem we want to model is very famous and can be described as follows: "write a handwritten digit classifier for the MNIST database[1]. These are composed of 70000 28×28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set."

Figure 1.1: Example of data presents in the MNIST dataset



Then the pictures can be represented as 784-dimensional (28×28) feature vectors (see section 2 for more details) of real values between 0 (black) and 1 (white).

[1]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. The database is also widely used for training and testing in the field of machine learning.

Once we obtain all the data, we can train and test our model with these different techniques:

- ***SVM*** using linear, polynomial of degree 2, and RBF kernels;
- ***Random forests***;
- ***Naive Bayes*** classifier;
- ***k-NN***;

SVM and random forests will be implemented using the scikit-learn library. The Naive Bayes and k-NN classifier will be created from zero and explained in the details in the next sections.

One last thing to say is that we will use 10 way cross validation to optimize the hyperparameters for each classifier.

Chapter 2

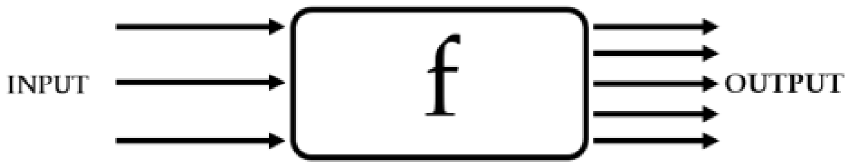
Theory concepts

In this chapter, we will look at the main concepts behind "Learning" and some of the most famous technique, understanding how they work.

2.1 Learning

The main goal of the algorithms is not to memorize but to generalize or predict the output given some input, after a training process, performed using some examples. This can be seen as: we want to find a function which will be a good predictor of our output for a future input. In order to compute this function f we use our set of examples.

Figure 2.1: Main concept behind learning



From now on, we will call the **Training Set**, the set of examples used to compute the mapping relation. We can have mainly three type of learning based on the structure of our problem:

- ***Supervised Learning***: all our data has an associate label that refers to the expected output.

- ***Unsupervised Learning***: all our data are only collection of information, and we try to group them into "natural cluster".
- ***Reinforcement Learning***: an agent interacting with the environment, makes observations, takes actions, and is rewarded or punished; The agent tries to maximize the rewards.

All the techniques we are going to look in this paper are based on "***Supervised Learning***" (MNIST dataset provide all the information about the expected output of each picture). Now, we can formalize our Learning Problem as:

$$L : (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{H}$$

where:

- $\mathcal{X}$ is the Set of possible inputs
- $\mathcal{Y}$ is the Set of possible outputs
- $X : \Omega \to \mathcal{X}andY : \Omega \to \mathcal{Y}$ represents two Random Variables
- $\mu(X, Y)$ is the UNKNOWN join distribution
- $\mathcal{H}$ is a space of functions $f : X \to Y$

Then we need to choose the best function f_s from $\mathcal{H}$ that predict better our model. In order to do that, we select the function that minimize the error of our trained map. To proceed, using the **loss function** V (function that express the penalty for a wrong prediction, this can be possible because our training set is labeled) we can define the **true** and the **empirical error**.
True error is what we really want to minimize, but can be calculated only if we know the exact Join distribution (but this cannot happen). So we compute the ***empirical error*** as a measure of the generalization of our model:

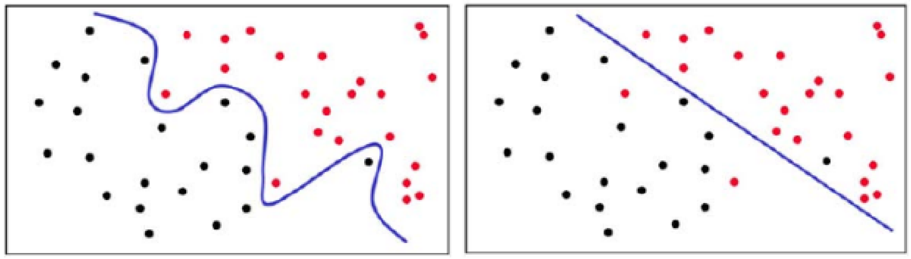
$$l_s[f] = \frac{1}{l} \sum_{i=1}^{l} V(f, z_i)$$

Remember that the training error for the solution must converge to the true error and thus be a proxy for it, otherwise the solution would not be predictive.

In order to select the correct function (classifier), we need also to consider the ***Bias-Variance tradeoff***. Bias-Variance problem reasons on the fact that we want to choose a good function that works well on our data, but it's not too complicated, we can have a situation in which:

- The function is too simple. This may result in many errors (high bias and underfitting)
- The function is too complicated. We probably are considering a model with very few errors. This means that our model learned also the "noise" present in the training set (not good generalization, ***overfitting*** the data) and also the function may be very complex and difficult to adapt to new information (high variance)

Figure 2.2: Examples of bias-variance tradeoff



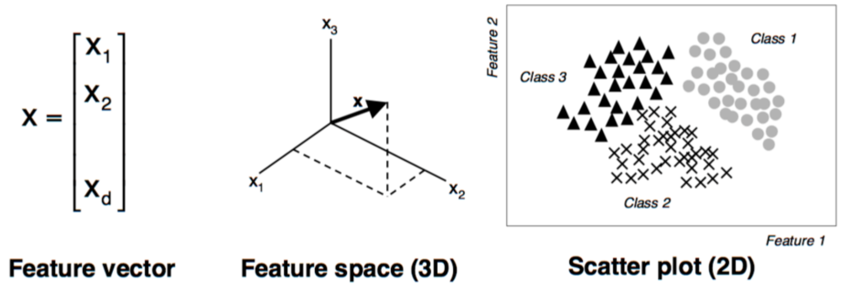
To compute and select the best function, we will use the ***Leave-One-Out K-Fold Cross-Validation***. This technique permits to estimate the (expected) error of the algorithm. This work as follows:

1. We randomly divide the training set of into K folds (typically $K = 5$ or 10)
2. The first fold is treated as a **validation set** (used to test our parameters), and the method is tested on the remaining $K - 1$ folds.
3. The process is repeated K times, taking out a different part each time.
4. By averaging the K estimates of the test error, we get an estimated validation (test) error rate for new observations.

One important last thing to say is that the original data set should be split in the training set and in the test set (often, test set = 30% of the data). Then the training set will be used in cross-validation and the final best classifier can be tested out using the test set. It's really significant that the **data in the test set are never seen by the models** for a good measure of how much our it is accurate.

From now on, we will use the term ***feature vector***. A **vector** is a series of numbers, like a matrix with one column but multiple rows, that can typically be represented spatially. A **feature** is a numerical or symbolic property of an aspect of an object. A feature vector is a vector containing multiple elements about an object. Putting feature vectors for objects together can make up a ***feature space***.

Figure 2.3: Feature vector



2.2 Discriminative Classifier

Discriminative learning algorithms are algorithms that try to learn $p(y|x)$ directly. So, knowing the feature vector, they compute the probability for the element to appertain to a class $y \in \mathcal{Y}$ and choose the class with the highest value:

$$y = \underset{y}{\text{argmax}}\ p(y|x)$$

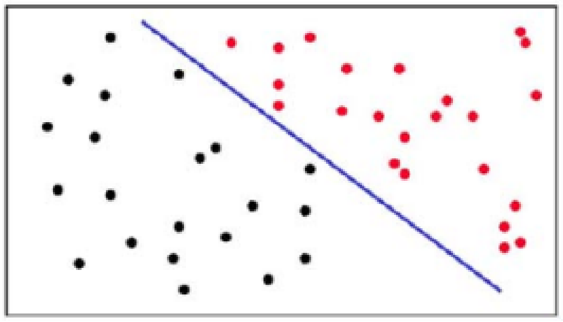
2.2.1 SVM

Linear separators are characterized by a **single linear decision** boundary in the space. As you can see from the figure 2.4 there can be many ways to find a linear classifier. Support Vector Machines are based on the idea to **maximize the geometric margin** between the decision boundary and the points in the training set. This problem is formalized as an optimization constrained problem:

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}||w||^2 \\ \text{subject to} \quad & \forall i \in 1, ..., n. y_i(w \cdot x_i) - 1 \geq 0 \end{aligned} \tag{2.1}$$

In which the constraint specifies the fact that the distance from the margin should be at least 1. Once we find the correct classifier solving the previous problem, we

Figure 2.4: Examples of linear separable data



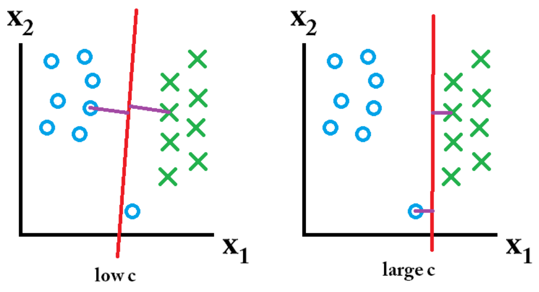
can understand if an object appertain to a class using the **sign function** (in order to understand in which side of the boundary the feature vector is situated). Some observations:

- The optimal classier is defined uniquely (**no local maxima**)
- Max margin **lowers** hypothesis **variance**
- **Learning Polynomial** in number of data and dimensionality
- The classier depends only on the ***support vectors***[1] (feature vectors on the same distance from the margin)
- Classification linear in number of points in support vector
- Learning depends only on **dot product** of sample pairs

The main disadvantages are that we require that the data must be linearly separable. To overcome this issue, we can proceed with two idea. Firstly, we allow some sort of ***misclassification***, changing the constraint in the optimization problem in order to make him dependent to a parameter C. In this way we can soft the margin and obtain better classifier in some situation.

[1]Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

Figure 2.5: C-Parameter Interpretation



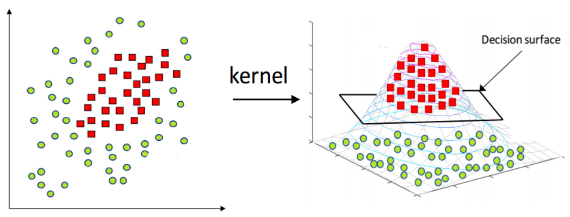
One other thing we could perform is **transform a problem** that was not linearly separable into one that is. In order to achieve this, we can consider a **higher dimensional feature space** and fit our feature vectors into it. Thanks to the fact that SVM only use dot products of the data to achieve such transformation we need only to use the concept of kernels and the "Mercer's Reproducer Theorem" which states:

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:

$$K(x_1, x_2) = \phi(x_1) \cdot \phi(x_2)$$

This means that you can substitute dot products with any positive-definite function K (called kernel)

Figure 2.6: Kernel Interpretation



2.2.2 Random Forest

Before explaining what ***Random Forest*** are, we need to introduce the concept of ***Decision Trees***. A decision tree is a structure in which each internal node contain a condition on an attribute , each branch represents the possible results of the test, and each leaf node represents a class label. The paths from root to leaf represent **classification rules**. Remember that our purpose it to learn from examples, so we need to categorize them and find some attributes to test. To understand which nodes put in which positions (including root and leaf) a common practice is to use a measure call ***Gain*** (based on a notion of ***Entropy***[2]). Given a set of examples S and an attribute A the Gain is obtained as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

One very important question is ***how deep to grow a Tree***: if we choose a very low depth we obtained a bad model (low generalization) and if we choose a very high depth we may obtain a situation of overfitting in which we memorize the dataset instead of generalize the rule. As we will see, depth will be one of the main hyper-parameter[3] in the Random Forest algorithm.

We can improve decision tree generalization by **increasing the number of trees** using

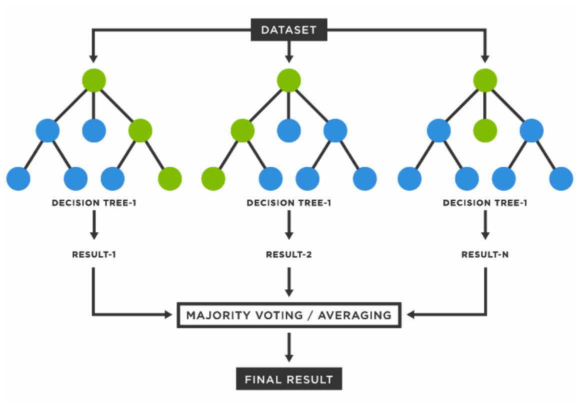
- ***Bagging***
    1. Sample records with replacement (aka "bootstrap" the training data)
    2. Fit an overgrown tree to each resampled data set
    3. Average predictions
- ***Random Forest***
    1. Follow a similar bagging process
    2. each time a split is to be performed, the search for the split variable is limited to a random subset of m of the p variables

Combined, these techniques provides a more diverse set of trees that almost always lowers our prediction error.

[2]In information theory, the entropy of a random variable is the average level of "information", "surprise", or "uncertainty" inherent to the variable's possible outcomes.

[3]In machine learning, a hyperparameter is a parameter whose value is used to control the learning process. By contrast, the values of other parameters (typically node weights) are derived via training.

Figure 2.7: Random Forest example



2.3 Generative Classifier

Discriminative learning algorithms are algorithms that try to model $p(x|y)$ and $p(y)$. So, once we model the two probabilities, we can obtain the $p(y|x)$ by applying the Bayes' theorem:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

We can avoid computing $p(x)$ since our purpose is to evaluate $\underset{y}{\operatorname{argmax}}\, p(y|x)$

2.3.1 Naïve Bayes

Naïve Bayes is a general class of generative models which, as the name states, is strongly based on the **Bayes' theorem**. Remember that what we want to model is $p(x|y)$. In our case of study (Handwritten digits) the dimension of x is 784, and it's hard to compute $p(x_1, \ldots, x_{784}|y) = p(x_1|y)p(x_2|y, x_1)p(x_3|y, x_1, x_2)\ldots$.
Here comes, to help, the fundamental assumption of Naïve Bayes: all the x_i's are ***conditionally independent*** given y. It's really important to understand the meaning of this: if I know that a digit is 2 the fact that a particular pixel is equal to 0.23 doesn't affect the knowledge of other pixels. Note that this is not the same as saying that the 784 pixels are independent to each other. So to classify our problem we use this formula and in the end pick the class with the **highest joint probability**.

$$p(x, y = 0) = p(x|y = 0)p(y = 0) = (\prod_{i=1}^{784} p(x_i|y = 0))p(y = 0)$$

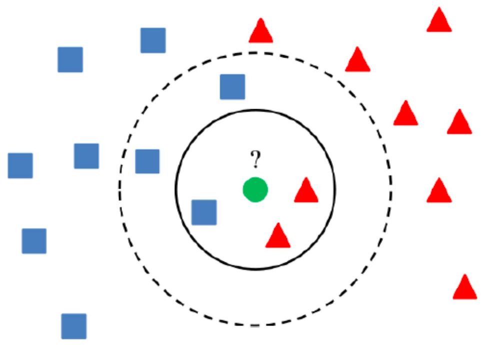
$$\dots$$

$$p(x, y = 9) = p(x|y = 9)p(y = 9) = (\prod_{i=1}^{784} p(x_i|y = 9))p(y = 9)$$

2.3.2 k-NN

k-NN classifier is a generalization of the **nearest neighbor classifier** based on the **nearest neighbor rule**: Look for the nearest training sample and give the new point the same label. It's essential to choose the correct way to measure the difference between two training samples. In our case of handwritten digits, a good approach to measure the difference between two picture could be the total sum of the differences for each pixel.
k-NN instead look for the k nearest neighbors and assign the label according to ***majority vote***[4]. k acts as a smoothing parameter in the decision boundary, and also we can say that as n and k increase (k more slowly than n) k-NN converges to the optimal classier.

Figure 2.8: k-NN example



(a) If we choose $k = 3$ our point will be classified as a red triangle, but if we choose $k = 5$ will be classified as a blue square

[4]Most present label in the k neighbors

Chapter 3

Implementation

In this chapter we will see how the models has been implemented using some useful famous libraries.

3.1 General Notes

In this section it will be described all the methods, function and code snippet that are going to be used largely in all the models giving some insight on how they work and why we used it.

3.1.1 Fetch Data

All the data were fetched only ones and **saved in a npz file** in order to save time during the testing of the code. Pay attention to the fact that the feature vector is normalized by diving each value by 255.0. At the end we obtain 70000 feature vector X of 784 real values between 0 and 1 and 70000 labels.

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

3.1.2 Libraries

In order to compute all the operations, we are going to use some very useful libraries:

- ***NumPy***: offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.
- ***pandas***: fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- ***time***: permits to measure timestamp.
- ***Scikit-learn***: Simple and efficient tools for predictive data analysis. It will be central in this paper, and in the next sections we will look at some powerful methods.
- ***Seaborn*** and ***Matplotlib***: libraries used to plot graph, in our case we are going to use them for the confusion matrix.

3.1.3 GridSearchCV

GridSearchCV is a very powerful method provided by sklearn that permits to perform **cross validation** (discussed in section 2). The function once it completes all the evaluation made available a list of measures about how the cross validation has performed, the best hyperparameter and the best classifier among all. An example of how this can be used in SVM:

```
search = GridSearchCV(SVC(), param_grid, cv=10, n_jobs=16)
search.fit(X_search, y_search)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

df = pd.DataFrame(search.cv_results_)
```

The most impart things to notice are:

- **param_grid**: list of hyperparameters to test
- **cv**: number of folds of our cross validation
- **n_jobs**: number of thread of our CPUs we can use
- **search.best_score_**: return the best accuracy for the best parameters saved in search.best_params_
- **search.cv_results_**: permits to save all the result in a CSV file

3.1.4 Fit and Predict

Fit and *Predict* are the two main methods, that each classifier has, to **train** (fit) and **use** (predict) a model. Once we perform such operation, sklearn provide also the method *accuracy_score* that evaluate the differences between what our model predict and the expected values of our test set:

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

Once we understand all the main methods available on the libraries, implementing the classifiers becomes very easy. The main important part for each classifier is the choice of the **hyperparameter** we want to test. When the GridSearchCV end its computation, we can measure time for training and for testing using the best parameters. This will be our main approach to all the algorithms we are going to implement.

3.2 SVM

In the SVM algorithm, as we discuss in the previous chapter, we will look how the classifier works with different type of kernels and with variation of **parameters C**, in particular:

```
param_grid = {
    'C': [1, 3, 7, 10],
    'kernel': ('linear','poly','rbf')
}
search = GridSearchCV(SVC(degree=2), param_grid, cv=10, n_jobs=16)
search.fit(X_search, y_search)
```

Then we will save the **best C for each kernel** and test the time and accuracy with the three classifier:

```
for kernel in ['linear', 'poly', 'rbf']:
    if kernel == 'linear':
        model = LinearSVC(C=C[kernel])
    else:
        model = SVC(C=C[kernel], kernel=kernel, degree=2)
    start_fit = time.time()
    model.fit(X_train, y_train)
    end_fit = time.time()

    start = time.time()
    y_pred = model.predict(X_test)
    end = time.time()
```

Note: we use ***LinearSVC*** instead of SVC with linear kernel only for improving the performance.

3.3 Random Forest

In the Random Forest the number of hyperparameters which are important to consider increase, in this paper we analyze four of them:

- ***n_estimators***: numbers of tree in the "forest"
- ***max_depth***: maximum depth of the trees in the forest (too much depth may result in overfitting)
- ***min_samples_leaf***: minimum number of samples required to be at a leaf node.
- ***min_samples_split***: minimum number of samples required to split an internal node.

```
param_grid = {
    'n_estimators': [10, 100, 1000],
    'max_depth': [2, 10, 40, 100],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 5, 10],
}

search = GridSearchCV(RandomForestClassifier(), param_grid,
                      cv=10, n_jobs=16)
search.fit(X_search, y_search)
```

3.4 Naive Bayes

In order to use the algorithms we are going to create with the same API of sklearn, we need to create a class with fit and predict method[1]. **Fit method** in the Naive Bayes algorithm is really simple: it computes the **Mean** and **Variance** for each pixel and for each class in order to find the ***alpha*** and the ***beta***: parameters of the ***Beta-distribution***. We assume, in fact, that the pixel of the picture follows this type of distribution, so we can model

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

Other important things to notice:

- how we compute α and β: we used the so-called ***method of moments***[2]

$$\alpha = K \cdot E[X]$$

$$\beta = K \cdot (1 - E[X])$$

$$K = \frac{E[X](1-E[X])}{var(X)} - 1$$

- Add a small ϵ to numerator and denominator of k. This is necessary because some pixels are all equal to zero for some particular class, this means $variance = 0$ and $mean = 0$. **In order to model a distribution** to this pixel (in which is very probable to be 0) we add 0.0001 in the computation of K.

- compute the $p(y)$ as number of feature vector of class y over number of all feature vectors.

```
def fit(self, X, Y)
    self.X = X
    self.y = y
```

[1]also other methods are important but not discussed in this paper

[2]In statistics, the method of moments is a method of estimation of population parameters. It starts by expressing the population moments (i.e., the expected values of powers of the random variable under consideration) as functions of the parameters of interest. Those expressions are then set equal to the sample moments. The number of such equations is the same as the number of parameters to be estimated. Those equations are then solved for the parameters of interest. The solutions are estimates of those parameters.

```
    for i in range(10):
        X_i = self.X[self.y == i]
        mu_i = np.mean(X_i, axis=0)
        sigma_i = np.var(X_i, axis=0)

        for j in range(784):
            k = (mu_i[j] * (1 - mu_i[j]) - sigma_i[j] + 0.001 )
                                        /( sigma_i[j] + 0.001)
            self.alphas[i][j] = 1 + mu_i[j] * k
            self.betas[i][j] = 1 + (1 - mu_i[j]) * k
        self.priors[i] = X_i.shape[0] / self.X.shape[0]
```

What we need to notice in the **predict method** is listed as:

- compute using the **beta distribution** the probability of each pixel to appertain to a certain class
- multiply, using the **Naive Bayes assumption**, each probability in order to calculate $p(x|y)$ where x is the entire feature vector
- multiply the precedent result with $p(y)$ and after have done this for all the 10 digits, select the label which corresponds to the **highest probability**.

```
def predict(self, X):
    len = X.shape[0]
    y_pred = np.zeros(len)
    for i in range(len):
        probs = np.zeros(10)
        for j in range(10):
            prod = 1
            for k in range(784):
                if(self.alphas[j][k]>0 and self.betas[j][k]>0):
                    p = math.gamma(self.alphas[j][k]+self.betas[j][k])
                    p1 = math.gamma(self.alphas[j][k])
                    p2 = math.gamma(self.betas[j][k])
                    cost = p/(p1*p2)
                    x = X[i][k]
                    if X[i][k]==0:
                        x=0.0001
                    elif X[i][k]==1:
```

```
                    x=0.9999
                ex = (x**(self.alphas[j][k]-1)) *
                     ((1-x)**(self.betas[j][k]-1))
                value = cost*ex
                prod *= value

            probs[j] = self.priors[j] * prod
        y_pred[i] = np.argmax(probs)
    return y_pred
```

3.5 k-NN

In order to implement k-NN and use it as a sklearn function (in this way we can pass it to the **GridSearchCV**) we need to define a ***class kNN*** which implement all the necessary methods. As we saw in Naive Bayes, we will look to fit and predict.

The fit method is very simple and only need to memorize the training set inside the class:

```
def fit(self, X, y):
    self.X_train = X
    self.y_train = y
```

The predict method will **loop on the number of picture** to predict and will assign the labels basing on the **majority vote on the k nearest neighbors**. The distances between the current feature vector and the ones present in the model are calculated as the ***Euclidean distance*** for each pixel. Using in the end ***argsort*** we can retrieve the indexes of the k nearest feature vectors and find the label.

```
def predict(self, X):
    len = X.shape[0]
    y_pred = np.zeros(len)
    for i in range(len):
        distances = np.sqrt(np.sum((X[i] - self.X_train)**2, axis=1))
        indexes = np.argsort(distances)
        best_k_idx = indexes[:self.k]
        k_nearest_labels = [self.y_train[i] for i in best_k_idx]
        y_pred[i] = max(k_nearest_labels, key=k_nearest_labels.count)
    return y_pred
```

Finally, we can use the GridSearchCV to perform 10-fold cross validation:

```
param_grid = {
    'k': [1, 3, 5, 7, 9]
}
search = GridSearchCV(kNN(), param_grid, cv=10, n_jobs=16)
start_search = time.time()
search.fit(X_search, y_search)
```

Chapter 4

Evaluation

Before starting to analyze all the performances of all the algorithms remember that the set of data (70000 rows) is split in:

- 10000: **search set** used in cross validation (different from the test set in order to have less computational effort).
- 60000: *training set.*
- Last 10000: **test set**.

4.1 Parameters' tuning

Now we will look at how the various algorithm works when we tune their parameter explaining their behavior.

4.1.1 SVM

In the support vector machine, we can notice from the table that:

- Varying the **kernel** improve the **mean score** (rbf perform better than poly and linear). This happens because, as we saw, the **kernels** permits to **recognize** also non-linearly separable data and **the shape they can generalize** depends on the kernel itself. So linear is more limited than poly, and poly more limited than rbf
- Parameter C **too small**, as we discuss, may lead to less generalized model. But with some kernel also **big values** for C may lead to too much data to be incorrectly classified on some side of the boundary.

- Increasing the complexity of kernel, the time for training the model increased as well
- **Rbf kernel** with $C = 3$ is our best solution.

mean_fit_time	**C**	**kernel**	**mean_score**	**rank**
45.0986	1	linear	0.9178	9
68.6491	1	poly	0.9555	8
79.5505	1	rbf	0.9604	6
40.7619	3	linear	0.9135	10
59.0563	3	poly	0.9606	5
75.0713	3	rbf	0.9671	1
41.3415	7	linear	0.9134	11
53.0924	7	poly	0.9607	4
74.2467	7	rbf	0.9669	3
41.4170	10	linear	0.9134	11
53.9003	10	poly	0.9604	6
62.7643	10	rbf	0.9671	1

4.1.2 Random Forest

In the Random Forest, we need to tune several parameters:

- For all the parameter we have the same concept: low value $\rightarrow$ low generalization, high value $\rightarrow$ overfitting $\rightarrow$ low generalization
- **D**: ***depth***. 100 get the best results, but also 40 with other parameters perform very well.
- **SL**: ***min sample leaf***. 1 is more than sufficient
- **SS**: ***min samples split***. also here we notice that with high value we aggregate too many samples in one branch, loosing generalization
- **E**: ***number of estimators***. The number of trees is a fundamental hyperparameter, and we can notice that when this increments we achieve great results. Notice that the increase from 10 to 100 is different from the one from 100 to 1000. This happens because after some trees, the **accuracy will stabilize and oscillate around that value**. Notice also that the time for training the model increased more or less linearly with the number of estimators.

mean_fit_time	**D**	**SL**	**SS**	**E**	**mean_score**	**rank**
0.2078	2	1	2	10	0.5611	106
1.6722	2	1	2	100	0.6493	92
16.763	2	1	2	1000	0.6595	85
0.1899	2	1	5	10	0.5529	108
...	...	...	...	...	...	...
15.7596	2	4	10	1000	0.6598	84
0.6838	10	1	2	10	0.8995	80
5.9936	10	1	2	100	0.9375	39
...	...	...	...	...	...	...
5.3422	10	4	10	100	0.9324	52
52.9744	10	4	10	1000	0.9333	50
...	...	...	...	...	...	...
0.7146	40	1	2	10	0.9062	72
6.8489	40	1	2	100	0.9466	10
67.8682	40	1	2	1000	0.9502	2
0.6901	40	1	5	10	0.9118	59
...	...	...	...	...	...	...
6.3381	40	2	5	100	0.9433	19
63.6879	40	2	5	1000	0.9497	3
0.6436	40	2	10	10	0.9099	64
...	...	...	...	...	...	...
0.7164	100	1	2	10	0.9078	71
6.7691	100	1	2	100	0.9466	10
67.1286	100	1	2	1000	0.9502	1
0.6881	100	1	5	10	0.9108	61
6.5383	100	1	5	100	0.9465	12
...	...	...	...	...	...	...
58.9385	100	4	5	1000	0.9426	23
0.632	100	4	10	10	0.9079	70
5.9156	100	4	10	100	0.9372	41
49.4397	100	4	10	1000	0.9416	28

4.1.3 k-NN

In k-NN we only got the k parameter. From the table we can understand that only considering the nearest neighbor ($k = 1$) is too **restrictive** and instead considering many neighbors will result in taking labels from **too distant feature vector**. The best choice for k is 3. Notice that the time displayed here refers to the test (we will see why in the next sections).

mean_score_time	**k**	**mean_score**	**score**
159.3213	1	0.9429	3
152.7122	3	0.944	1
164.5849	5	0.9431	2
152.5493	7	0.942	4
127.8768	9	0.9391	5

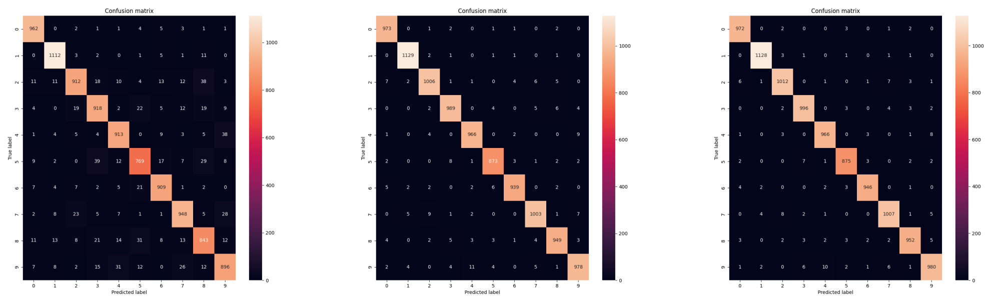
4.2 Accuracy

For all the models, we provide the ***confusion matrix*** in order to analyze how it performs in terms of accuracy. A confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm. Each row of the matrix represents the instances in an actual class, while each column represents the instances in a predicted class.

4.2.1 SVM

When we notice the main concentration of value in the main diagonal means that the algorithm works very well, as we can see all the SVM had great results. The third confusion matrix in Figure 4.1 has lighter colors than the first one, so we can understand that **rbf has a better accuracy than the linear kernel**.

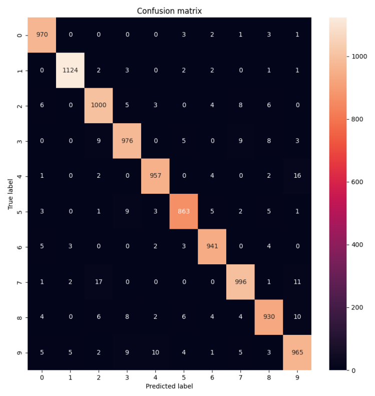
Figure 4.1: SVM confusion matrix



(a) On the left: linear. In the center: poly of degree 2. On the right: rbf

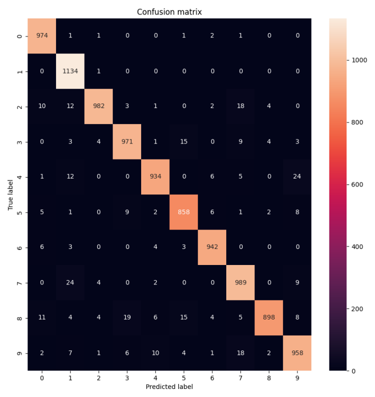
4.2.2 Random Forest

Figure 4.2: Random Forest confusion matrix



4.2.3 K-NN

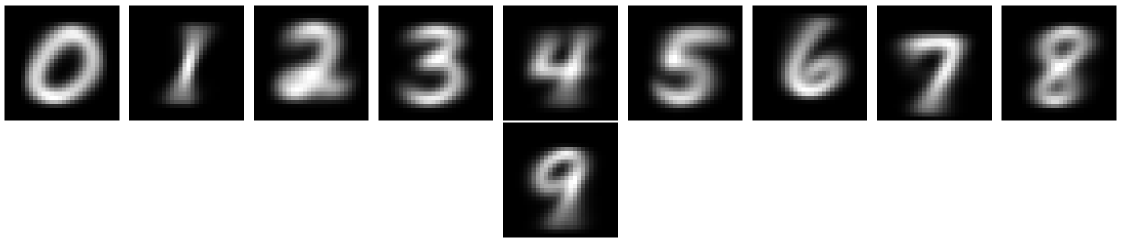
Figure 4.3: k-NN confusion matrix



4.2.4 Naive Bayes

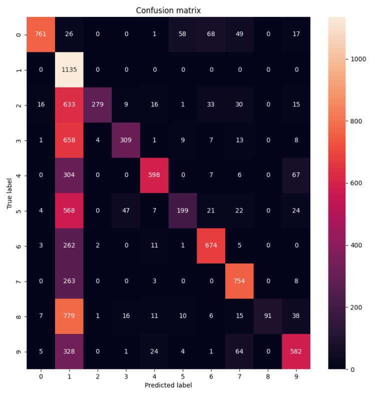
In naive Bayes if, after the training process, we reshape the feature vector using for each pixel the mean of its distribution we can see **what the model is learning**

Figure 4.4: Naive Bayes Learning



The confusion matrix in the Naive Bayes shown very well why the algorithm has a **low accuracy**: Most of the time the algorithm **misclassify** the digits assigning the label 1. This may be due to the choice of the Beta distribution.

Figure 4.5: Naive Bayes confusion matrix



4.2.5 All

At the end we can see how the models works: **SVM**, in particular with polynomial of degree two and rbf kernel, has very **good results**. **Random forest** perform **very well too**. It's very interesting to see how generative classifiers performs:

- Even if **k-NN** is very simple and straight forward, we achieve an accuracy of 97%, and so we can use it as a **good model for our problem**.
- **Naive Bayes** as we saw in the confusion matrix has some limitations **due to the various assumption we have made during the implementation**. The advantages of this model came from the fact that the training is very simple and require small computational effort.

	Accuracy
Linear SVM	0.9182
Poly SVM	0.9805
Rbf SVM	0.9834
Random Forest	0.9722
k-NN	0.9717
Naive Bayes	0.5382

4.3 Performance

It's important to discuss the models also in terms of **performance and time computation**. The first thing we notice is that:

- ***Discriminative classifier*** require more time to train and less time to predict. This is what we expected so far, in fact, once we have trained our model we need to pass the vector to it and let it decide the results (either by choosing in which part of the hyperplane the vector is or searching in the trees)
- ***Generative classifier*** require more time to predict and less time to train, in particular the **training process is almost null** since we only need to save the information in order to then compute the $p(x|y)$ of the test vector.

	Train time	**Predict Time**
Linear SVM	61.3820s	0.0340s
Poly SVM	96.8360s	23.1336s
Rbf SVM	124.0010s	49.7495s
Random Forest	45.3482s	0.5417s
k-NN	0.0000s	846.4977s
NaiveBayes	0.3523s	470.9374s

We can notice how, increasing the complexity of the kernel, we obtain higher times. this because we are **increasing the feature space and this require more calculation** to do in order to understand where the feature vector is situated (note that linear SVM has predicted time almost 0).

Random forest are very powerful: with little time, we reach high accuracy results. Remember that if we choose the number of estimators equal to 100 we would have more or less the same accuracy with a training time of 4/5 seconds!

k-NN and **NaiveBayes** have high time to predict , but but we need to remember that these classes were implemented by hand (with discriminative classifier we used sklearn optimized for multiprocessing). The great advantage of these algorithms is that if we want to predict single item we have the **best performance so far**: the phase of training is practically null, and we need only to test the item by either calculate 784 distances (Train time: 0.0s, Test time: 0.103s) or computing 7840 Beta distribution (Train time: 0.30s, Test time: 0.047s)

Chapter 5

Conclusion

To conclude, we resume **advantages** and **disadvantages** of both discriminative and generative classifier: **discriminative classifier seems to be more accurate** because of how they predict the model using $p(y|x)$ and results in algorithms with **high training time and low test time**. **Generative classifier** on the other end need **no training** since they only need to find $p(x|y)$ and $p(y)$ and the main computation effort is shifted to the predict method. Generative classifier, even if are much simpler to implement and to deal with, can reach very great accuracy and so, generalize very well the problem, as we saw with k-NN.

It does not exist the best algorithm in general, but we always need to think about the problem and find our best solution, remembering also to test out what the parameters could be using cross-validation.