



Ca' Foscari  
University  
of Venice

Foundations of Artificial Intelligence

## Assignment 2: Discriminative and Generative Classifiers

**Professor**

Andrea Torsello

**Student**

Run Jie Xia

Matriculation Number 879779

**Academic Year**

2022-23



# Contents

<b>0</b>	<b>Introduction</b>	<b>3</b>
0.1	Assignment . . . . .	3
<b>1</b>	<b>Theoretical Background</b>	<b>5</b>
1.1	Discriminative and Generative Models . . . . .	5
1.1.1	Discriminative Models . . . . .	5
1.1.2	Generative Models . . . . .	6
<b>2</b>	<b>Project Setup</b>	<b>7</b>
2.1	Training, Validation and Testing . . . . .	7
2.1.1	Dataset Splits . . . . .	7
2.1.2	K-Fold Cross Validation . . . . .	7
2.1.3	GridSearchCV . . . . .	8
2.2	Evaluation Metrics . . . . .	8
2.2.1	Scoring Method . . . . .	8
2.2.2	Confusion Matrix . . . . .	9
2.3	Data . . . . .	9
2.3.1	Limitations . . . . .	10
2.3.2	Specifications of the Machine . . . . .	10
<b>3</b>	<b>Support Vector Machine Classifier</b>	<b>11</b>
3.1	Theory . . . . .	11
3.2	Implementation . . . . .	13
3.3	SVM: Linear Kernel . . . . .	13
3.3.1	Model Training . . . . .	13
3.3.1.1	Hyperparameters Tuning . . . . .	13
3.3.1.2	Best Estimator . . . . .	13
3.3.2	Train . . . . .	13
3.3.2.1	Test Score . . . . .	14
3.3.2.2	Confusion Matrix . . . . .	14
3.4	SVM: Polynomial Kernel of Degree 2 . . . . .	15
3.4.1	Model Training . . . . .	15
3.4.1.1	Hyperparameters Tuning . . . . .	15
3.4.1.2	Best Estimator . . . . .	16
3.4.2	Train . . . . .	16
3.4.2.1	Test Score . . . . .	16
3.4.2.2	Confusion Matrix . . . . .	17

3.5	SVM: Radial Basis Function (RBF) Kernel . . . . .	17
3.5.1	Model Training . . . . .	17
3.5.1.1	Hyperparameters Tuning . . . . .	17
3.5.1.2	Best Estimator . . . . .	18
3.5.2	Train . . . . .	18
3.5.2.1	Test Score . . . . .	18
3.5.2.2	Confusion Matrix . . . . .	19
<b>4</b>	<b>Random Forest Classifier</b>	<b>20</b>
4.1	Random Forest . . . . .	20
4.1.1	Theory . . . . .	20
4.1.2	Implementation . . . . .	21
4.1.3	Model Training . . . . .	21
4.1.3.1	Hyperparameters Tuning . . . . .	21
4.1.3.2	Best Estimator . . . . .	22
4.1.4	Train . . . . .	22
4.1.4.1	Test Score . . . . .	23
4.1.4.2	Confusion Matrix . . . . .	23
<b>5</b>	<b>Naive Bayes classifier</b>	<b>25</b>
5.1	Beta Distribution Naive Bayes . . . . .	25
5.1.1	Theory . . . . .	25
5.1.2	Implementation . . . . .	26
5.1.3	Model Training . . . . .	27
5.1.3.1	Hyperparameters Tuning . . . . .	27
5.1.4	Train . . . . .	27
5.1.4.1	Test Score . . . . .	27
5.1.4.2	Confusion Matrix . . . . .	27
5.1.4.3	Model learning . . . . .	28
<b>6</b>	<b>K-Nearest Neighbors Classifier</b>	<b>29</b>
6.1	K-Nearest Neighbors . . . . .	29
6.1.1	Theory . . . . .	29
6.1.2	Implementation . . . . .	30
6.1.3	Model Training . . . . .	31
6.1.3.1	Hyperparameters Tuning . . . . .	31
6.1.3.2	Best Estimator . . . . .	31
6.1.4	Train . . . . .	32
6.1.4.1	Test Score . . . . .	32
6.1.4.2	Confusion Matrix . . . . .	32
<b>7</b>	<b>Conclusion</b>	<b>34</b>
7.1	Model Comparison . . . . .	34
7.1.1	Model Accuracy . . . . .	34
7.1.2	Model's Prediction and training times: . . . . .	35

# 0 Introduction

This document reports on the theory and software implementation process used to achieve the goal of the assignment described below.

## 0.1 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. In Python you can automatically fetch the dataset from the net and load it using the following code:

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

This will result in 784-dimensional feature vectors (28\*28) of values between 0 (white) and 1 (black). Train the following classifiers on the dataset:

1. SVM using linear, polynomial of degree 2, and RBF kernels;
2. Random forests
3. Naive Bayes classifier where each pixel is distributed according to a Beta distribution of parameters  $\alpha, \beta$ :

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

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 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.

P.S. For a discussion about maximum likelihood for the parameters of a beta distribution you can look here. However, for this assignment the estimators obtained with the moments approach will be fine:

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

Note:  $\alpha/(\alpha + \beta)$  is the mean of the beta distribution. if you compute the mean for each of the 784 models and reshape them into 28x28 images you can have a visual indication of what the model is learning.

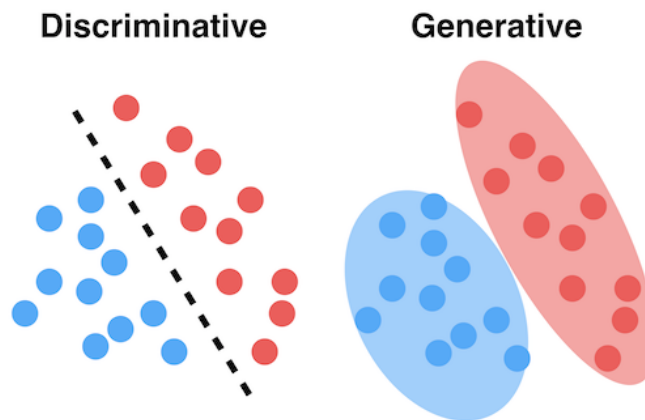
# 1 Theoretical Background

This chapter discusses the theoretical basis of each major concept for the experimental phase of the assignment.

## 1.1 Discriminative and Generative Models

Machine learning models can be divided into two classes of models: discriminative and generative.

Discriminative models draw boundaries in the data space, while generative models attempt to model how the data is distributed in the space. A generative model focuses on explaining how the data were generated and uses the distribution of the training dataset to determine a probability for a given sample, while a discriminative model focuses on making predictions about the labels of the unseen data based on the conditional probability.



### 1.1.1 Discriminative Models

These models try to find the decision boundary between classes in a dataset. This type of algorithm, to predict a point, it checks to which class the point belongs based on the decision boundary. Because of this property, discriminative models may misclassify, but they have the advantage of being more robust to outliers. In other words, discriminative algorithms focus on how to distinguish cases. Therefore, they focus on learning a decision boundary.

Examples of Discriminative Model:

- Logistic regression
- Support Vector Machine (SVM)

- k-Nearest neighbor
- Decision Tree and Random Forest

### 1.1.2 Generative Models

Generative model:

Generative models are models that base their prediction on the probabilistic distribution of classes in a dataset. The main concept of these models is to learn the distribution of data points.

Generative models estimate the probability that each class label is the correct prediction for specific points using joint probability theory. These types of models tend to be better predictors than the discriminative models, but they are more likely to be affected by outliers.

Examples of Generative Model:

- Naive Bayes

While a discriminative model separates classes by modeling the conditional probability and makes no assumptions about the data point, a generative model tends to model the joint probability of data points and make assumptions, creating new instances using probability estimates and maximum likelihood. In other words, generative algorithms learn the basic properties of the data and how to generate them. In contrast, discriminative algorithms focus on how to distinguish instances by learning the decision boundary. Also, generative models have the disadvantage of being less robust to outliers, unlike discriminative models.



## 2 Project Setup

This chapter describes the methods used for tuning, training and experimental evaluation of the models.

### 2.1 Training, Validation and Testing

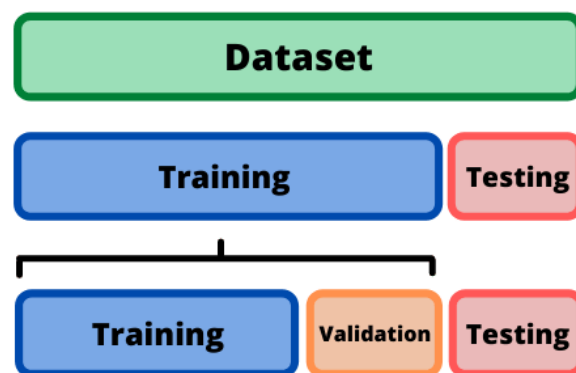
#### 2.1.1 Dataset Splits

To build and evaluate the performance of a machine learning model, we split our dataset into three different datasets. These three datasets are the training data, the validation data and the testing data.

Training data is the partial dataset that we use to train a model. Test data is the sub-dataset that we use to evaluate the performance of a model created with a training dataset.

Although we extract both training and test data from the same dataset, the test dataset should not contain data from the training dataset. The purpose of building a model is to predict unknown outcomes. The test data is used to check the performance, accuracy and precision of the model created with the training data.

Validation data is a subset of data, separate from the training data, that is used to validate the model during the training process. The validation process helps us tune the model's hyperparameters to achieve better results.



#### 2.1.2 K-Fold Cross Validation

Cross-validation is a resampling method used in machine learning to evaluate a model against a limited data sample. The data sample is split into a 'k' number of smaller samples, hence the name: K-fold Cross Validation.

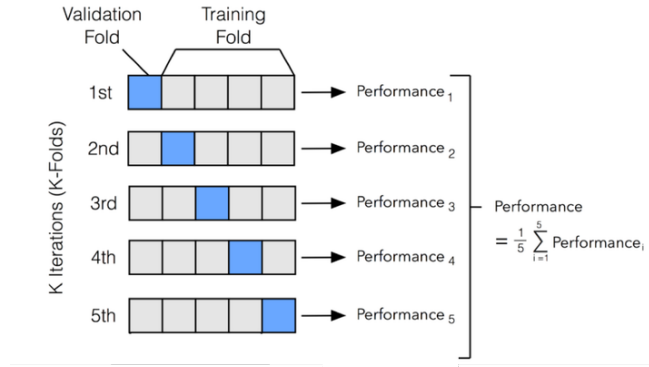


Figure 2.1: Example of 5 fold cross validation.

### 2.1.3 GridSearchCV

An important factor in the performance of machine learning models is their hyperparameters. Once we set appropriate values for these hyperparameters, the performance of a model can improve significantly.

GridSearchCV is the process of hyperparameter tuning to determine the optimal values of hyperparameters for a given model. This technique is then combined with the K-Fold Cross Validation discussed earlier to more reliably choose the configuration of the hyperparameters.

We perform a 10-fold cross validation as described in the assignment.

## 2.2 Evaluation Metrics

### 2.2.1 Scoring Method

**Accuracy Score** We use Accuracy as metric. Accuracy is a metric for evaluating classification models. Informally, accuracy is the proportion of predictions where our model was correct. Formally, accuracy has the following definition:

$$Accuracy = \frac{\text{number of correct predictions}}{\text{total number of predictions}}$$

**Precision** Precision is implied as the measure of the correctly identified positive cases from all the predicted positive cases. Thus, it is useful when the costs of False Positives is high.

**Recall** Recall is the measure of the correctly identified positive cases from all the actual positive cases. It is important when the cost of False Negatives is high.

**Why not F1 Score** F1-score is the harmonic mean of Precision and Recall and gives a better measure of the incorrectly classified cases than the Accuracy Metric.

F1-score is a better metric when there are imbalanced classes as in the above case, while accuracy can be used when the class distribution is similar.

## 2.2.2 Confusion Matrix

A confusion matrix is a technique for summarising the performance of a classification algorithm. Classification accuracy alone can be misleading if we have an unequal number of observations in each class or if you have more than two classes in your dataset. Calculating a confusion matrix can give you a better idea of what your classification model is getting right and what types of errors it is committing.

It is a table with 4 different combinations of predicted and actual values.

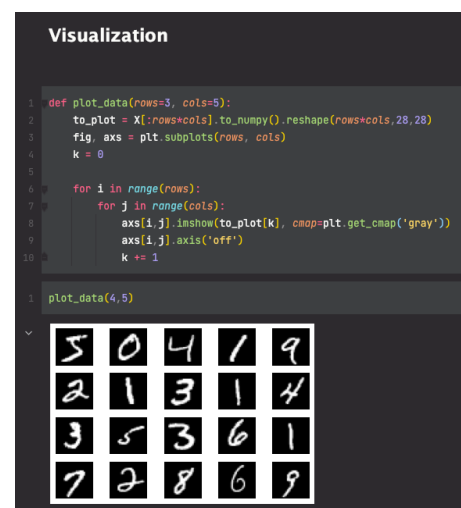
		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Figure 2.2: Example of a binary classification.

- True positive (TP): The prediction is positive and it is true.
- True Negative (TN): The prediction is negative and it is true.
- False positive (FP): The prediction is positive and it is false.
- False Negative (FN): The prediction is negative and it is false.

## 2.3 Data

We will use 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.



### 2.3.1 Limitations

In order to reduce the workload of GridSearchCV for the tuning step, we will only use 7000 training samples, 700 for each class, sampled using the Stratified Sampling technique.



### 2.3.2 Specifications of the Machine

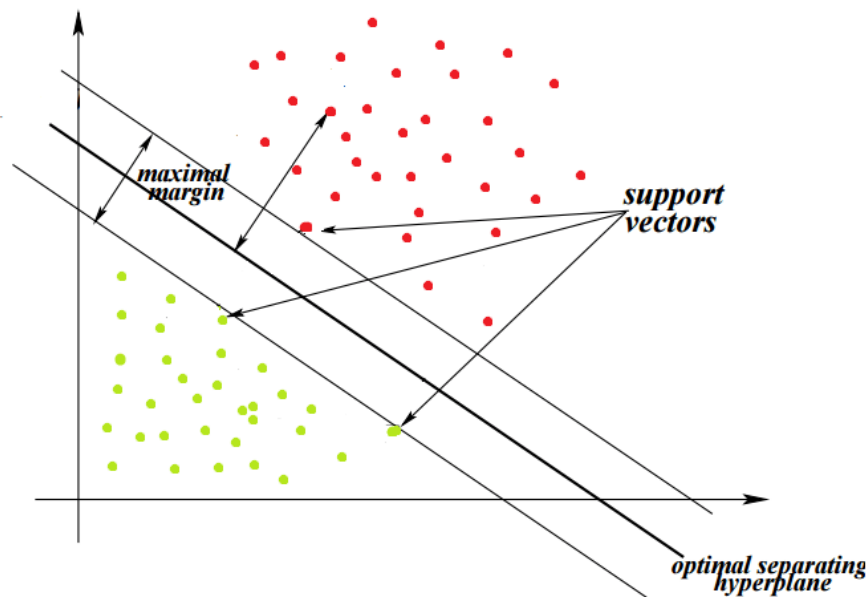
The tests were performed on a MacBook Pro 2017 with the following characteristics:

- Processor: 2,3 GHz Dual-Core Intel Core i5
- RAM: 8 GB
- SSD: 256 GB
- macOS Monterey 12.0.1

# 3 Support Vector Machine Classifier

## 3.1 Theory

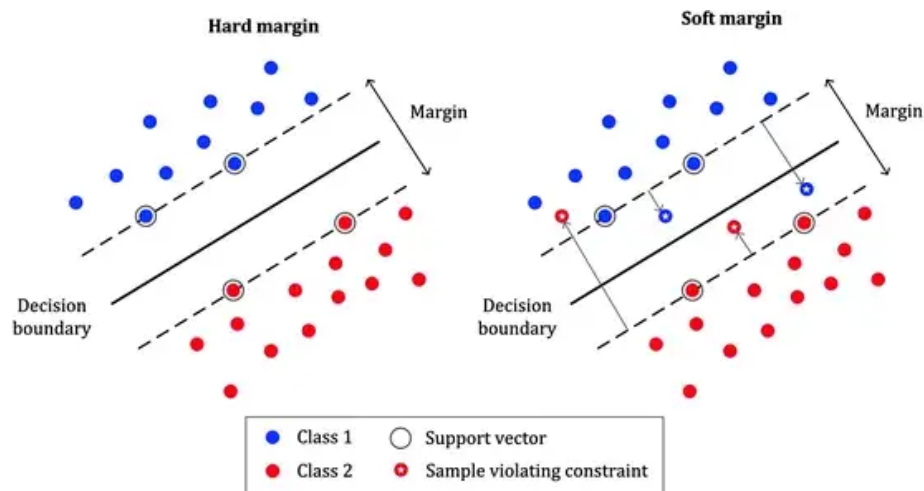
Support Vector Machines (SVMs) are discriminative supervised learning models for classification and regression problems. Its objective is to find the optimal separating hyperplane that distinctly classifies data points belonging to different classes. Hyperplanes are decision boundaries that help in classifying the data points. It is based on the so-called support vectors, the data points that are closest to the hyperplane, and influence the position and orientation of the hyperplane. Based on the position of the support vectors, we maximize the Margin of the classifier, i.e. the sum of the distances that need to be maximized to find the optimal decision boundary. Given a point, the prediction of its class is determined by the distance of this point from the separating hyperplane.



**Margin** By default, the Support Vector Machine implements the SVM with a hard margin. This only works well if our data is linearly separable. The SVM with hard margin does not allow misclassification.

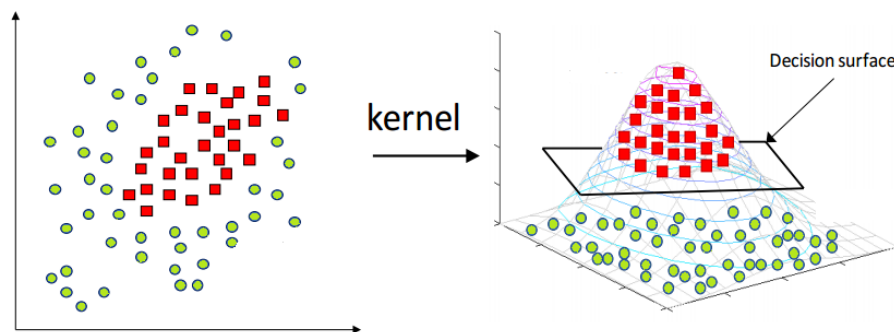
However, if our data is not linearly separable, Hard Margin does not provide a hyperplane because it is not able to separate the data. This is where Soft Margin comes in. Soft Margin SVM allows some misclassification by relaxing the hard constraints of the Support Vector Machine.

Soft Margin SVM is implemented using a hardness coefficient called Regularization parameter ( $C$ ). It allows us to specify how much we want to avoid misclassification of the training samples. If  $C$  is high, we get Hard Margin SVM, the separating hyperplane is calculated by trying to avoid misclassification as much as possible, at the cost of the hyperplane having a smaller margin; conversely, if  $C$  is low, we get Soft Margin SVM and the algorithm searches for a hyperplane with a larger margin, even if this hyperplane misclassifies more points.



**Kernel** Another way to deal with non-linearly separable data is to use the kernel function. A kernel function is always used by the SVM, regardless of whether the data is linear or non-linear. However, its true value only comes into play when the data is not linearly separable. For linear data, we have a linear kernel, but for non-linear data, SVM uses the kernel trick. The idea is to map the non-linearly separable data into a higher dimensional space by using different values of the kernel to find a hyperplane. For example, the mapping function transforms the non-linear 2D input space into a 3D output space using kernel functions. The complexity of finding the mapping function in SVM is significantly reduced by using kernel functions.

There are many types of kernel functions that compute separation line in higher dimensions where it is linearly separable. (Gaussian, Polynomial, etc.).



The required kernels to be tested in this assignment are:

- LinearKernel
- Polynomial Kernel, 2nd degree
- Radial Basis Function (RBF) Kernel

## 3.2 Implementation

As advised in the assignment, a scikit-learn library was used for SVM: sci-kit learn's SVC.

## 3.3 SVM: Linear Kernel

### 3.3.1 Model Training

#### 3.3.1.1 Hyperparameters Tuning

For SVM with Linear Kernel, the kernel is set to linear and the only other hyperparameter that needed tuning was C, the Regularization parameter. GridSearchCV was run using a train set of 7000 samples.

```
SVM using Linear kernel

Note that LinearSVC is another implementation of Support Vector Classification for the case of a linear kernel

1 # automatic parameters tuning
2 svcclsf_lin=SVC(random_state=28)
3 properties_lin={
4     "C": [0.01,0.05,0.1,0.5,1,10,100,1000], # soft to hard margin
5     "kernel": ["linear"]
6 }
7
8 tuned_svcclsf_lin = ms.model_selector(estimator=svcclsf_lin,properties=properties_lin,scoring="accuracy",cv=10,verbose=5,jobs=4,
9                                     x_train=x_data_small,y_train=y_data_small)

Fitting 10 folds for each of 8 candidates, totalling 80 fits
2022-12-28 15:18:05.222 | INFO | src.utilities.model_selection:model_selector:26 - --- 740.8506820201874 seconds ---
```

#### 3.3.1.2 Best Estimator

The best estimator was the one with C=0.05, and the GridSearchCV took 740 seconds to execute.

```
1 print(tuned_svcclsf_lin.best_estimator_)
2 print(tuned_svcclsf_lin.best_score_)

SVC(C=0.05, kernel='linear', random_state=28)
0.9252857142857142
```

### 3.3.2 Train

The model was then trained using the best parameter on the whole train set (60000 samples). This took 352 seconds.

### Model Training

```
1 # best parameters from automatic parameters tuning
2 start_time = time.time()
3 svc_lin_clsf = SVC(**tuned_svc_clsf_lin_best_params_)
4 svc_lin_clsf.fit(data_split.x_train, data_split.y_train)
5 logger.info("Training: - %s seconds -" % (time.time() - start_time))

> [CV 1/10] END C=0.01, kernel=linear;; score=(train=0.941, test=0.910) total time= 22.1s\n[CV 8/10] END C=0.01, kernel=...

2022-12-28 15:23:58.233 | INFO      | __main__:<module>:5 - Training: - 352.0582478046417 seconds -
```

#### 3.3.2.1 Test Score

The testing step took 153 seconds giving an accuracy of 94.07%.

### Performance

```
1 start_time = time.time()
2 svc_lin_test_pred = svc_lin_clsf.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 svc_lin_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=svc_lin_test_pred)

2022-12-28 15:29:42.843 | INFO      | __main__:<module>:3 - Prediction: - 153.135657787323 seconds -

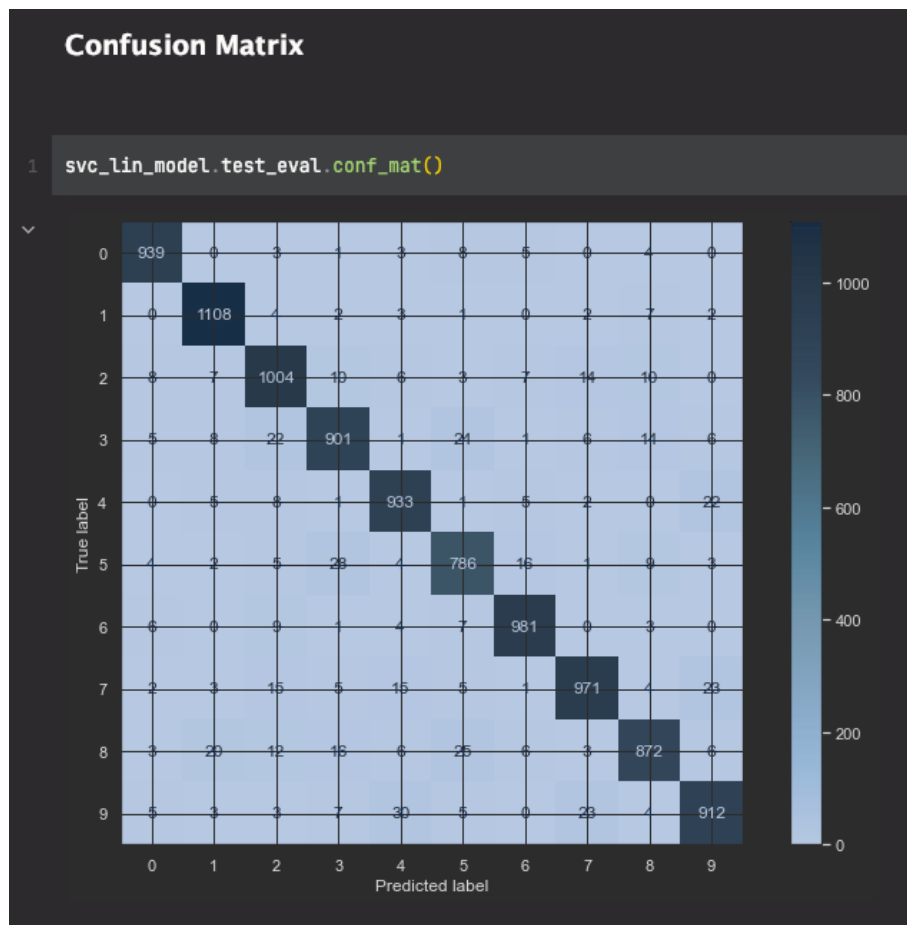
1 print("Testing:")
2 svc_lin_test_eval.acc_eval()

Testing:
----Model Evaluations:-----
Accuracy score: 0.9407
```

#### 3.3.2.2 Confusion Matrix

Below is the confusion matrix for the Linear Kernel SVM. The label 1 is the most correctly classified, and 5 the worst.





## 3.4 SVM: Polynomial Kernel of Degree 2

### 3.4.1 Model Training

#### 3.4.1.1 Hyperparameters Tuning

For SVM with Polynomial Kernel of Degree 2, the kernel is set to "poly", the parameter degree to 2, some values for the gamma parameter, and some values for the regularization parameter C. GridSearchCV was run using a train set of 7000 samples.

### SVM using Polynomial of degree 2 kernel

```
1 # automatic parameters tuning
2 svcclsf_pol=SVC(random_state=28)
3 properties_pol={
4     "C": [0.01,0.05,0.1,0.5,1,10,100,1000], # soft to hard margin
5     "kernel": ["poly"],
6     "degree": [2],
7     "gamma": ["auto",0.1,1]
8 }
9
10 tuned_svcclsf_pol = ms.model_selector(estimator=svcclsf_pol,properties=properties_pol,scoring="accuracy",cv=10,verbose=5,jobs=4,
11                                     x_train=x_data_small,y_train=y_data_small)
```

Fitting 10 folds for each of 24 candidates, totalling 240 fits

2022-12-28 16:22:49.569 | INFO | src.utilities.model\_selection:model\_selector:26 - --- 3139.978579844342 seconds ---

### 3.4.1.2 Best Estimator

The best estimator was the one with  $C=0.1$  and  $\gamma=0.1$ . The GridSearchCV took 3139 seconds to execute.

```
1 print(tuned_svcclsf_pol.best_estimator_)
2 print(tuned_svcclsf_pol.best_score_)

SVC(C=0.1, degree=2, gamma=0.1, kernel='poly', random_state=28)
0.9512857142857143
```

### 3.4.2 Train

The model was then trained using the best parameter on the whole train set (60000 samples). This took 159 seconds.

```
Model Training

1 # best parameters from automatic parameters tuning
2 start_time = time.time()
3 svc_pol_clsif = SVC(**tuned_svcclsf_pol.best_params_)
4 svc_pol_clsif.fit(data_split.x_train, data_split.y_train)
5 logger.info("Training: - %s seconds -" % (time.time() - start_time))

2022-12-28 16:25:30.924 | INFO | __main__:<module>:5 - Training: - 159.4725148677826 seconds -
```

#### 3.4.2.1 Test Score

The testing took 42 seconds giving an accuracy of 97.96%.

```
Performance

1 start_time = time.time()
2 svc_pol_test_pred = svc_pol_clsif.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 svc_pol_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=svc_pol_test_pred)

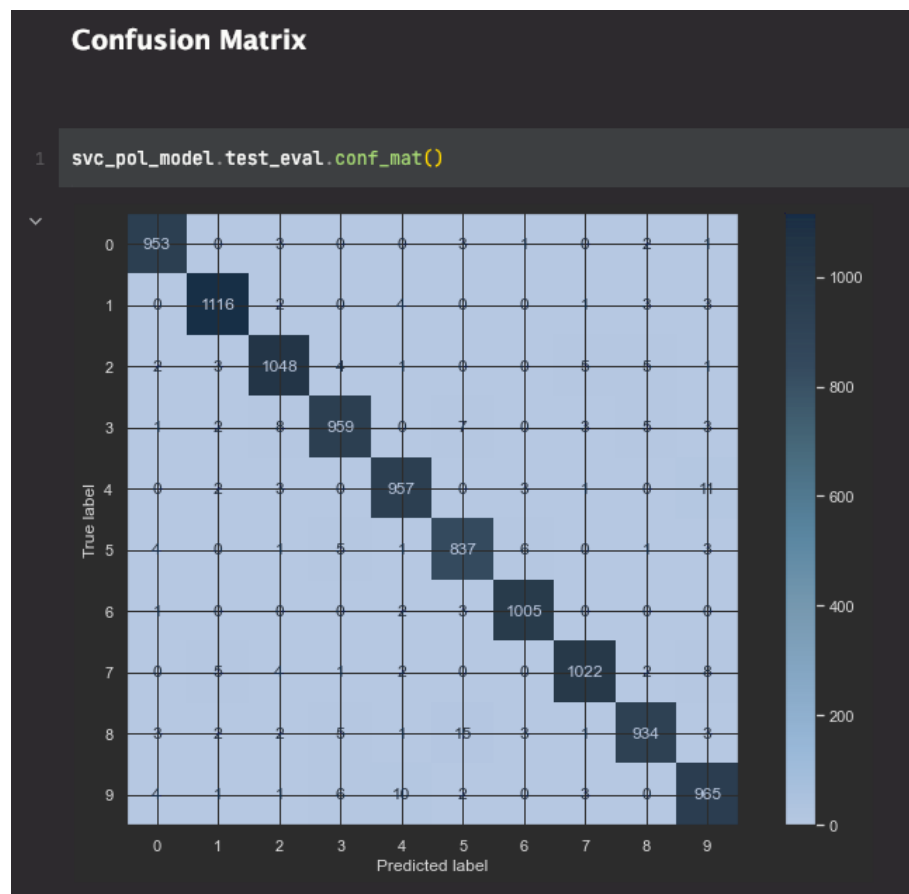
2022-12-28 16:26:13.397 | INFO | __main__:<module>:3 - Prediction: - 42.44668173789978 seconds -

1 print("Testing:")
2 svc_pol_test_eval.acc_eval()

Testing:
-----Model Evaluations:-----
Accuracy score: 0.9796
```

### 3.4.2.2 Confusion Matrix

Below is the confusion matrix for the Polynomial Kernel of degree 2 SVM. The label 1 is easily classified, while 5 is harder to classify.



## 3.5 SVM: Radial Basis Function (RBF) Kernel

### 3.5.1 Model Training

#### 3.5.1.1 Hyperparameters Tuning

For SVM with RBF Kernel, the kernel is set to "rbf", some values for the gamma parameter, and some values for the regularization parameter C. GridSearchCV was run using a train set of 7000 samples.

```
SVM using RBF kernel

1 # automatic parameters tuning
2 svcclsf_rbf=SVC(random_state=28)
3 properties_rbf={
4     "C": [0.01, 0.05, 0.1, 0.5, 1, 10, 100, 1000], # soft to hard margin
5     "kernel": ["rbf"],
6     "gamma": ["auto", 0.1, 1]
7 }
8
9 tuned_svcclsf_rbf = ms.model_selector(estimator=svcclsf_rbf, properties=properties_rbf, scoring="accuracy", cv=10, verbose=5, jobs=4,
10 x_train=x_data_small, y_train=y_data_small)

Fitting 10 folds for each of 24 candidates, totalling 240 fits

2022-12-28 18:18:20.991 | INFO | src.utilities.model_selection:model_selector:26 - --- 6656.828664064407 seconds ---
```

### 3.5.1.2 Best Estimator

The best estimator was the one with  $C=10$  and  $\gamma='auto'$ . The GridSearchCV took 6656 seconds to execute.

```
1 print(tuned_svcclsf_rbf.best_estimator_)
2 print(tuned_svcclsf_rbf.best_score_)

SVC(C=10, gamma='auto', random_state=28)
0.9338571428571427
```

## 3.5.2 Train

The model was then trained using the best parameter on the whole train set (60000 samples). This took 177 seconds.

```
Model Training

1 # best parameters from automatic parameters tuning
2 start_time = time.time()
3 svc_rbf_cls = SVC(**tuned_svcclsf_rbf.best_params_)
4 svc_rbf_cls.fit(data_split.x_train, data_split.y_train)
5 logger.info("Training: - %s seconds -" % (time.time() - start_time))

2022-12-28 18:21:19.417 | INFO | __main__:<module>:5 - Training: - 177.51079320907593 seconds -
```

### 3.5.2.1 Test Score

The testing took 67 seconds giving an accuracy of 95.89%.

## Performance

```
1 start_time = time.time()
2 svc_rbf_test_pred = svc_rbf_clsf.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 svc_rbf_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=svc_rbf_test_pred)

2022-12-28 18:22:27.008 | INFO | __main__:<module>:3 - Prediction: - 67.57905101776123 seconds -

1 print("Testing:")
2 svc_rbf_test_eval.acc_eval()

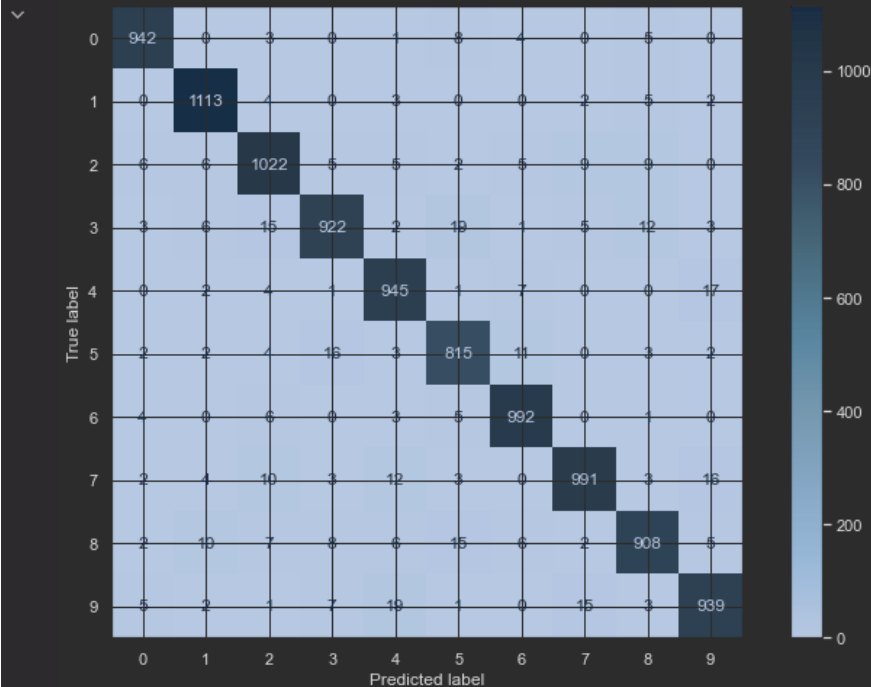
Testing:
-----Model Evaluations:-----
Accuracy score: 0.9589
```

### 3.5.2.2 Confusion Matrix

Below is the confusion matrix for the RBF Kernel SVM. It can be noted that it performs well for the label 1, and the worst is 5.

## Confusion Matrix

```
1 svc_rbf_model.test_eval.conf_mat()
```

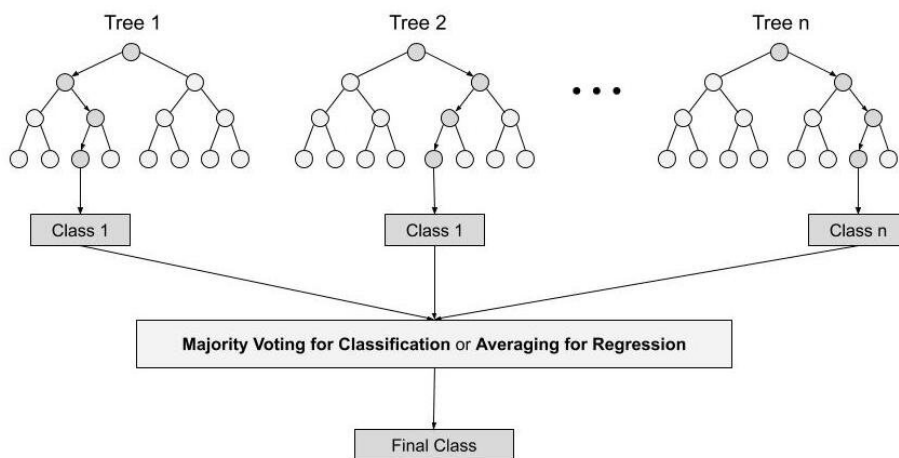


# 4 Random Forest Classifier

## 4.1 Random Forest

### 4.1.1 Theory

The Random Forest is a supervised discriminative machine learning algorithm consisting of the application of the bagging technique to decision trees, often used in classification and regression problems. It builds decision trees on different samples and uses their majority vote for classification and average in the case of regression.



**Decision Trees** Decision trees are the building blocks of the Random Forest model. they are discriminative machine learning models based on the concepts of predicates and splits. In a decision tree, each internal node denotes a test on a feature that defines how the data will be split, each branch represents a result of the test, and each leaf node contains a class label. An important factor for decision trees is that each node focuses on identifying an attribute and a splitting condition for that attribute, minimizing the mixing of class labels and consequently creating relatively pure subsets. To define how good a partitioning is, we have splitting criteria such as Entropy, Information Gain, Gain Ratio, and Gini Index.

**Entropy** Entropy is the degree of uncertainty, impurity or randomness or a measure of the purity of data points. It characterizes the impurity of any class of samples.

**Information Gain** Based on the concept of entropy, information gain is defined as the error of a dataset measured as the entropy that a split achieves over the pre-split dataset.

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

- where  $p_i$  is the probability/frequency of label  $i$
- This is indeed the **entropy**, i.e., a measure of the randomness of the labels
- Maximum:  $\log m$ , where  $m$  is the number of classes, when records are equally distributed among all classes implying least information
- Minimum: 0.0 when all records belong to one class, implying most information (assume  $0 \log 0 = 0$ )

**GINI Index** GINI is a measure of statistical dispersion developed by the Italian statistician and sociologist Corrado Gini (the index was published in 1912). Like Information Gain, it favours "pure" partitions, i.e. datasets containing only one class. The Gini cost measure is defined as follows:

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

- Maximum:  $(1 - 1/m)$  when records are equally distributed among all classes, implying least interesting information
- Minimum: (0.0) when all records belong to one class, implying most interesting information

**Ensemble Methods** Ensemble simply means the combination of several models. Thus, a collection of models is used to make predictions, rather than a single model. There are two types of methods:

**Boosting** It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example: ADA BOOST.

**Bagging** It creates a different training subset from training data with replacement and the final output is based on majority voting. Random Forest works on the Bagging principle. It is also known as Bootstrap Aggregation. Bagging involves selecting a random sample from the dataset. Each model is thus generated from the samples (Bootstrap Samples) of the original data with replacement, which is called row sampling. This step of row sampling with replacement is called bootstrapping. Now each model is trained independently, which produces results. The final output is based on a majority vote after the results of all models are combined. This step, where all results are combined and the output is based on the majority decision, is called aggregation.

## 4.1.2 Implementation

As advised in the assignment, a scikit-learn library was used for random forest: sci-kit learn's RandomForestClassifier.

## 4.1.3 Model Training

### 4.1.3.1 Hyperparameters Tuning

The following hyperparameters were chosen to perform the grid search on:

- n\_estimators
- min\_samples\_leaf

- max\_leaf\_nodes

GridSearchCV was run using a train set of 7000 samples.

```

Tuning

1 # automatic parameters tuning
2 rfclf=RandomForestClassifier(random_state=28)
3 properties={
4     "n_estimators": [x for x in range(50,201,50)],
5     "min_samples_leaf": [x for x in range(50,201,50)],
6     "max_leaf_nodes": [x for x in range(75,176,25)]
7 }
8
9 tuned_rfclf = ms.model_selector(estimator=rfclf,properties=properties,scoring="accuracy",cv=10,verbose=5,jobs=4,
10                                x_train=x_data_small,y_train=y_data_small)

  Fitting 10 folds for each of 80 candidates, totalling 800 fits
[CV 2/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=50; score=(train=0.901, test=0.873) total time= 5.1s
[CV 6/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=50; score=(train=0.901, test=0.899) total time= 3.6s
[CV 9/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=50; score=(train=0.901, test=0.871) total time= 4.2s
[CV 3/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=100; score=(train=0.907, test=0.890) total time= 7.9s
[CV 7/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=100; score=(train=0.906, test=0.883) total time= 10.2s
[CV 2/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=150; score=(train=0.908, test=0.880) total time= 15.6s
[CV 6/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=150; score=(train=0.911, test=0.899) total time= 16.2s
[CV 9/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=150; score=(train=0.910, test=0.891) total time= 15.1s
[CV 3/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=200; score=(train=0.911, test=0.894) total time= 18.9s
[CV 8/10] END max_leaf_nodes=75, min_samples_leaf=50, n_estimators=200; score=(train=0.911, test=0.884) total time= 15.1s
[CV 2/10] END max_leaf_nodes=75, min_samples_leaf=100, n_estimators=50; score=(train=0.867, test=0.854) total time= 3.2s
[CV 4/10] END max_leaf_nodes=75, min_samples_leaf=100, n_estimators=50; score=(train=0.867, test=0.843) total time= 3.3s
[CV 6/10] END max_leaf_nodes=75, min_samples_leaf=100, n_estimators=50; score=(train=0.874, test=0.864) total time= 3.2s

2022-12-28 14:58:29.194 | INFO | src.utilities.model_selection:model_selector:26 - --- 1441.3851199150085 seconds ---

```

### 4.1.3.2 Best Estimator

The grid search took 1441 seconds to execute, and the best hyperparameters are:

- n\_estimators = 200
- min\_samples\_leaf = 50
- max\_leaf\_nodes = 75

```

1 print(tuned_rfclf.best_estimator_)
2 print(tuned_rfclf.best_score_)

  RandomForestClassifier(max_leaf_nodes=75, min_samples_leaf=50, n_estimators=200,
                        random_state=28)
0.8885714285714286

```

### 4.1.4 Train

The model was then trained using the best parameters on the whole train set (60000 samples). This took 90 seconds.



## Model Training

```
1 # best parameters from automatic parameters tuning
2 start_time = time.time()
3 rf_clsf = RandomForestClassifier(**tuned_rfclf.best_params_)
4 rf_clsf.fit(data_split.x_train, data_split.y_train)
5 logger.info("Training: - %s seconds -" % (time.time() - start_time))

2022-12-28 15:00:04.059 | INFO | __main__:<module>:5 - Training: - 90.1204719543457 seconds -
```

### 4.1.4.1 Test Score

The testing took 1.9 seconds giving an accuracy of 90.39%

## Performance

```
1 start_time = time.time()
2 rf_test_pred = rf_clsf.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 rfreg_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=rf_test_pred)

2022-12-28 15:03:03.290 | INFO | __main__:<module>:3 - Prediction: - 1.914968729019165 seconds -

1 print("Testing:")
2 rfreg_test_eval.acc_eval()

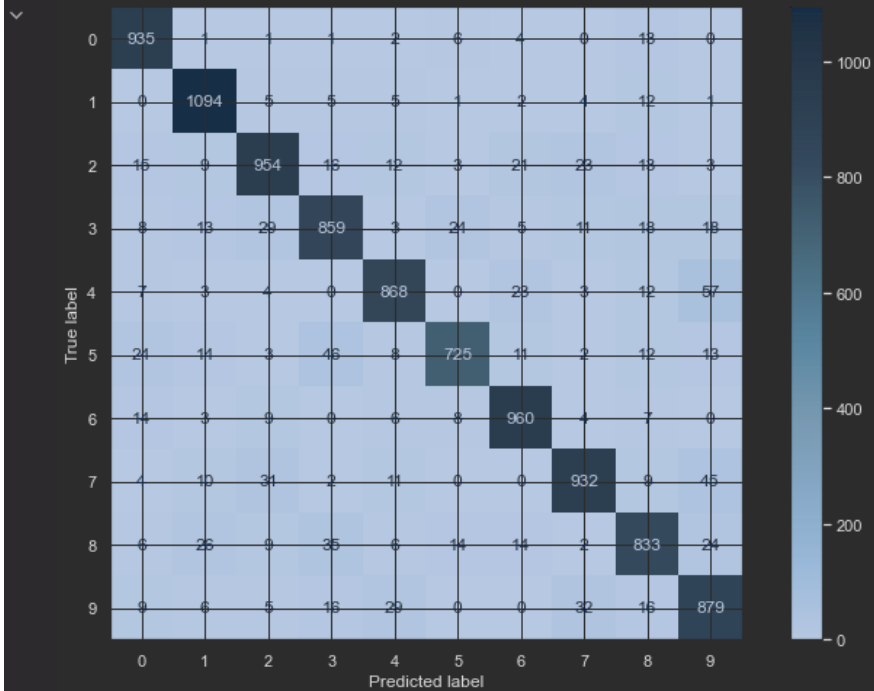
Testing:
-----Model Evaluations:-----
Accuracy score: 0.9039
```

### 4.1.4.2 Confusion Matrix

Below is the confusion matrix for the RandomForestClassifier. It can be noted the labels 1 can be classified easily, and the hardest is 5.

## Confusion Matrix

```
1 rf_model.test_eval.conf_mat()
```



# 5 Naive Bayes classifier

## 5.1 Beta Distribution Naive Bayes

### 5.1.1 Theory

Naive Bayes classifiers are a collection of classification algorithms based on Bayes' theorem. Bayes' theorem determines the probability of an event occurring given the probability of another event that has already occurred.

With Bayes' theorem we can determine the probability of A occurring if B has already occurred. Here B is the evidence and A is the hypothesis. The assumption made here is that the features are independent. That is, the presence of one particular feature does not affect the other. This is why it is called naive.

Bayes' theorem is as follows:

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

Where:

- $P(A)$  = The probability of A occurring
- $P(B)$  = The probability of B occurring
- $P(A | B)$  = The probability of A given B
- $P(B | A)$  = The probability of B given A
- $P(A \cap B)$  = The probability of both A and B occurring

In classification, B is the sample for which we want to make a prediction and A is the class for which we want to calculate the probability.

Naive Bayes classifier predicts the probabilities for each class, i.e. the probability that a particular data set or data point belongs to a particular class. The class with the highest probability is considered the most likely class. This is also called Maximum A Posteriori (MAP).

Types of Naive Bayes Classifiers:

- **Multinomial Naive Bayes:**

This classifier is mostly used for classifying documents i.e. whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of words present in the document.

- **Bernoulli Naive Bayes:**

This is similar to multinomial Naive Bayes but the predictors are Boolean variables. The parameters we use to predict the class variables only take the values yes or no, e.g. whether a word occurs in the text or not.

- **Gaussian Naive Bayes:**

If the predictors take a continuous value and are not discrete, we assume that these values are drawn from a Gaussian distribution.

In the assignment we are asked to use a Beta distribution of parameters  $\alpha, \beta$ :

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

## 5.1.2 Implementation

Here are the implementation of the Naive Bayes Classifier.

```
def fit(self, X_train: pd.DataFrame, y_train: np.ndarray):
    self.X_train = X_train
    self.y_train = y_train

    # get labels
    self.labels = unique_labels(y_train)

    for label in self.labels:
        # get all samples of a specific label
        label_samples = self.X_train[self.y_train == label]

        # compute mean and variance for each feature
        label_mean = label_samples.mean(axis=0)
        label_var = label_samples.var(axis=0)

        # parameter estimation using the moments approach as described in the assignment
        label_k = ((label_mean * (1 - label_mean)) / label_var) - 1
        label_alpha = label_k * label_mean
        label_beta = label_k * (1 - label_mean)

        # handle negative alpha and beta
        label_alpha[label_alpha <= 0] = label_alpha[label_alpha > 0].min()
        label_beta[label_beta <= 0] = label_beta[label_beta > 0].min()

        # label frequency
        frequency = self.y_train[self.y_train == label].size / self.y_train.size

        # mean of the beta distribution, i.e. indication of what the model is learning
        label_betadistr_mean = label_alpha / (label_alpha + label_beta)

        # save params for each unique label
        self.labels_params[label] = {'alpha': label_alpha,
                                     'beta': label_beta,
                                     'frequency': frequency,
                                     'mean_beta_distribution': label_betadistr_mean}

    # Return the classifier
    return self
```

```
def predict(self, X_test: pd.DataFrame) -> pd.Series:
    y_pred = [] # predictions
    idx_pred = [] # indices of the samples predicted

    for row_index, row_feats in X_test.iterrows():
        all_probs = []
        row_feats = row_feats.to_numpy()

        for label in self.labels:
            lbl_params = self.labels_params[label]
            alpha = lbl_params['alpha']
            beta = lbl_params['beta']
            epsilon = 0.05

            # compute probability
            probs = stat.beta.cdf(row_feats * epsilon, alpha, beta) - stat.beta.cdf(row_feats - epsilon, alpha, beta)
            # handle case of variance equal to 0
            np.nan_to_num(probs, copy=False, nan=1.0)
            probability = lbl_params['frequency'] * np.product(probs)

            all_probs.append((probability, label))

        # get the highest probability
        max_prob, max_prob_label = max(all_probs)
        y_pred.append(max_prob_label)
        idx_pred.append(row_index)

    return pd.Series(data=y_pred, index=idx_pred)
```

As described in the assignment, there is also function to plot what the model is learning.

```
def plot_beta_means(self):
    """
    plot what the model is learning using the mean of the beta distribution
    :return:
    """
    to_plot = []
    for label in self.labels:
        to_plot.append(self.labels_params[label]["mean_beta_distribution"].to_numpy().reshape(28, 28))

    fig, axes = plt.subplots(2, 5)
    k = 0
    for i in range(2):
        for j in range(5):
            axes[i, j].imshow(to_plot[k], cmap=plt.get_cmap('gray'))
            axes[i, j].axis('off')
            k += 1
```

## 5.1.3 Model Training

### 5.1.3.1 Hyperparameters Tuning

There is no hyperparameter for this model.

### 5.1.4 Train

The model was trained on the whole train set (60000 samples). This took seconds.

```
Model Training

1 start_time = time.time()
2 b_nb=Beta_NB()
3 b_nb.fit(data_split.x_train, data_split.y_train)
4 logger.info("--- %s seconds ---" % (time.time() - start_time))

2022-12-28 17:49:03.927 | INFO | __main__:<module>:4 - --- 3.434068202972412 seconds ---
```

#### 5.1.4.1 Test Score

The testing took 231 seconds giving an accuracy of 83.08%

```
Performance

1 start_time = time.time()
2 b_nb_test_pred = b_nb.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 b_nb_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=b_nb_test_pred)

2022-12-28 17:52:56.725 | INFO | __main__:<module>:3 - Prediction: - 231.1885838508606 seconds -

1 print("Testing:")
2 b_nb_test_eval.acc_eval()

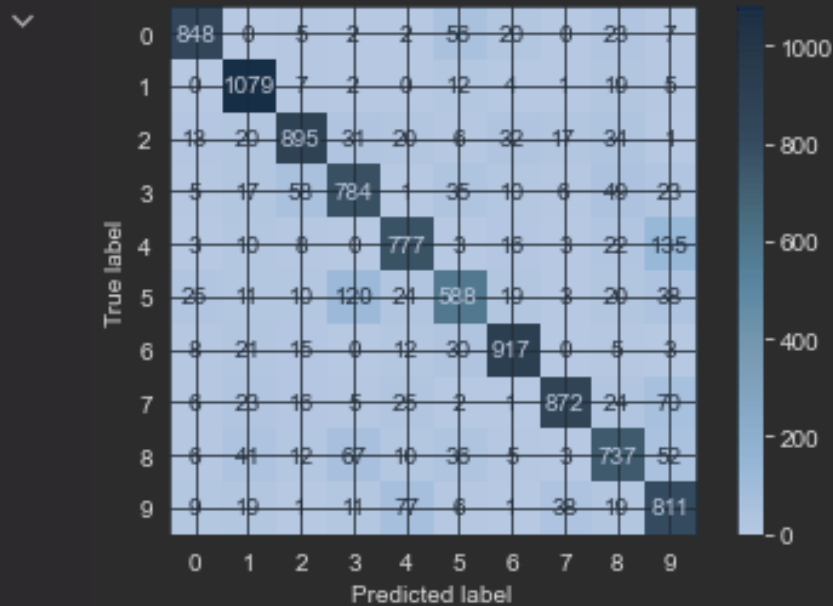
Testing:
----Model Evaluations:----
Accuracy score: 0.8308
```

#### 5.1.4.2 Confusion Matrix

Below is the confusion matrix for the Beta distribution Naive Bayes Classifier. It can be noted the labels 1 can be classified easily, and the worst is 5.

## Confusion Matrix

```
1 b_nb_model.test_eval.conf_mat()
```



### 5.1.4.3 Model learning

Using the mean of the beta distribution we can get a visual indication of what the model is learning.

Visual indication of what the model is learning using the mean of the beta distribution

```
1 b_nb.plot_beta_means()
```



# 6 K-Nearest Neighbors Classifier

## 6.1 K-Nearest Neighbors

### 6.1.1 Theory

K-Nearest Neighbors (KNN) is a supervised machine learning algorithm used for both regression and classification to predict the label of data points by considering the majority of nearest neighbours.

Given a certain number of neighbours  $k$  within a certain distance, the algorithm estimates how likely it is that a data point belongs to one class or another by looking at which class the majority of the  $k$  nearest instances belong to and assigning the class to the new data points.

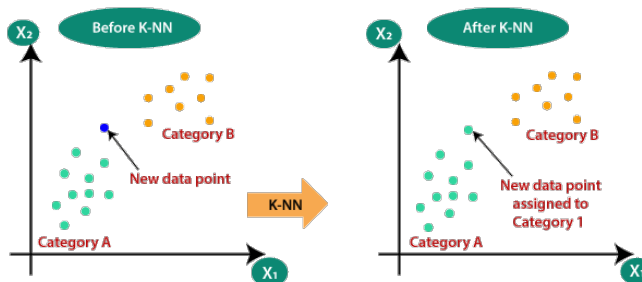


Figure 6.1: Before and after k-NN

K-NN algorithm:

1. Choose the number  $K$  of neighbours
2. Calculate the distance of the  $K$  number of neighbours
3. Select the  $K$  nearest neighbours according to the calculated distance metric.
4. Among the  $k$  neighbours, count the number of data points in each class.
5. Assign the new data points to the class for which the number of neighbours is the highest.

KNN is also called a lazy learner because it does not really do anything during the training phase. It just remembers all the data because pre-computing distances would be too expensive, and then uses it when predicting a new instance. This means that we are quite fast in the training phase, but slow in the testing phase.

**Determine the distance metric** KNN relies heavily on the concept of distance between points defining the neighbourhood, which means two main things:

- the choice of distance measure strongly affects the behaviour of the model; commonly used measures are Minkowski distance, Manhattan distance, Euclidean distance and Hamming distance.
- the model is sensitive to outliers and noise in the training data.

**Determine the K-neighbours** The algorithm also relies on the value of the hyperparameter K. The k value in the k-NN algorithm determines how many neighbours are checked to determine the classification of a particular point. Determining k can be a balancing act, as different values can lead to over or under-fitting. Lower values of k can result in high variance but low bias, while larger values of k can result in high bias and lower variance. It is recommended to use the grid search with cross-validation method to find the optimal k for the dataset.

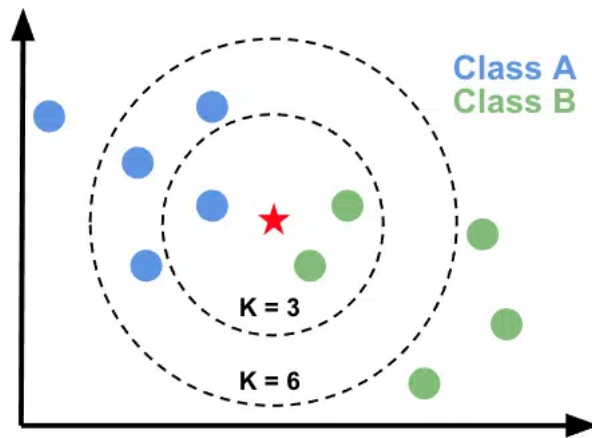


Figure 6.2: In this chart, when  $k=3$  the prediction will be green, and when  $k=6$  the prediction will be blue.

**K-NN and Naive Bayes** A fundamental difference between the K-NN classifier and the Naive Bayes classifier is that the former is a discriminative classifier, while the latter is a generative classifier. More specifically, the K-NN classifier is a supervised lazy classifier that has local heuristics. Since it is a lazy classifier, it is difficult to use it for real-time predictions. Naive Bayes is an eager learning classifier and it is much faster than K-NN. Therefore, it can be used for real-time predictions. Usually, the Naive Bayes classifier is used when filtering email spam.

### 6.1.2 Implementation

Here are the implementation of the K-Nearest Neighbors Classifier.



```
def fit(self, X_train: pd.DataFrame, y_train: pd.DataFrame | np.ndarray) -> KNN:
    """
    Checks X and y for consistent length, and convert them to ndarray if necessary
    the shape of X is expected to be (n_samples, n_features)
    the shape of y is expected to be (n_samples, 1)
    :param X_train: Training samples.
    :param y_train: Training labels.
    :return:
    """
    X_train, y_train = check_X_y(X_train, y_train)
    self.X_train = X_train
    self.y_train = y_train

    return self
```

```
def predict(self, X_test: np.ndarray) -> np.ndarray:
    y_pred = []

    # train set as rows, test set as columns
    # store the distances, sorted in ascending order for each test sample column
    if self.metric == "euclidean":
        distances = np.argsort(euclidean_distances(self.X_train, X_test), axis=0) # Euclidean distance
    else:
        distances = np.argsort(manhattan_distances(self.X_train, X_test), axis=0) # Manhattan distance

    # compute distances
    for i in range(X_test.shape[0]): # for each test sample
        # get k nearest neighbors, label
        # the i-th column represents the distances between the train set samples and the i-th test sample
        neighbors = []
        for nbr in range(self.k):
            neighbors.append(self.y_train[distances[:, i][nbr]])

        # compute predictions by getting the mode
        y_pred.append(st.mode(neighbors, axis=None, keepdims=False).mode)

    return np.array(y_pred)
```

## 6.1.3 Model Training

### 6.1.3.1 Hyperparameters Tuning

The following hyperparameters were chosen to perform the grid search on:

- k (number of neighbors)
- metric

GridSearchCV was run using a train set of 7000 samples.

```
Tuning

1 # automatic parameters tuning
2 knnc1sf=KNN()
3 properties={
4     "k" : [5,10,15,25,35,50],
5     "metric" : ['euclidean','manhattan']
6 }
7
8 tuned_knnc1sf = ms.model_selector(estimator=knnc1sf,properties=properties,scoring="accuracy",cv=10,verbose=5,jobs=4,
9                                   x_train=x_data_small,y_train=y_data_small)

Fitting 10 folds for each of 12 candidates, totalling 120 fits

2022-12-28 15:52:42.621 | INFO | src.utilities.model_selection:model_selector:26 - --- 7622.17564281355 seconds ---
```

### 6.1.3.2 Best Estimator

The grid search took 7622 seconds to execute, and the best hyperparameters are:

- k = 5
- metric = 'euclidean'

```
1 print(tuned_knnc1sf.best_estimator_)
2 print(tuned_knnc1sf.best_score_)

KNN()
0.940857142857143
```

## 6.1.4 Train

The model was then trained using the best parameters on the whole train set (60000 samples). This took 0.82 seconds.

```
Model Training

1 # best parameters from automatic parameters tuning
2 start_time = time.time()
3 knn_clsif = KNN(**tuned_knnclsf.best_params_)
4 knn_clsif.fit(data_split.x_train, data_split.y_train)
5 logger.info("Training: - %s seconds -" % (time.time() - start_time))

2022-12-28 15:52:45.987 | INFO | __main__:<module>:5 - Training: - 0.8279318889589277 seconds -
```

### 6.1.4.1 Test Score

The testing took 884 seconds giving an accuracy of 97.07%

```
Performance

1 start_time = time.time()
2 knn_test_pred = knn_clsif.predict(data_split.x_test)
3 logger.info("Prediction: - %s seconds -" % (time.time() - start_time))
4 knn_test_eval = eva.Evaluation(y_true=data_split.y_test, y_pred=knn_test_pred)

2022-12-28 17:23:00.629 | INFO | __main__:<module>:3 - Prediction: - 884.8449940681458 seconds -

1 print("Testing:")
2 knn_test_eval.acc_eval()

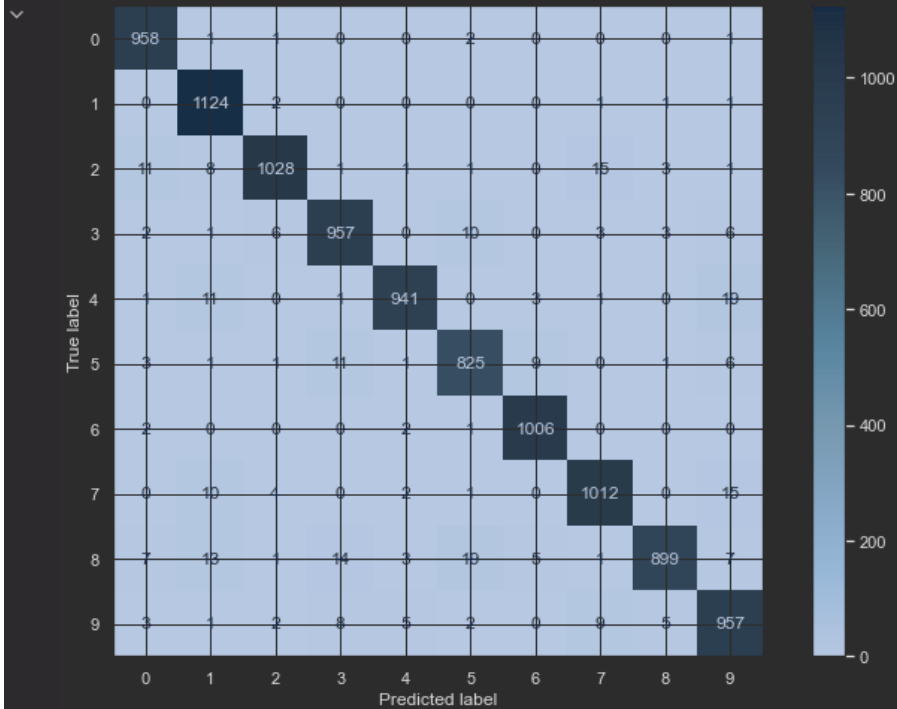
Testing:
-----Model Evaluations:-----
Accuracy score: 0.9707
```

### 6.1.4.2 Confusion Matrix

Below is the confusion matrix for the K-Nearest Neighbors Classifier. It can be noted the labels 1 can be classified easily, and the one classified correctly the fewest is 5.

## Confusion Matrix

```
1 knn_model.test_eval.conf_mat()
```



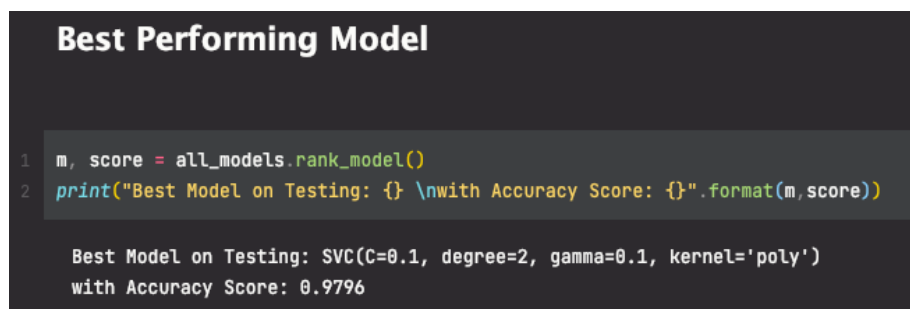
# 7 Conclusion

## 7.1 Model Comparison

### 7.1.1 Model Accuracy

As we can see from the models' confusion matrix, it seems that each model performs better when the classification includes instances of 1s, while the accuracy decreases for instances of 5s. This is because some digits are more similar in shape to other digits, such as 0 and 6, 2 and 9, or 3 and 5. So we could say that the models are the same in this behaviour.

When it comes to overall accuracy, the best model is the Polynomial Kernel of Degree 2 SVM. But as we can see from the second picture, its accuracy is close to the K-Nearest Neighbours Classifier. Other models might perform slightly worse, but we cannot be sure of this result because the grid search did not use the entire training dataset due to some limitations imposed by the machine used for the assignment, and the hyperparameters chosen also play a role in this.



```
Best Performing Model

1 m, score = all_models.rank_model()
2 print("Best Model on Testing: {} \nwith Accuracy Score: {}".format(m, score))

Best Model on Testing: SVC(C=0.1, degree=2, gamma=0.1, kernel='poly')
with Accuracy Score: 0.9796
```

Figure 7.1: Best model.

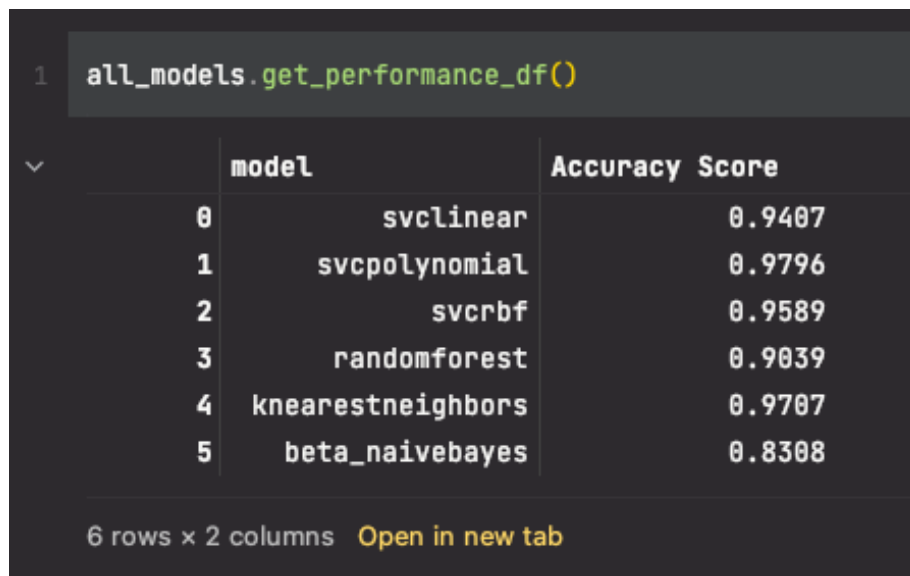


Figure 7.2: Score of all the models.

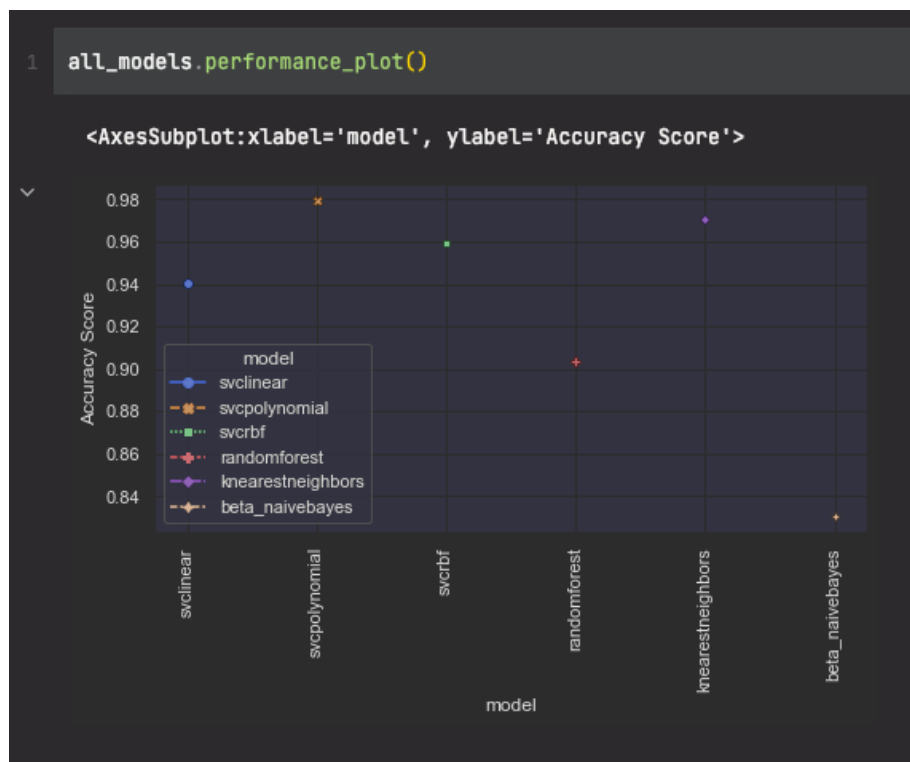


Figure 7.3: Comparison of all the scores.

## 7.1.2 Model's Prediction and training times:

The training and prediction times are given below:

Training:

- 352 - SVM Linear Kernel

- 159 - SVM Polynomial Kernel of Degree 2
- 177 - SVM RBF Kernel
- 90 - Random Forest
- 3.4 - Naive Bayes Classifier
- 0.82 - K-Nearest Neighbours

Testing:

- 153 - SVM Linear Kernel
- 42 - SVM Polynomial Kernel of Degree 2
- 67 - SVM RBF Kernel
- 1.9 - Random Forest
- 231 - Naive Bayes Classifier
- 884 - K-Nearest Neighbours

When it comes to testing and prediction time, the most balanced model is the Random Forest. All three SVMs take some time in both cases, while Naive Bayes and k-NN are fastest in training but slowest in prediction due to their implementation. In fact, the complexity of Naive Bayes for training is almost linear and k-NN is also known as a lazy model that does not perform any computations in the training phase but just remembers the dataset. On the other hand, Random Forest is so fast because it benefits from parallelization.

It should be noted that Naive Bayes and k-NN are implemented by students, which means they are not very efficient. By using the Numpy library, they would probably be much better at both prediction and testing.

# List of Figures

2.1	Example of 5 fold cross validation. . . . .	8
2.2	Example of a binary classification. . . . .	9
6.1	Before and after k-NN . . . . .	29
6.2	In this chart, when k=3 the prediction will be green, and when k=6 the prediction will be blue. . . . .	30
7.1	Best model. . . . .	34
7.2	Score of all the models. . . . .	35
7.3	Comparison of all the scores. . . . .	35

# List of Tables