Report Homework 1 Machine Learning

Edoardo Caciolo 1793918

17 November 2023

List of abbreviations

Acc Accuracy

CV Cross Validation

DT Decision Tree

F1 F1-Score

KNN K-Nearest Neighbors

LR Logistic Regression

Pre Precision

Rec Recall

RF Random Forest

SVM Support Vector Machine

XGB XGBoost

Contents

Chapter 1

Introduction

This first chapter will describes the scope of the project and the algorithms used.

1.1 Project Scope

The aim of this project is to solve a classification problem, on two different datasets with the same structure. Both training datasets consist of `50000` observations, differing in the size of features: the first one `(dataset1.csv)` has `100` fetutures for each label, the second one `(dataset2.csv)` has `1000` each. It was chosen to solve the problem using a multiclass classificatory, rather than a regressive, approach because the labels are all discrete (values from `0` to `9`). The choice of alogorithm, used for the final prediction of the two `blind_test`, was weighted on the basis of some hardware computation parameters as well as, clearly, evaluation metrics.

`dataset1.csv` → `blind_test1.csv`
`dataset2.csv` → `blind_test2.csv`

1.2 Alogorithms

The classification algorithms tested are:

1. Logistic Regression
2. Decision Tree
3. Random Forest
4. Support Vector Machine
5. K-Nearest Neighbors
6. XGBoost

All of these were tested as without Cross-Validation.

1.2.1 Logistic Regression

This type of statistical model is used for classification and predictive analysis. It estimates the probability of an event occurring on a binary basis. In the simplest case, *Logistic Regression* uses the Logit function to model the relationship between a set of independent variables and the probability of belonging to one of the categories. The logit function transforms the linear relationship, between the independent variables and the logarithm of the probabilities, ensuring that the result is between 0 and 1. Although *Logistic Regression*, by default, is limited to two class classification problems, it is also possible to apply this algorithm to the datasets under consideration. This extension is done through the *Multinomial Logistic Regression*. It is a simple and interpretable model; however, it can suffer from underfitting if the data are complex and nonlinear.

1.2.2 Decision Tree

Decision Tree learning is a strategy that uses *"dividi et impera"* logic, to create hierarchical decision trees. This algorithm is based on iterative search for the best separation points, to divide the data into specific classes. When a tree becomes too large and too complex, there is poor representation of data in some parts of the tree, which can cause overfitting (*data fragmentation*) problems. Decision trees prefer smaller structures, following *Occam's Razor principle of parsimony*, seeking the simplest explanation and avoiding unnecessary complexity. However, in addition to being able to handle various types of data (e.g. continuous and discrete), by converting continuous values to categorical values through the use of thresholds, they can also handle datasets with missing values (problematic for other classifications such as *Naïve Bayes*). The best attribute at each node is calculated through two main methods; *information gain* and

Gini impurity. They help assess the quality, of each test condition, and how well it will classify samples into a class.

1.2.3 Random Forest

To maintain accuracy, Decision Trees can be integrated into a set using *Random Forest* algorithms; this produces more accurate results when individual structures are not closely related. The *Random Forest* algorithm consists of a set of decision tree structures; each tree structure is a set consisting of a data sample, taken from a training set with replacement, called a bootstrap example. Of this training sample, one third is set aside as test data, known as the *oob* (out-of-bag) sample. The method has good accuracy and resistance to overfitting, but can be computationally intensive on large data sets.

1.2.4 Support Vector Machine

Support Vector Machine is a robust classification technique, that optimises the predictive accuracy of a model, without overfitting the training data. SVM is particularly suitable for analysing data, with very high numbers of predictor fields. SVM maps the data to a high-dimensional feature space, so that it is possible to categorise data points, even when the data cannot be separated linearly in any other way. A separator is found between the categories and then the data is transformed, so that the separator can be plotted as a hyperplane. Afterwards, the characteristics of the new data, can be used to predict the group to which a new record should belong. It can be sensitive to the choice of kernel and requires a good optimisation of hyperparameters.

1.2.5 K-Nearest Neighbors

The *K-Nearest Neighbours* algorithm uses proximity to make classifications, about the clustering of a single data point. It is based on the assumption that similar points can be found close to each other. For classification problems, a class label is assigned on the basis of a majority vote. To determine which data points are closest to a given query point, it will be necessary to calculate the

distance, between the query point and the other data points. The applicable distance matrices are: *Euclidean*, *Manhattan*, *Minkowski* and *Hamming Distance*. Although it is an easy to implement and adaptable algorithm, it does not scale well. It, also, does not work well with high-dimensional data inputs. Finally, even though it requires only a few hyperparameters, the choice of K (value defines how many neighbours will be checked, to determine the classification of a specific query point) can greatly influence the training result.

1.2.6 XGBoost

The *XGBoost* is an algorithm based on the Gradient Boosting method. In this approach, several weak models (in this case regular decision trees) are built iteratively, with each trying to correct errors in the previous models. Each tree is trained to predict the residual (difference between the actual value and the current predicted value). An objective function is also used, which must be minimised during training. This function includes two main terms: the loss function, that measures the discrepancy between model predictions and actual values, and a regularisation term to control model complexity and prevent overfitting. The *XGBoost* algorithm is designed to operate in parallel, making it fast when training on large datasets. It can also, automatically, handle missing values during model training. Despite all this, however, it can be sensitive to data noise and requires careful adjustment of hyperparameters.

1.2.7 Cross Validation

Cross Validation is a technique used, in Machine Learning, to eliminate the problem of overfitting in training sets. It is based on training on k subsets of the available input data and evaluating them against a complementary subset of them. One of the most efficient *Cross Validation* methods is *k-fold Cross-Calidation*. This divides the input data into k subsets (also known as folds). A model is trained on *k-1* subsets; then the model is evaluated on the subset that was not used for training. This process is repeated k times, each time with a different subset reserved for evaluation.
Five folds were used for this project (`k=cv=5`).

Chapter 2

Training

This chapter will explain the analysis techniques used and the partial results obtained. These will be preparatory to the choice of the optimal algorithm, used for the final prediction of the two `blind_tests`.
For the sake of brevity, the source code `Training_Dataset.ipynb` will be omitted; all results shown were produced by this script.

2.1 Methodology

For all six algorithms tested, the hyperparameters were chosen through the use of the `GridSearchCV` function of the `sklearn.model_selection` library. This trains the specified algorithm with the dataset, under test by iterating a permutation of several chosen hyperparameters. At the end of the process it generates a `<class.'dict'> best_params_[*]`, which is passed as an object to the corresponding algorithm.
Due to the considerable use of computational resources encountered, the optimal hyperparameters were found for all algorithms used with the exception of `Logistic Regression`. In view of the fact that the extremely high computation time does not justify such a search in this specific case, it was decided to leave the default hyperparameters for this one.
The resulting hyperparameters are shown below:

```
[1]: Decision Tree: Time 0m 34 s
{'max_depth': 15, 'min_samples_leaf': 2, 'min_samples_split': 2}
```

```
[2]: Random Forest: Time 4m 32s
{'max_features': 'log2', 'min_samples_split': 10, 'n_estimators': 200}
[3]: SVM: Time 17m 6s
{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
[4]: KNN: Time 0m 26s
{'metric': 'manhattan', 'n_neighbors': 9, 'weights': 'distance'}
[5]: XGBoost: Time 3m 12s
{'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
```

For the preprocessing of the data, standardisation was applied via the `Standard Scaler` function of the `sklearn.preprocessing` library. This standardises the features by removing the mean and scaling, so that the variance is unity. This process helps to improve the performance of the algorithm, reduceing training time.

Another preventive analysis was to verify that the dataset was balanced.

C	0	1	2	3	4	5	6	7	8	9
N	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000

Table 2.1: `dataset1` Class Distribution

C	0	1	2	3	4	5	6	7	8	9
N	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000

Table 2.2: `dataset2` Class Distribution

The results show a perfect balance of the two datasets. Each class covers exactly 10% of the total observations. Otherwise, algorithms of oversampling (or undersampling), class weighting and synthetic data generation would have had to be applied. This is to avoid problems of biased learning and overfitting.
Given the large number of observations, it was decided to split the training dataset into training and evaluation data (`X_train`, `Y_train`, `X_test`, `Y_test`), with a 5:1 ratio.

At this point, the two datasets were trained. For all six algorithms used, training was performed with and without *Cross-Validation*. The main classification eval-

uation metrics (*Accuracy*, *Recall*, *Precision*, *F1 Score*, *Confusion Matrix*) were extracted. In classification tasks, metrics such as accuracy, precision, recall and F1 score are commonly used to evaluate the performance of machine learning models. Accuracy represents the percentage of correct predictions out of the total number of predictions made by the model; precision and recall are useful for identifying false positives and false negatives, respectively. The F1 score is the harmonic mean between accuracy and recall and is particularly useful when dealing with unbalanced classes. Furthermore, it was considered important, in order to choose the optimal algorithm, also extract data on the computational power used; this includes: calculation time, data on RAM and CPU utilisation. The trade-off in the evaluation of these measures, makes it possible to choose the best algorithm, depending on the problem to be addressed.

2.2 Results

Below are the two tables with the results, about the training, obtained through the six algorithms (twelve in total) on both datasets [1].

Training Without Cross-Validation							
Algorithm	Acc [%]	F1 [%]	Rec [%]	Pre [%]	Time [s]	CPU [%]	RAM [GB]
LR	98.65	98.66	98.66	98.66	3.84	20.0	12.12
DT	97.48	97.50	97.50	97.50	4.17	10.2	12.22
RF	98.61	98.62	98.62	98.62	40.42	12.5	12.17
SVM	98.69	98.70	98.70	98.70	4.76	14.3	13.32
KNN	98.58	98.59	98.59	98.59	2.39	95.5	13.27
XGB	98.57	98.58	98.58	98.58	5.59	18.3	13.13
Training With Cross-Validation							
Algorithm	Acc [%]	F1 [%]	Rec [%]	Pre [%]	Time [s]	CPU [%]	RAM [GB]
LR	98.79	98.79	98.79	98.79	197.03	23.4	12.25
DT	97.78	97.74	97.74	97.75	43.35	12.6	12.17
RF	98.78	98.79	98.79	98.79	421.86	12.5	13.26
SVM	98.86	98.86	98.86	98.86	39.08	71.4	13.27
KNN	98.80	98.80	98.80	98.80	22.50	99.8	13.10
XGB	98.70	98.70	98.70	98.70	21.45	14.3	13.31

Table 2.3: Training Results `dataset1`

[1] The best results for each measurement are highlighted in light grey.

Training Without Cross-Validation							
Algorithm	Acc [%]	F1 [%]	Rec [%]	Pre [%]	Time [s]	CPU [%]	RAM [GB]
LR	96.45	96.47	96.48	96.46	62.45	14.3	17.56
DT	95.55	95.57	95.58	95.57	71.90	28.6	16.64
RF	97.07	97.09	97.10	97.08	66.40	28.6	17.64
SVM	96.96	96.97	96.99	96.96	57.59	16.9	17.47
KNN	96.86	96.87	96.89	96.86	21.60	96.3	17.60
XGB	96.83	96.85	96.86	96.84	40.83	22.9	17.55
Training With Cross-Validation							
Algorithm	Acc [%]	F1 [%]	Rec [%]	Pre [%]	Time [s]	CPU [%]	RAM [GB]
LR	97.12	97.12	97.12	97.11	475.49	42.9	17.78
DT	95.60	95.61	95.60	95.63	642.33	16.7	17.69
RF	97.34	97.33	97.34	97.34	1037.42	31.6	17.23
SVM	97.35	97.35	97.35	97.35	525.42	17.7	17.59
KNN	97.09	97.09	97.09	97.08	165.33	99.7	17.48
XGB	97.12	97.12	97.12	97.12	428.60	29.1	20.50

Table 2.4: Training Results `dataset2`

One thing that is immediately noticeable, for both datasets, is that the algorithms return equal evaluation metrics values (with the exception of Accuracy), with reference to their specific subtest. This is a clear indicator of the perfect balance of the various classes, as shown above.

Regarding the `dataset1`, (Tab. 2.3) shows how long the execution time of the *Random Forest* algorithm is (up to `10.84` times slower than the fastest), despite being the best, in terms of computational resources, using Cross-Validation and the second best without it. On the contrary, algorithms such as *XGB* and *KNN* are extremely fast (in absolute terms or at least compared to other algorithms); the latter, however, uses considerable hardware resources. Finally, the *SVM* algorithm achieve the highest score in terms of accuracy, (`98.86%` with and without Cross-Validation); however, with the use of Cross-Validation, it has showed a very high CPU utilisation, but with a good execution time.

Concerning `dataset2`, the (Tab. 3.2) shows that the highest accuracy was achieved by the *Random Forest* and *SVM* algorithms, without and with Cross-Validation, respectively. Similar to dataset1, the *KNN* algorithm proved to be the fastest, but at the expense of the high computing power used. Not sur-

prisingly, the method that took the longest time to run was *Random Forest* (especially with the use of Cross-Validation).

In general, for both datasets, the RAM was not overloaded, despite the fact that it was used to a large extent. It is clear, however, that the calculation parameters depend entirely on the hardware used and thus depend, in this case, on the PC used.
Another interesting aspect is that the accuracies of `dataset2` are homogeneously lower than the corresponding ones of `dataset1`. This despite both having very similar data. Likewise, it is evident how calculation times are drastically lengthened. This is solely related to the length of the feture of each observation (`10` times larger in `dataset2`). In a very empirical way, you can check if there is any proportionality between the calculation time and the size of the features, in the two datasets:

$$\gamma_k = \frac{\sum_{i=1}^{12} \text{Time}_i}{N_{fk}} \tag{2.1}$$

Where k is the dataset considered (*1* or *2*), $Time_i$ is the column of time from the chosen dataset, N_{fk} is the number of features in the dataset. The results are:

$\gamma_{\texttt{dataset1}} \simeq 8.06$
$\gamma_{\texttt{dataset2}} \simeq 3.60$

Thus, there is not much proportionality between the two coefficients γ_i. On the contrary, this shows that on average `dataset2` takes less time to process than the other one. This is, more or less, clear looking at the values of the models with Cross-Validation of the two datasets.

2.2.1 Graphs

Below are the Confusion Matrices 2.1 of the algorithms without Cross-Validation, applied to the `dataset1` (Fig. 2.1) and to the `dataset2` (Fig. 2.2).

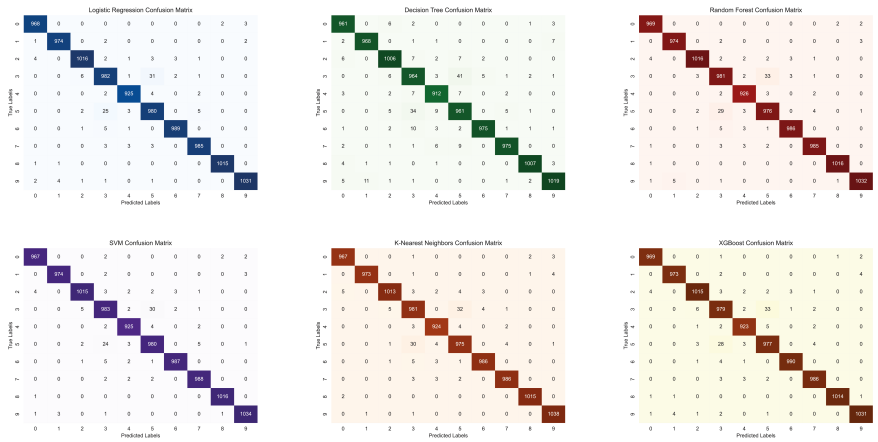

Figure 2.1: Confusion Matrices `dataset1`

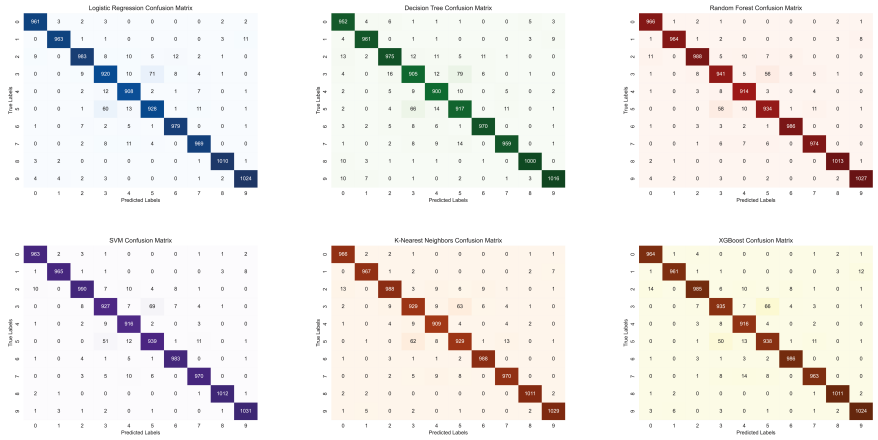

Figure 2.2: Confusion Matrices `dataset2`

For all algorithms, on both datasets, the biggest training problem occurred on the classification between labels `3` and `5`. In fact, it can be seen there was a discrete tendency to invert these two classes during training. This is evident from the observation of the individual tables.

Chapter 3

Prediction

In this chapter, the final choice of algorithms from those analysed will be dealt with, based on the considerations made in the previous chapter. These will then be used to give a prediction on the respective `blind_test1` and `blind_test2`.

3.1 Choice and application

On the basis of the considerations made thus far, the choice of the algorithm to be used was made on the basis of a compromise between all the performances emerging from the training. The most immediate choice would have been to favour the algorithm that, on paper, seemed to be the most suitable; the one with the highest accuracy: *SVM* with Cross-Validation for both datasets (98.86% and 97.35%). However, the analysis of this metric alone is not enough.

In fact, as far as `dataset1` is concerned, the *SVM* without Cross-Validation not only has a high accuracy, but also a sufficiently low computation time; more importantly, it requires completely acceptable computational resources in contrast, for example, to the `KNN`. The *XGBoost* algorithm with Cross-Validation also showed very good performance, but for the same accuracy (0.01% more) and CPU consumption, it is not worth using five times the calculation time of the *SVM* without Cross-Validation.

For `dataset2`, on the other hand, the use of Cross-Validation showed too long calculation times in all cases (with the exception of *KNN* but at the expense

of maximum CPU consumption). Since it made no great difference in terms of accuracy, compared with its non-use, Cross-Validation was therefore discarded. The following considerations will therefore be made with reference to the algorithms without its use. The *XGBoost* algorithm has the lowest execution time (if we exclude *KNN* due to its usual resource consumption) and good hardware handling. The same can be said for the *Random Forest* algorithm albeit with longer but still acceptable times. As far as accuracy is concerned, the data shows a difference of 0.24% in favour of the latter model; a difference that is not excessive but still present. However, it is known from the literature that, as the *Random Forest* is a *bagging* algorithm, it uses tree aggregation to reduce variance and control overfitting. It is therefore, in theory, less prone to this problem than *XGBoost*.

For all these reasons, the final decision for the final prediction was:

`dataset1.csv` → `SVM without CV` → `blind_test1.csv`
`dataset2.csv` → `Random Forest without CV` → `blind_test2.csv`

Despite this choice, as a proof of concept, it was decided to apply all algorithms with and without cross-validation to both datasets.
The interest was to see how similar the predictions of each algorithm with and without Cross-Validation were.
Below are the comparative tables of the results obtained:

Modello	Equal results [%]
LR	82.02
DT	36.01
RF	74.29
SVM	87.59
KNN	90.85
XGB	69.45

Table 3.1: Percentage of equality between prediction results with and without Cross - Validation [`dataset1`]

Modello	Equal results [%]
LR	92.96
DT	62.40
RF	95.57
SVM	56.21
KNN	97.22
XGB	94.79

Table 3.2: Percentage of equality between prediction results with and without Cross - Validation [`dataset2`]

Chapter 4

Conclusions

As is well known, in the field of Machine Learning there is no ideal algorithm, perfect for every situation. In fact, analysis of the data collected showed that the final choice was the consequence of a compromise between results, performance and theoretical knowledge of the algorithm. Even two data sets, structurally very similar, required the use of two different but optimal predictive methods for the specific case.

The partial predictions of all illustrated algorithms, applied to both `blind_tests`, are briefly reported.

Predictions `blind_test1`												
	Without CV						With CV					
`id`	LR	DT	RF	SVM	KNN	XGB	LR	DT	RF	SVM	KNN	XGB
0	3	3	3	3	3	3	3	4	3	3	3	3
1	8	8	8	8	8	8	1	4	8	8	8	8
2	8	8	8	8	8	8	0	4	8	8	8	6
3	0	0	0	0	0	0	0	8	8	0	0	0
4	6	6	6	6	6	6	6	2	6	6	6	6
5	6	6	6	6	6	6	6	2	6	6	6	6
...	...	...	...	...	...	...	...	...	...	...	...	...
9997	5	5	5	5	5	5	2	5	5	5	5	5
9998	1	1	1	1	1	1	6	2	9	5	5	5
9999	7	7	7	7	7	7	7	4	7	7	7	7

Table 4.1: Predictions Results of `blind_test1`

Predictions `blind_test2`												
	Without CV						With CV					
`id`	LR	DT	RF	SVM	KNN	XGB	LR	DT	RF	SVM	KNN	XGB
0	3	3	3	3	3	3	3	5	3	5	3	3
1	8	8	8	8	8	8	8	8	8	8	8	8
2	8	8	8	8	8	8	8	8	8	8	8	8
3	0	0	0	0	0	0	0	0	0	0	0	0
4	6	6	6	6	6	6	6	2	6	2	6	6
5	6	6	6	6	6	6	6	2	6	2	6	6
...	...	...	...	...	...	...	...	...	...	...	...	...
9997	5	5	5	5	5	5	5	3	5	5	5	5
9998	1	1	1	1	1	1	1	9	1	9	1	1
9999	7	7	7	7	7	7	7	7	7	2	7	7

Table 4.2: Predictions Results of `blind_test2`

The complete predictions can be found in the files:

`d1_1793918.csv` (for `dataset1` - *SVM* no CV)
`d2_1793918.csv` (for `dataset2` - *RF* no CV)

The complete source code is available in the `ML_Homework1_1793918.zip` and contains the files:

`load_data.py`
`ML_Homework1_SourceCode_1793918.ipynb`

Appendix A

Source Code

A.1 Libraries used

```
import psutil, time
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
```