REPORT HOMEWORK 1 - MACHINE LEARNING

Author
Mattia Maffongelli 1941528

Contents

1 Project Overview

The objective of this assignment is to address two distinct **10-class classification** problems associated with disparate datasets—one characterized by relative simplicity, and the other presenting a more intricate challenge.

1.1 Datasets

The two datasets primarily differ in their **input space dimensions**: the first dataset has an input space dimension of **100**, whereas the second dataset has a dimension of **1000**. Nevertheless, both share a common structure:

Index	X	Y
0	$[V^0_1, V^0_2, V^0_3, V^0_4, \dots V^0_j, \dots, V^0_d]$	C_k
…	…	…
i	$[V^i_1, V^i_2, V^i_3, V^i_4, \dots V^i_j, \dots, V^i_d]$	C_k
…	…	…
N	$[V^N_1, V^N_2, V^N_3, V^N_4, \dots V^N_j, \dots, V^N_d]$	C_k

Figure 1: Training files structure

In the "X" column, we have sets of features. For the first dataset, each set has 100 features, and for the second dataset, it's 1000 features. These features represent our input data. The "Y" column shows the labels for each features vector. Lastly, the index column is just a count of the samples in the training file. In this context, I utilized the given "load-data.py" function to retrieve both the X and Y columns. Subsequently, I engaged in **preprocessing** tasks to refine and prepare the data for further analysis.

1.2 Pre-processing

In the dataset pre-processing phase, I use at first the **Standard Scaler** function from Sklearn to standardize the features vectors. The primary objective of Standard Scaler is to make these vectors comparable and improve the **convergence** of machine learning algorithms. Basically, this procedure scales each feature such that their distribution has a mean of zero and a standard deviation of one.

Subsequently, in addition to this approach, I opt to utilize the "**normalize**" function found in the Sklearn library. The primary distinction between the two methods lies in the fact that, while the normalize function operates on each observation by scaling it to have

a norm equal to 1—useful when the absolute magnitude of features is not significant but the direction or proportionality of features relative to each other is crucial—the Standard Scaler function directly manipulates the **distribution** of the data.

1.3 Metrics

The metrics I used to evaluate the models are: **accuracy, precision, recall, f1-score.** Additionally, to provide a visual representation of the outcomes, I used also the **confusion matrix**.

1.3.1 Accuracy

The **Accuracy** metric provides a quick overview of whether a model is learning well and its overall performance. However, it lacks detailed insights into its application to specific problems, so using only this metric can be problematic, especially when there's a significant class imbalance in the data (or a huge number of classes). The related function that I used is:

```
sklearn.metrics.accuracy_score(y_true, y_pred, *, normalize=True, sample_weight=None)
```

It return the fraction of correctly classified samples, simply using the formula:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of total predictions}} \tag{1}$$

1.3.2 Precision and Recall

The **Precision** metric helps when the costs of false positives are high, while the Recall metric helps when the cost of false negatives is high.

In particular, Precision is defined as the number of true positives over the number of true positives plus the number of false positives:

$$\text{Precision} = \frac{T_p}{T_p + F_p} \tag{2}$$

While, **Recall** is defined as the number of true positives over the number of true positives plus the number of false negatives:

$$\text{Recall} = \frac{T_p}{T_p + F_n} \tag{3}$$

For this metrics I used two functions of the Sklearn Library:

```
sklearn.metrics.precision_score(y_true, y_pred, *, labels=None, pos_label=1,
average='binary',
sample_weight=None, zero_division='warn');

sklearn.metrics.recall_score(y_true, y_pred, *, labels=None, pos_label=1,
average='binary',
sample_weight=None, zero_division='warn');
```

1.3.3 F1-Score

The **F1-score** is a comprehensive metric that assesses a model's accuracy by taking into account both precision and recall. Its primary goal is to minimize both the false positive rate and the false negative rate, aiming to achieve the highest possible F1-score. An F1-score of 1 is considered perfect, indicating a model with ideal precision and recall, while a score of 0 suggests a complete failure of the model. It is defined by:

$$\text{F1-Score} = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

1.3.4 Confusion Matrix

The **confusion matrix** is a 2x2 table that provides a summary of the results produced by a classification model. It compares the prediction through four components:

1. True Positives (TP): The number of instances that were correctly predicted as positive.
2. True Negatives (TN): The number of instances that were correctly predicted as negative.
3. False Positives (FP): The number of instances that were predicted as positive but were actually negative.
4. False Negatives (FN): The number of instances that were predicted as negative but were actually positive.

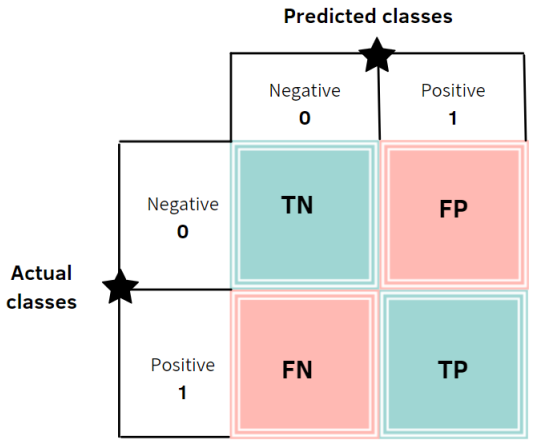

Figure 2: Example of Confusion Matrix

2 First Dataset - Classification

2.1 Split data

This section remains the same for both the first and second datasets. It involves dividing the data, which is loaded using the given "load-data.py" function, into x-train, y-train, x-

test, and y-test. This division is accomplished using a function from the Scikit-learn library called "train-test-split".

```
sklearn.model_selection.train_test_split(*arrays, test_size=None, train_size=None,
random_state=None, shuffle=True,
stratify=None)[source]
```

The main parameters include the "arrays", where the X column and Y column of the dataset are passed, the "test-size" determining the proportion of data used for the test set (in my case I use 0.2), and finally, the "random-state," serving as the seed for the random number generator during the split. As mentioned before, I process the data with Standard Scaler or "Normalize", before to put them in the function, for better efficiency.

2.2 Models

2.2.1 Decision Tree

The initial model I've selected is the **Decision Tree** model. This is a non-parametric supervised learning technique employed for both classification and regression. The aim is to build a model that predicts the value of a target variable by learning straightforward decision rules derived from the features in the data.

The reasons of this choice are based on different considerations: Decision Trees are **simple** to understand and to interpret, requires little data preparation, are able to handle multi-output problems and, finally, the cost of using them is **logarithmic** in the number of data points. However, they have some **disadvantages**, like as:

- They can create over-complex trees that do not generalize the data well (overfitting);
- They can be unstable because small variations in the data might result in a completely different tree being generated;
- They provide predictions that are not smooth or continuous; instead, they offer piecewise constant approximations.

In order to use this model, I found the function in the Sklearn library:

```
sklearn.tree.DecisionTreeClassifier(*, criterion='gini', splitter='best',
max_depth=None, min_samples_split=2,
min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=None, random_state=None,
max_leaf_nodes=None,
min_impurity_decrease=0.0,
class_weight=None, ccp_alpha=0.0);
```

As I expected, it gave lower metric values because it's **not very robust**. Still, it seems to work well enough for this dataset. Indeed, the output obtained is shown in the figure 3.

As we see, the output is slightly improved by preprocessing the data through the Normalize function, although the results are still relatively similar.

I also choose to display the confusion matrix for a visual comparison with the other model (Figure 4), and we can observe that the count of accurate predictions (on the main

```
Accuracy of Decision Tree model:
 0.972
Precision of Decision Tree model:
 [0.98979592 0.98282828 0.97019231 0.92416582 0.96834817 0.93227092
 0.98185484 0.99199199 0.99284254 0.98623402]
Precision mean of Decision Tree:
 0.97205
Recall of Decision Tree model:
 [0.98377282 0.98681542 0.97206166 0.92416582 0.97219464 0.94070352
 0.98682877 0.97924901 0.99284254 0.981409  ]
Recall mean of Decision Tree:
 0.972
F1_score of Decision Tree model:
 [0.98677518 0.98481781 0.97112608 0.92416582 0.97026759 0.93646823
 0.98433552 0.98557931 0.99284254 0.9838156 ]
F1_score mean of Decision Tree:
 0.97202
```

(a) Output obtained with StandardScaler

```
Accuracy of Decision Tree model:
 0.975
Precision of Decision Tree model:
 [0.98677518 0.9889001  0.97200772 0.92376238 0.97529644 0.94580777
 0.98189135 0.9880597  0.99184506 0.9960396 ]
Precision mean of Decision Tree:
 0.97504
Recall of Decision Tree model:
 [0.98377282 0.99391481 0.97013487 0.94337715 0.98013903 0.92964824
 0.98885512 0.9812253  0.99488753 0.98434442]
Recall mean of Decision Tree:
 0.97503
F1_score of Decision Tree model:
 [0.98527171 0.99140111 0.9710704  0.93346673 0.97771174 0.93765839
 0.98536093 0.98463064 0.99336396 0.99015748]
F1_score mean of Decision Tree:
 0.97501
```

(b) Output obtained with Normalize

Figure 3: Decision Tree model's output

diagonal) is generally good, and for certain labels (e.g. "2" and "9"), the count is nearly optimal.

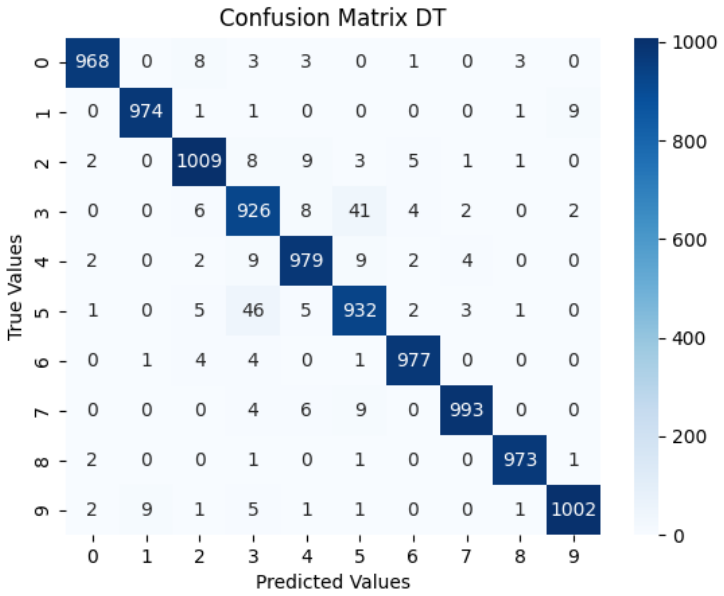

Figure 4: Decision Tree confusion matrix, dataset 1

Still, there could be better models. Let's check.

2.2.2 Perceptron

The **Perceptron** model is another simple and linear classification algorithm suitable for large scale learning. In comparison to Decision Trees models, it exhibits a more sophisticated and intricate structure. Some advantages are: it remains **simple to understand** and implement, it can be trained relatively quickly and it also can adapt to new data in real-time. However, it has limitations in handling complex patterns and, especially, the convergence is **not guaranteed** for datasets that are not **linearly separable**. In such cases, the weights may oscillate without reaching a stable solution. I used this model with the Sklearn library's function:

```
sklearn.linear_model.Perceptron(*, penalty=None, alpha=0.0001, l1_ratio=0.15,
fit_intercept=True, max_iter=1000,
tol=0.001, shuffle=True,
verbose=0, eta0=1.0,
n_jobs=None, random_state=0,
early_stopping=False, validation_fraction=0.1, n_iter_no_change=5,
class_weight=None, warm_start=False)
```

As we see, it requires some optional parameters that efficiently change the performance of the model. Despite being able to manually tweak the parameters, I opted to leverage a useful function provided by the Sklearn library, namely:

```
sklearn.model_selection.RandomizedSearchCV(estimator, param_distributions, *,
n_iter=10, scoring=None,
n_jobs=None, refit=True,
cv=None, verbose=0,
pre_dispatch='2*n_jobs',
random_state=None, error_score=nan,
return_train_score=False)[source];
```

It allows the discovery of the **optimal model parameters** by conducting a random search within a predefined set of possibilities. In my case, I choose to fine-tune using four parameters: "**penalty**['l2', 'l1', 'elasticnet']", which specifies the type of regularization; "**alpha**[0.001, 0.0001, 0.00001]", the constant multiplying the regularization; "**maxiter**[range: 100, 1000]", determining the number of iterations; and "**eta0**"[range: 0.01,0.1], the constant by which the updates are multiplied.

Previously, I attempted to utilize the "GridSearchCV" function, which is more systematic than the "RandomizedSearchCV" function. However, it demanded a significant amount of runtime due to the dataset's size. The primary distinction lies in the fact that the first method fine-tunes through all the parameters and their specified ranges, whereas the second method fine-tunes through a randomly selected subset of parameters. However, it appears to be more **efficient** compared to the Decision Tree model. In fact, the metrics I obtained are outlined in the output in the figure 5:

In this case as well, the output is better when using the Normalize function. Additionally, we can observe the result of hyperparameter tuning, which is subsequently used to train the selected model.

While, in this case, the confusion matrix is shown in the figure 6.

As we see, the number of correct predicted labels are, more or less, similar to the previous confusion matrix, but there is still a small yet meaningful improvement.

Let's see now the best model I used for this homework.

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'penalty': 'l1', 'max_iter': 700, 'eta0': 0.01, 'alpha': 1e-05}
(40000, 100)
Accuracy of perceptron model:
 0.9832
Precision of Perceptron model:
 [0.99388379 0.99591002 0.98737864 0.95504496 0.96624879 0.97713098
 0.98686869 0.99011858 0.99591002 0.98448109]
Precision mean of Perceptron model:
 0.9833
Recall of Perceptron model:
 [0.98884381 0.98782961 0.97976879 0.96663296 0.99503476 0.94472362
 0.98986829 0.99011858 0.99591002 0.99315068]
Recall mean of Perceptron model:
 0.98319
F1_score of Perceptron model:
 [0.9913574  0.99185336 0.98355899 0.96080402 0.98043053 0.96065406
 0.98836621 0.99011858 0.99591002 0.98879688]
F1_score mean of Perceptron model:
 0.98319
```

(a) Output obtained with StandardScaler

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'penalty': 'l2', 'max_iter': 900, 'eta0': 0.060000000000000005, 'alpha': 1e-05}
(40000, 100)
Accuracy of perceptron model:
 0.9856
Precision of Perceptron model:
 [0.98989899 0.9969419  0.99513619 0.9291866  0.98619329 0.98333333
 0.99588054 0.9950446  0.99388379 0.99314398]
Precision mean of Perceptron model:
 0.98586
Recall of Perceptron model:
 [0.99391481 0.99188641 0.98554913 0.9817998  0.99304866 0.94874372
 0.97973658 0.99209486 0.99693252 0.99217221]
Recall mean of Perceptron model:
 0.98559
F1_score of Perceptron model:
 [0.99190283 0.99440773 0.99031946 0.95476893 0.9896091  0.9657289
 0.98774259 0.99356754 0.99540582 0.99265786]
F1_score mean of Perceptron model:
 0.98561
```

(b) Output obtained with Normalize

Figure 5: Perceptron model's output

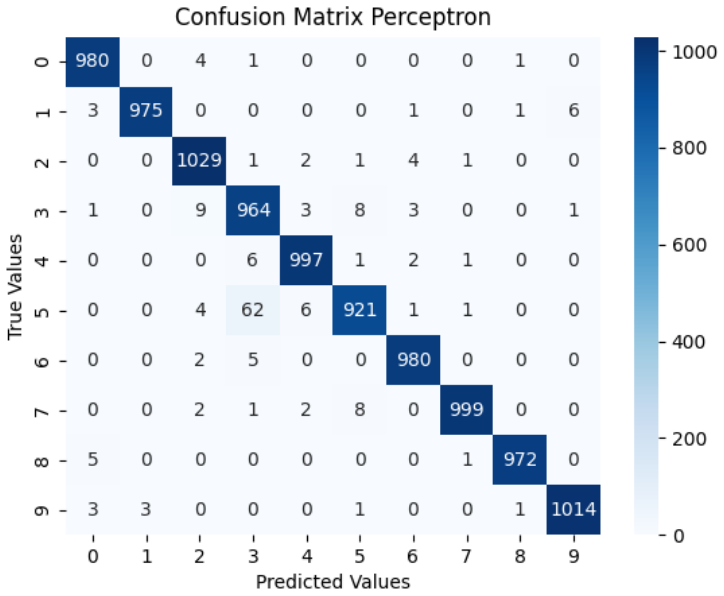

Figure 6: Perceptron confusion matrix, dataset 1

2.2.3 Kernelized SVM model

Support vector machines (SVMs) are a powerful set of supervised learning methods used for classification, regression and outliers detection. Also with this models we have advantages and disadvantages. The **advantages** are:

1. They are effective in high dimensional spaces;
2. They are also very much versatile: different Kernel functions can be specified for the decision function;
3. Still effective in cases where number of dimensions is greater than the number of samples;

However, it's crucial to address the issue of overfitting by selecting the appropriate kernel functions. These **kernel functions** can be any of the following:

- Linear;
- Polynomial, specified by parameter degree;
- Rbf, specified by parameter gamma, greater than 0;
- Sigmoid;

Naturally, therefore, it is necessary, in this case as well, to proceed with a meticulous search for the best parameters. This was accomplished through the "RandomizedSearchCV" function, which explored among the following parameters:

1. Kernel: ['linear', 'poly', 'rbf', 'sigmoid'];
2. Degree: [3, 4];
3. C (regularization parameter): [range: 0.1, 2.0];
4. Gamma: [0.1, 1, 10];

Once the optimal parameters were identified (Figure 7), I employed the "SVC" function from the sklearn library to initialize the model with these parameters. The results obtained were excellent, as evidenced by the output of the metrics (figure 8).

The most significant evidence can be observed from the confusion matrix for the utilized model, which exhibits numerous improvements compared to the previously analyzed ones.

The figure is referred to a SVM model with kernel = **RBF** and C = **1.6**, obtained by the RandomizedSearch function. However, I obtained significant results also with other kernel or other values of the parameters, demonstrating the effectiveness of this machine learning method (following there are outputs for different parameters of the model, with pre-processed data using Normalize function).

Now let's talk about the second dataset.

```
y
Data pre-processed:

[[ 1.19375851e+00 -6.53868089e-01 -1.77706533e-01 ... -9.30129158e-01
  -2.67380531e-01 -1.60982287e-01]
 [-7.32849539e-01  6.55147275e-02 -1.77706533e-01 ...  2.58997255e+00
  -5.73269397e-02 -1.60982287e-01]
 [-7.32849539e-01  1.69395749e-03 -1.77706533e-01 ...  1.92442658e+00
  -2.67380531e-01 -1.60982287e-01]
 ...
 [-7.32849539e-01  8.02923151e-01 -1.77706533e-01 ...  2.11553930e+00
  -2.67380531e-01 -1.60982287e-01]
 [ 1.11398598e-01 -8.80689150e-01 -1.77706533e-01 ... -3.27156551e-01
  -2.67380531e-01 -1.60982287e-01]
 [-2.75378960e-01 -8.80689150e-01 -1.77706533e-01 ... -7.06968762e-01
  -2.67380531e-01 -1.60982287e-01]]
(40000, 100)
(40000,)
Fitting 5 folds for each of 32 candidates, totalling 160 fits
Best parameters found: {'C': 1.6, 'degree': 3, 'kernel': 'rbf'}
0.9886

Process finished with exit code 0
```

Figure 7: Best parameters for the SVM model

```
Accuracy of SVM model:
 0.9886
Precision of SVM model (RBF, C = 1.6):
 [0.99592668 0.9949187  0.98561841 0.9668008  0.98716683 0.9786802
 0.99190283 0.99603568 0.99490316 0.99412341]
Precision mean of SVM model (RBF, C = 1.6):
 0.98861
Recall of SVM model (RBF, C = 1.6):
 [0.99188641 0.99290061 0.99036609 0.97168857 0.99304866 0.96884422
 0.9929078  0.993083   0.99795501 0.99315068]
Recall mean of SVM model (RBF, C = 1.6):
 0.98858
F1_score of SVM model (RBF, C = 1.6):
 [0.99390244 0.99390863 0.98798654 0.96923853 0.99009901 0.97373737
 0.99240506 0.99455715 0.99642675 0.99363681]
F1_score mean of SVM model (RBF, C = 1.6):
 0.98859
```

(a) Output obtained with StandardScaler

```
Accuracy of SVM model:
 0.9889
Precision of SVM model (RBF, C = 1.6):
 [0.99592668 0.99593909 0.98747592 0.9668008  0.98716683 0.9786802
 0.99091826 0.9950544  0.99693565 0.99412916]
Precision mean of SVM model (RBF, C = 1.6):
 0.9889
Recall of SVM model (RBF, C = 1.6):
 [0.99188641 0.99492901 0.98747592 0.97168857 0.99304866 0.96884422
 0.99493414 0.99407115 0.99795501 0.99412916]
Recall mean of SVM model (RBF, C = 1.6):
 0.9889
F1_score of SVM model (RBF, C = 1.6):
 [0.99390244 0.99543379 0.98747592 0.96923853 0.99009901 0.97373737
 0.99292214 0.99456253 0.99744507 0.99412916]
F1_score mean of SVM model (RBF, C = 1.6):
 0.98889
```

(b) Output obtained with Normalize

Figure 8: SVM (RBF, C = 1.6) model's output

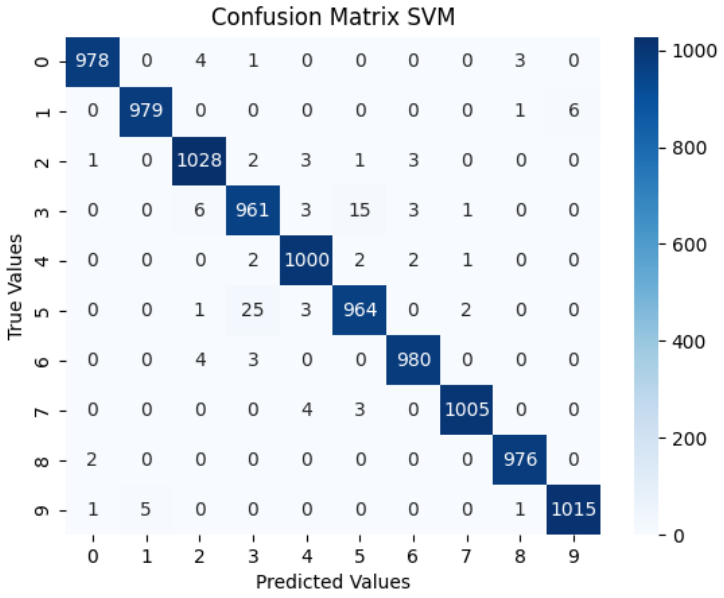

Figure 9: SVM (kernel RBF) confusion matrix, dataset 1

```
Accuracy of SVM model:
 0.9885
Precision of SVM model (poly, degree = 4, C = 1.6):
 [0.99490835 0.99493414 0.98653846 0.96676737 0.98813056 0.97474747
 0.9929078  0.9950446  0.99592253 0.99509804]
Precision mean of SVM model (poly, degree = 4, C = 1.6):
 0.9885
Recall of SVM model (poly, degree = 4, C = 1.6):
 [0.99087221 0.9959432  0.98843931 0.97067745 0.99205561 0.96984925
 0.9929078  0.99209486 0.99897751 0.99315068]
Recall mean of SVM model (poly, degree = 4, C = 1.6):
 0.9885
F1_score of SVM model (poly, degree = 4, C = 1.6):
 [0.99288618 0.99543842 0.98748797 0.96871847 0.9900892  0.97229219
 0.9929078  0.99356754 0.99744768 0.99412341]
F1_score mean of SVM model (poly, degree = 4, C = 1.6):
 0.9885
```

(a) Output SVM (Poly, degree = 4)

```
Accuracy of SVM model:
 0.9886
Precision of SVM model (sigmoid, C = 1.2):
 [0.99592668 0.99492386 0.98938224 0.96492986 0.98619329 0.97857143
 0.99092742 0.9950544  0.99591837 0.99412916]
Precision mean of SVM model (sigmoid, C = 1.2):
 0.9886
Recall of SVM model (sigmoid, C = 1.2):
 [0.99188641 0.99391481 0.98747592 0.97371082 0.99304866 0.9638191
 0.99594732 0.99407115 0.99795501 0.99412916]
Recall mean of SVM model (sigmoid, C = 1.2):
 0.9886
F1_score of SVM model (sigmoid, C = 1.2):
 [0.99390244 0.99441908 0.98842816 0.96930045 0.9896091  0.97113924
 0.99343103 0.99456253 0.99693565 0.99412916]
F1_score mean of SVM model (sigmoid, C = 1.2):
 0.98859
```

(b) Output SVM (Sigmoid, C = 1.2))

Figure 10: SVM's other output

3 Second Dataset - Classification

The second dataset on which I worked, as mentioned earlier, has a **larger** input space compared to the first. In other words, we have many more features for each sample. This introduces several additional challenges, including an increase in **computation time** and the consequent resource intensiveness, a higher risk of **overfitting**, especially if the number of samples does not grow proportionally to the number of features, and thus the need for more data to effectively train the models. Given the experience with the first dataset, I have decided to use only the **Normalize** function to preprocess the data for the second dataset. Despite this choice, I also experimented with **RobustScaler** (a function designed to make the model more robust to outliers) and StandardScaler. However, the results obtained with these alternatives were not superior to using Normalize.

These challenges have posed several problems for me, and I will describe the various attempts I made to address them below.

3.1 Hyperparameter tuning problem

The initial challenge revolved around **hyperparameter tuning** of the SVM model. Due to the dataset's extensive feature space, searching for the optimal parameters proved to be time-consuming, and, in most cases, the process never concluded, but failed in loop. This is typical for the type of dataset under analysis, and I experimented with various solutions to try to complete the process. For example, Instead of searching for parameters by analyzing each sample in the dataset, I extracted mini-batches (mostly 1000) from it through a random data **shuffle**. This approach allowed me to extract the best parameters from these **mini-batches**. I also modified certain parameters of the "RandomizedSearchCV" function, specifically reducing the number of iterations from 10 to 3, to expedite the process and obtain results more **quickly**. Despite this, the process took a considerable amount of time, but it eventually provided a partial result: similar to the first dataset, the best kernel was **RBF**, and the value of C closely resembled the C found for the first dataset. Starting from this point I tried to use the model with other values for the other parameters. Certainly, these results are **not precise** and are quite approximate, providing only a general sense of the best direction to take. Therefore, for all three models, I decided to use parameters as similar as possible to those used for the first dataset. However, I introduced a different type of for each.

3.2 Ensemble Methods

To improve the efficiency of the models, with the second dataset, I used two type of **ensemble** methods: A **Bagging** classifier is a special type of model that combines predictions from multiple simpler models. It does this by training each simpler model on different random portions of the original dataset. Then, it combines their predictions to make a final prediction. This approach helps reduce the **variability** of a single model and can be particularly **useful** when dealing with complex models. It is advantageous in scenarios with datasets containing **numerous features and samples**, like this case, for several reasons:

1. Bagging helps reduce the **variance** of a model, especially when the model is sensitive to the specific training data. In fact, it creates an ensemble that is more robust and

less prone to overfitting;

2. It enhances the **stability** of a model because it trains on different subsets, so it is less likely to be influenced by outliers or noise in the data;

3. It is also well-suited for datasets with a high number of features becuase it helps mitigate issues related to the curse of dimensionality especially it improves the performance;

A **random forest** is a special estimator that fits a number of decision tree classifiers on various subsamples of the dataset and uses averaging to improve the predictive accuracy and control overfitting. The sub-sample size is controlled with the max-samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

3.3 Models

3.3.1 Random Forest Classifier

Since the SKlearn library includes a classifier that inherently uses multiple decision trees, in this case, instead of employing a bagging method, I focused primarily on the Random Forest classifier. The results were indeed **quite favorable**, as can be seen in the Figure 11. The function that I used is:

```
sklearn.ensemble.RandomForestClassifier(n_estimators=100, *, criterion='gini',
max_depth=None, min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features='sqrt', max_leaf_nodes=None,
min_impurity_decrease=0.0, bootstrap=True,
oob_score=False, n_jobs=None, random_state=None,
verbose=0, warm_start=False, class_weight=None,
ccp_alpha=0.0, max_samples=None)
```

It's intriguing to examine the confusion matrix generated using this learning method for the second dataset, because, as you can see, the results are remarkably similar to those obtained on the first dataset without the use of an ensemble method (the values obtained are smaller as we expect, but quite similar).

```
Accuracy of Forest Classifier model:
 0.9721
Precision of Forest Classifier model:
 [0.98181818 0.99185336 0.97773475 0.90693069 0.96963761 0.94039054
 0.9828629  0.98511905 0.99487179 0.99015748]
Precision mean of Forest Classifier:
 0.97214
Recall of Forest Classifier model:
 [0.98580122 0.98782961 0.97302505 0.92618807 0.98311817 0.91959799
 0.98784195 0.9812253  0.99182004 0.98434442]
Recall mean of Forest Classifier:
 0.97208
F1_score of Forest Classifier model:
 [0.98380567 0.9898374  0.97537422 0.91645823 0.97633136 0.92987805
 0.98534613 0.98316832 0.99334357 0.98724239]
F1_score mean of Forest Classifier:
 0.97208
```

Figure 11: Random Forest classifier output, dataset 2

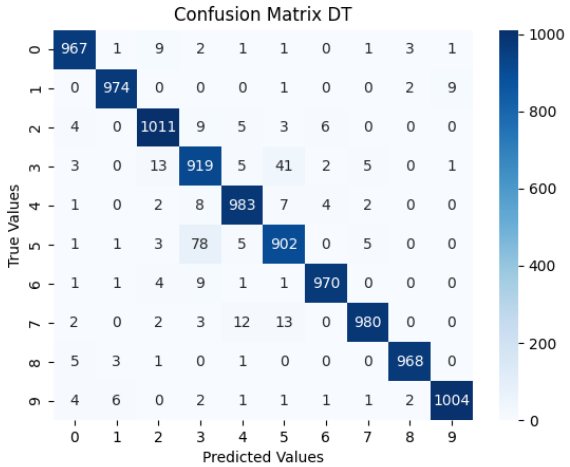

Figure 12: Random Forest classifier confusion matrix, dataset 2

3.3.2 Perceptron

For the other two models, I decided to use the Sklearn Bagging Classifier, denoted by this function:

```
sklearn.ensemble.BaggingClassifier(estimator=None, n_estimators=10, *, max_samples=1.0,
max_features=1.0,
bootstrap=True, bootstrap_features=False,
oob_score=False, warm_start=False,
n_jobs=None, random_state=None,
verbose=0, base_estimator='deprecated')
```

For this model, it was **feasible** to search for parameters using the Randomized Search function since the computation time was still manageable. However, the number of iterations was reduced to 3, providing a result that is not entirely precise but still **acceptable**. When combining all of this with the use of a classifier like Bagging, the obtained results, as visible in Figure 13, can be considered quite good and similar to the previous model.

```
Run Perceptron...

Fitting 5 folds for each of 3 candidates, totalling 15 fits
Best parameters found: {'penalty': 'elasticnet', 'max_iter': 100, 'eta0': 0.060000000000000005, 'alpha': 1e-05}
(40000, 1000)
Accuracy of perceptron model:
 0.9715
Precision of Perceptron model:
 [0.98284561 0.99284254 0.97490347 0.90757129 0.95568401 0.95063025
 0.98092369 0.98798799 0.99486125 0.98823529]
Precision mean of Perceptron model:
 0.97165
Recall of Perceptron model:
 [0.98782961 0.98478702 0.97302505 0.93326593 0.98510427 0.90954774
 0.98986829 0.97529644 0.98977505 0.98630137]
Recall mean of Perceptron model:
 0.97148
F1_score of Perceptron model:
 [0.98533131 0.98879837 0.97396336 0.92023928 0.97017115 0.92963534
 0.98537569 0.98160119 0.99231164 0.98726738]
F1_score mean of Perceptron model:
 0.97147
```

Figure 13: Perceptron model output, dataset 2

Also in this case I provide the confusion matrix (Figure 14).

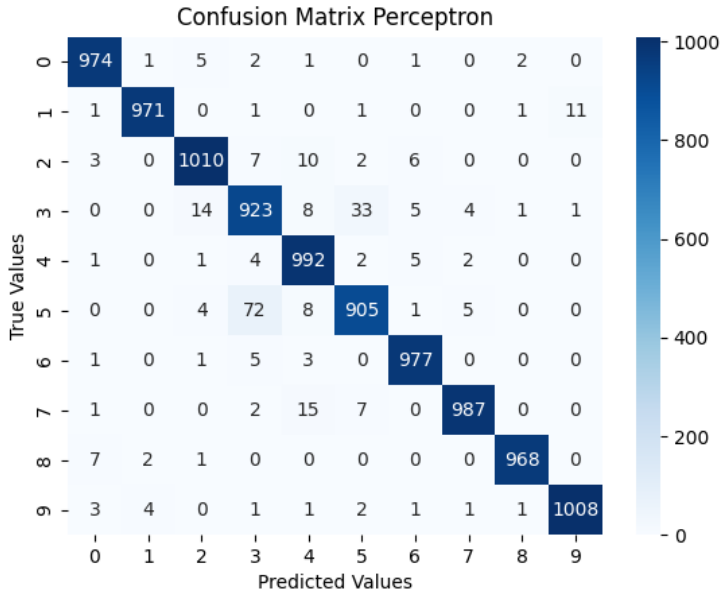

Figure 14: Perceptron confusion matrix, dataset 2

3.3.3 Kernelized SVM Model

Finally, for this model, I still attempted to find the best parameters, both through a Randomized Search across the entire dataset and on a small and shuffled subset of the dataset. However, due to the enormous input space, the computation time is **exceptionally long**. Therefore, in the end, I decided to **manually explore** the use of various parameters, starting from the best ones obtained for dataset 1. Fortunately, both due to the inherent power of the model itself and the use of bagging, the results obtained are **good**, regardless of the parameter choices. I show the results obtained in the Figure 15. As before, I also provide in the Figure 16 the **confusion matrix** for the best parameters I found. In conclusion, it can be affirmed that the obtained results are **satisfactory** and meaningful, especially considering the size of the input dataset.

```
Accuracy of SVM model:
 0.9729
Precision of SVM model (RBF, C = 1.6) with bagging:
 [0.97891566 0.99185336 0.97299904 0.91295747 0.97047244 0.95020747
 0.97895792 0.98807157 0.99487179 0.99014778]
Precision mean of SVM model (RBF, C = 1.6) with bagging:
 0.97295
Recall of SVM model (RBF, C = 1.6) with bagging:
 [0.98884381 0.98782961 0.97206166 0.93326593 0.97914598 0.92060302
 0.98986829 0.98221344 0.99182004 0.98336595]
Recall mean of SVM model (RBF, C = 1.6) with bagging:
 0.9729
F1_score of SVM model (RBF, C = 1.6) with bagging:
 [0.98385469 0.9898374  0.97253012 0.923      0.97478992 0.93517101
 0.98438287 0.9851338  0.99334357 0.98674521]
F1_score mean of SVM model (RBF, C = 1.6) with bagging:
 0.97288
```

(a) SVM, kernel=RBF, C=1.6

```
Accuracy of SVM model:
 0.9667
Precision of SVM model (poly, C = 1.6, degree=3) with bagging:
 [0.98674822 0.99384615 0.98617966 0.8183391  0.97502498 0.96017223
 0.98769231 0.9959596  0.99587203 0.99208704]
Precision mean of SVM model (poly, C = 1.6, degree=3) with bagging:
 0.96919
Recall of SVM model (poly, C = 1.6, degree=3) with bagging:
 [0.98174442 0.98275862 0.96242775 0.95652174 0.96921549 0.89648241
 0.97568389 0.9743083  0.98670757 0.981409  ]
Recall mean of SVM model (poly, C = 1.6, degree=3) with bagging:
 0.96673
F1_score of SVM model (poly, C = 1.6, degree=3) with bagging:
 [0.98423996 0.98827129 0.97415895 0.88205128 0.97211155 0.92723493
 0.98165138 0.98501499 0.99126862 0.98671913]
F1_score mean of SVM model (poly, C = 1.6, degree=3) with bagging:
 0.96727
```

(b) SVM, kernel=poly, C=1.6, degree=3

```
Accuracy of SVM model:
 0.9729
Precision of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 [0.98187311 0.99185336 0.97672163 0.91141732 0.96951819 0.94906445
 0.97893681 0.98415842 0.99486653 0.99115914]
Precision mean of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 0.97296
Recall of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 [0.98884381 0.98782961 0.97013487 0.93629929 0.97914598 0.91758794
 0.98885512 0.98221344 0.99079755 0.98727984]
Recall mean of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 0.9729
F1_score of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 [0.98534613 0.9898374  0.97341711 0.92369077 0.9743083  0.93306081
 0.98387097 0.98318497 0.99282787 0.98921569]
F1_score mean of SVM model (sigmoid, C = 1.6, degree=3) with bagging:
 0.97288
```

(c) SVM, kernel=Sigmoid, C=1.6

Figure 15: Output for Kernelized SVM model

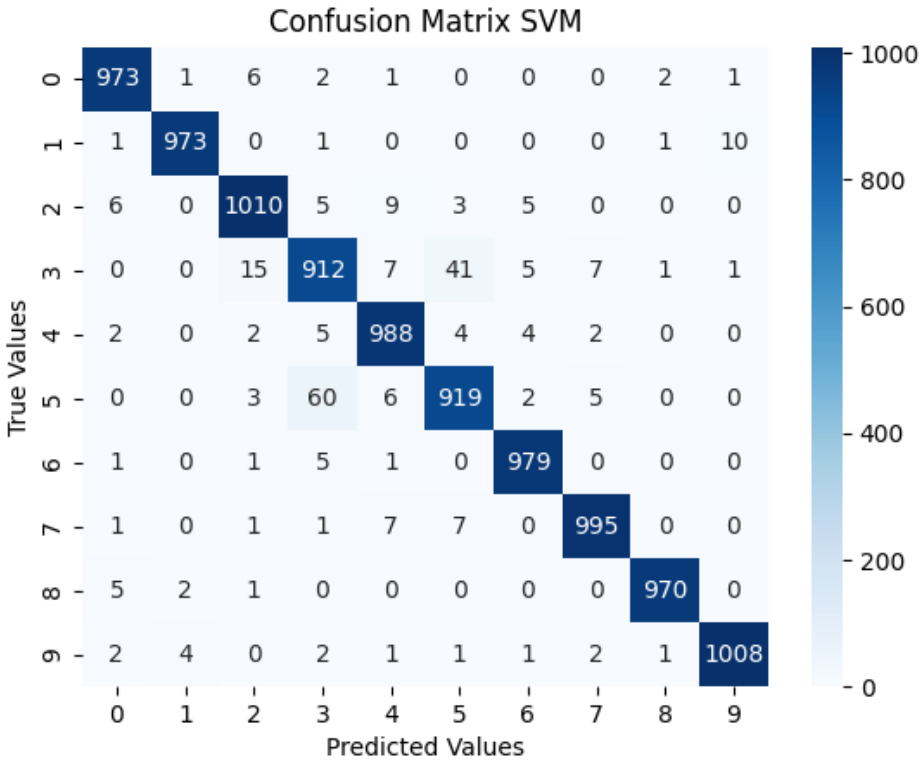

Figure 16: Kernelized SVM confusion matrix, dataset 2