Project Part 2 - RP

Giovanni Maffeo: 2024232210
Diogo Pinto: 2024156583

Source code: GitHub Repository

Contents

1 Introduction

In this project we will implement a URL phishing detector with machine learning algorithms. For that we will train and test using different techniques for feature selection and reduction with different parameters so that we can see what are the best options for better performance, to avoid overfitting and problems like the curse of dimensionality. Finally, we can analyze the performance using different kinds of classifiers.

2 Program Execution

2.1 Execution Diagram

As detailed previously in A, the program execution follows the same sequence as presented in the menu. In summary, the process consists of the following steps:

1. **Preprocessing:** Initial preparation of the data. At this stage, the dataset is split into two subsets: the development subset, which will be used for most of the training, hyperparameter tuning and validation procedures and the test subset, which comprises 30% of the data. The test subset will only be used in a final evaluation step (Section 7.4) to assess whether the classifiers generalize well to unseen data. Importantly, no feature selection or feature reduction techniques are applied to the test set, ensuring a reliable and unbiased performance evaluation.

2. **Feature Selection:** Execution (or not) of a feature selection algorithm, such as the Kruskal-Wallis Test (KS Test) or the ROC Curve analysis, using the number of features specified by the user.

3. **Feature Reduction:** Execution (or not) of a feature reduction algorithm based on the user's choice, such as PCA or LDA. For PCA, the user-defined dimensionality is also applied at this stage.

4. **Classification:** After feature selection and optional reduction have been applied to the initial dataframe, the classifier chosen by the user is applied. Available classifiers include Fisher LDA, Euclidean Minimum Distance Classifier, Mahalanobis Minimum Distance Classifier, K-Nearest Neighbors (KNN), Naive Bayes, and Support Vector Machine (SVM). It is important to note that some combinations are not appropriate, for instance, applying PCA before Fisher LDA is redundant, as Fisher LDA inherently performs its own dimensionality reduction.

A diagram illustrating this flow is shown in Figure 1.

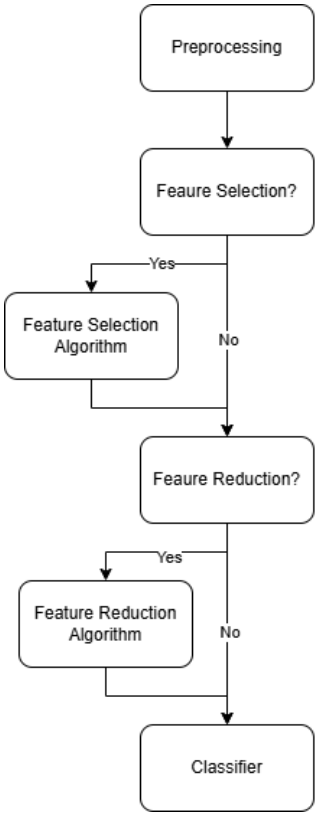


Figure 1: Program Execution Diagram

3 Pre Processing

For pre processing we first removed all the features that are non-numeric to simplify the project, we analyzed the dataset to see if there was missing values, which wasn't the case and finally we normalized the data with a Min Max Scaler.

4 Feature Selection

4.1 Feature Correlation

An important step during feature selection is to assess redundancy, since most feature selection methods are typically limited to evaluating feature relevance, as in the case of the Kruskal–Wallis test or feature discriminative, as in the case of ROC Curve.

As a result, two features may be ranked as highly relevant to the problem after applying a selection method, yet they can be strongly correlated. This redundancy, often referred to as "nearly identical features", is undesirable and such features should be discarded.

To address this, the `numpy.corrcoef` function was used to compute the correlation matrix, which was then visualized as a heatmap. Based on this heatmap, redundant features with high correlation were removed, following the strategies described in Sections 4.2 and 4.3.

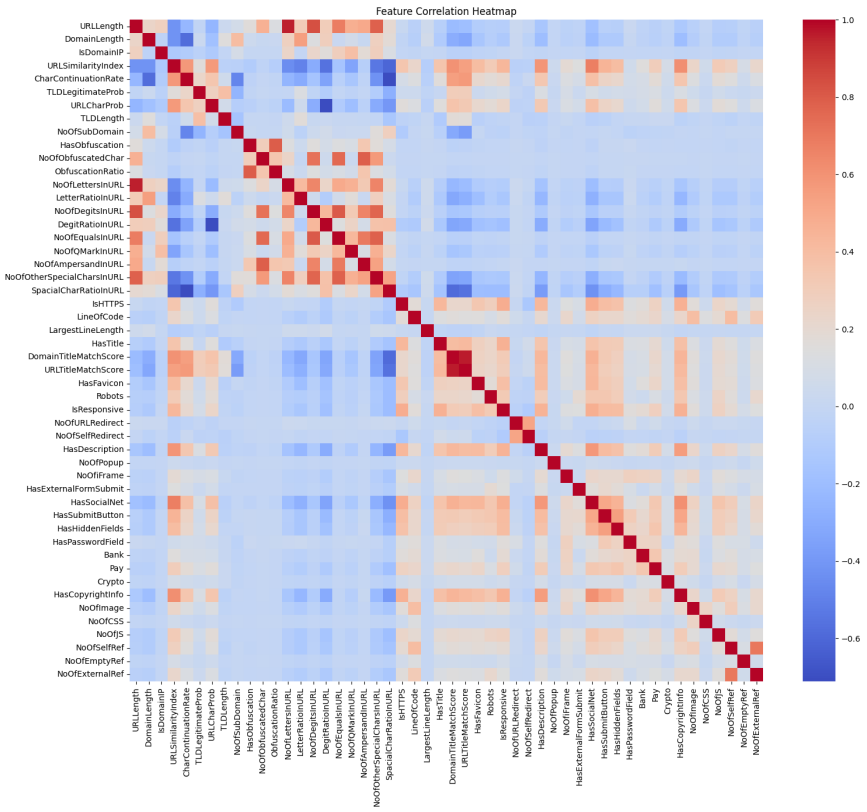


Figure 2: Feature Correlation Heatmap

4.2 Kruskal-Wallis Test

The Kruskal-Wallis Test (KS Test) was implemented as follows: the function receives dfData, which contains the dataset's features and dfLabels, which stores the target column for each sample. Additionally, the number of features to be selected is passed as a parameter, along with the option for the user to choose whether or not to remove correlated features. The execution of the KS test can be divided into two main parts.

In the first part, we create separate arrays for each class. In our case, we define a dataframe isReliable, where the target column is equal to 1, and another dataframe isNotReliable, where the target is 0. Then, we create a dictionary Hs to store the H values for each feature. We iterate over the features in dfData, compute the H value for each using scipy.stats.kruskal, and store these values in the Hs dictionary. As a result, we obtain the H statistic for each feature (those with the highest values are considered the most relevant for selection). The resulting ranking is presented below:

Feature	H	Feature	H
URLSimilarityIndex	210584.390	NoOfExternalRef	167404.031
NoOfImage	167374.106	LineOfCode	166758.998
NoOfSelfRef	162123.579	NoOfJS	159715.894
NoOfCSS	152179.270	HasSocialNet	145026.309
HasCopyrightInfo	130295.141	HasDescription	112336.958
NoOfOtherSpecialCharsInURL	88465.339	IsHTTPS	87489.400
HasSubmitButton	78927.976	DomainTitleMatchScore	76545.302
IsResponsive	70966.977	LargestLineLength	69335.656
NoOfDegitsInURL	68961.070	DegitRatioInURL	67811.984
SpacialCharRatioInURL	65590.234	URLTitleMatchScore	65028.727
NoOfiFrame	63975.058	NoOfEmptyRef	62589.425
HasHiddenFields	60785.627	HasFavicon	57474.839
CharContinuationRate	52089.106	HasTitle	49834.303
URLCharProb	49701.383	NoOfLettersInURL	40733.688
URLLength	37479.055	Robots	36347.742
LetterRatioInURL	32077.035	Pay	30515.890
DomainLength	10750.930	TLDLegitimateProb	9216.915
NoOfPopup	8777.780	Bank	8419.143
NoOfQMarkInURL	8406.971	NoOfEqualsInURL	7383.037
HasExternalFormSubmit	6621.303	HasPasswordField	4502.371
Crypto	2339.584	NoOfSelfRedirect	1378.592
NoOfAmpersandInURL	1189.386	IsDomainIP	854.598
TLDLength	698.279	HasObfuscation	649.232
NoOfObfuscatedChar	649.232	ObfuscationRatio	649.232
NoOfURLRedirect	508.884	NoOfSubDomain	263.171

Table 1: Kruskal-Wallis Test Ranking

In the second part, if the user has chosen to do so, instead of simply selecting the top features based solely on the number defined by the user, we first remove highly correlated features, which is a necessary step to ensure diversity among the selected features. This is because two features might have high H values while still being strongly correlated, thus providing redundant information. In such cases, keeping only one of them is preferable. To achieve this, we compute the correlation matrix of the features using numpy.corrcoef, and apply a threshold of 0.9. If two features have a correlation coefficient above this threshold, one of them is removed from the Hs dictionary. Finally, the dictionary is sorted by the H values, and the top features are selected according to the number specified by the user. The initial dfData is then reduced to include only these selected features and returned.

Initially, the correlation threshold was set to 0.8. However, after discussing with the professor that this might be too low, we increased the threshold to 0.9. As a result, two features were removed due to high correlation. In this case, the highly correlated features from which one was

removed are listed below:

1. `URLLength` (feature 0) and `NoOfLettersInURL` (feature 12): correlation of approximately 0.956.

2. `DomainTitleMatchScore` (feature 25) and `URLTitleMatchScore` (feature 26): correlation of approximately 0.961.

4.3 ROC Curve

Another strategy applied for feature selection was the use of Receiver Operating Characteristic (ROC) curves, a widely used tool for binary classification problems. The ROC curve plots the proportion of correct responses (True Positive Rate or Sensitivity) against the False Positive Rate (FPR) as the decision threshold varies.

The ideal ROC curve begins at the point (FPR = 0, Sensitivity = 0) and rises steeply to the point (FPR = 0, Sensitivity = 1), then extends horizontally to (FPR = 1, Sensitivity = 1). A perfect classifier reaches the point (FPR = 0, Sensitivity = 1) at a certain threshold, while a random classifier follows the diagonal line at 45 degrees. This can be visualized in the image below.

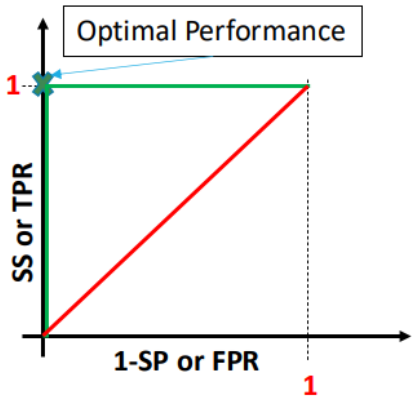


Figure 3: Theory ROC Curve

In this context, to assess classifier performance based on the ROC curve, we can use another concept called Area Under the Curve (AUC). The higher the AUC score, the closer the classifier is to being perfect and vice versa.

Although the ROC curve has other applications, it can also be used for feature ranking. The idea is to train a one-dimensional classifier for each feature and vary the threshold. The feature that yields the highest AUC score is considered the most discriminative and should be ranked higher.

Thus, the feature selection implemented in this section follows exactly the same structure as described in Section 4.2, replicating the two main parts. However, in the first part, instead of using H values, the ranking is obtained using the AUC score of each feature, computed with `sklearn.metrics.roc_auc_score`, which applies the strategy previously explained. The resulting ranking is presented below:

Feature	**AUC**	**Feature**	**AUC**
URLSimilarityIndex	0.996	LineOfCode	0.990
NoOfExternalRef	0.988	NoOfImage	0.980
NoOfSelfRef	0.972	NoOfJS	0.971
NoOfCSS	0.958	HasSocialNet	0.895
HasCopyrightInfo	0.875	HasDescription	0.846
LargestLineLength	0.816	DomainTitleMatchScore	0.792
HasSubmitButton	0.788	URLTitleMatchScore	0.771
IsResponsive	0.768	URLCharProb	0.768
NoOfiFrame	0.755	IsHTTPS	0.754
HasHiddenFields	0.749	NoOfEmptyRef	0.745
CharContinuationRate	0.744	HasFavicon	0.740
Robots	0.675	HasTitle	0.661
Pay	0.655	TLDLegitimateProb	0.609
Bank	0.564	NoOfPopup	0.545
HasPasswordField	0.542	HasExternalFormSubmit	0.535
Crypto	0.515	NoOfSubDomain	0.515
HasObfuscation	0.498	NoOfObfuscatedChar	0.498
ObfuscationRatio	0.498	IsDomainIP	0.497
NoOfAmpersandInURL	0.496	NoOfSelfRedirect	0.485
NoOfURLRedirect	0.484	TLDLength	0.474
NoOfEqualsInURL	0.473	NoOfQMarkInURL	0.470
DomainLength	0.376	LetterRatioInURL	0.285
DegitRatioInURL	0.274	NoOfDegitsInURL	0.272
URLLength	0.268	NoOfLettersInURL	0.258
SpacialCharRatioInURL	0.192	NoOfOtherSpecialCharsInURL	0.176

Table 2: ROC Curve Ranking

Another relevant analysis is the comparison of the ROC curves for the highest-ranked and lowest-ranked features, in order to theoretically validate the ranking. As shown in the figure, the feature `URLSimilarityIndex` achieves a high AUC score and its curve closely approaches the point of optimal performance, indicating strong discriminative power. On the other hand, the feature `NoOfOtherSpecialCharsInURL`, which was ranked last, exhibits a very low AUC score. Its curve lies below the diagonal that represents a random classifier (45-degree line), confirming that this feature alone results

in a performance worse than chance. This supports its classification as a feature with poor discriminative ability when considered individually.

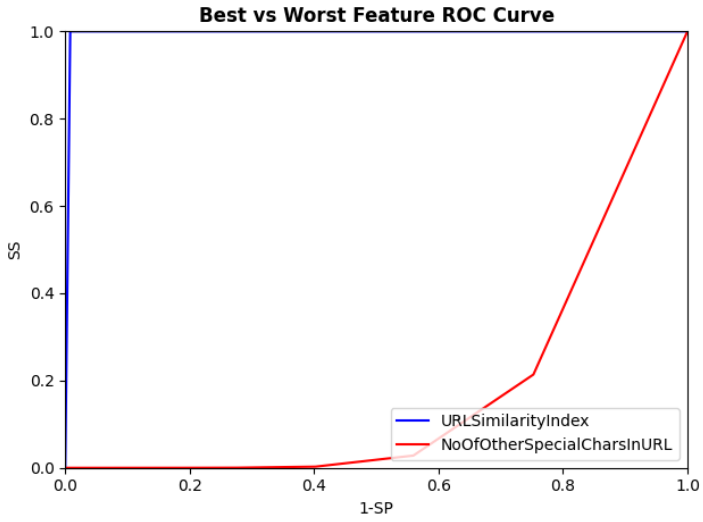


Figure 4: Best and Worst Feature ROC Curve

4.4 Feature Ranking Comparasion

A comparative analysis between the Kruskal–Wallis ranking and the ROC AUC ranking reveals that some features exhibit notable differences in their positions. This suggests that certain features may present high statistical relevance (high position in the Kruskal–Wallis ranking), yet low discriminative ability when used individually in classification tasks, as reflected by their low AUC scores. Table 3 highlights the features with the largest discrepancies between the two methods.

Feature	**Kruskal Rank**	**ROC Rank**	**Absolute Difference**
NoOfOtherSpecialCharsInURL	11	50	39
SpacialCharRatioInURL	19	49	30
NoOfDegitsInURL	17	46	29
DegitRatioInURL	18	45	27
NoOfLettersInURL	28	48	20

Table 3: Feature Ranking Comparasion

5 Feature Reduction

5.1 PCA

Principal Component Analysis (PCA) is an unsupervised dimensionality reduction technique that projects the data onto a lower-dimensional space by identifying the directions (principal components) that maximize variance. It aims to capture as much information as possible using fewer dimensions, without taking class labels into account.

In the implementation of PCA, we use the PCA module from sklearn. For the dimensionality selection based on eigenvalues, we implemented two different techniques:

- **Kaiser Criterion:** PCA selects the number of components equal to the number of eigenvalues greater than 1. This rule is based on the idea that, for standardized data, components with eigenvalues below 1 explain less variance than an original variable and are therefore considered not meaningful for dimensionality reduction.
- **Scree Test:** A threshold is calculated as 10% of the maximum difference between consecutive eigenvalues. When the difference between eigenvalues becomes smaller than this threshold, we assume the eigenvalues have stabilized. Therefore, only the components corresponding to differences greater than this threshold are considered for dimensionality reduction.

The method returns the reduced dataset and is shown below:

```
def featureReductionPCA(dfData, dfLabels, ...):
  pca = utils.PCA(n_components=dfData.shape[1])
  pca.fit(dfData)

  # components for reduction
  components = numberFeatures
  eigenvalues = pca.explained_variance_
  if (criterionPCAOption == 1):
    # kaiser criterion
    kaiserComponents = sum(eigenvalues > 1)
    components = kaiserComponents
  elif (criterionPCAOption == 2):
    # scree test: get difference between eigenvalues
    diff = utils.np.diff(eigenvalues) * -1
    threshold = 0.1 * max(diff) # set threshold for diff
    screeComponents = utils.np.argmax(diff < threshold) + 1
    components = screeComponents
  else:
```

```
    pca = utils.PCA(n_components=components)

dfPCA = utils.pd.DataFrame(pca.fit_transform(dfData))
return dfPCA
```

5.2 LDA

Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique that seeks to project the data onto a lower-dimensional space while maximizing class separability. Unlike Principal Component Analysis (PCA), which is an unsupervised method that identifies directions of maximum variance in the data without considering class labels, LDA takes label information into account. As a result, PCA emphasizes overall data spread, whereas LDA focuses on finding the directions that best separate the classes. Due to this fundamental difference, the two methods may produce different results when applied to the same dataset.

In this technique we reduce the number of features for the minimum value between the number of features and the number of classes - 1, since in this case we have a binary classification problem, we end up with a reduced dataset with only one dimension.

6 Classifiers

6.1 Fisher LDA Classifier

We use the LinearDiscriminantAnalysis module from sklearn to implement this classifier, which follows the same logic discussed in the theoretical class slides.

The method returns dfTargetTest and dfPredictions, containing the ground truth and the predicted classifications, respectively, which are then used to generate the evaluation output.

6.2 Eucludean Distance Classifier

For the Euclidean Minimum Distance Classifier (Euclidean MDC), we receive the following inputs: dfTrain, dfTest, dfTargetTrain, and dfTargetTest.

The first step is to compute the mean of the feature values for each class in the dataset. These class means are stored in an array. Then, we define a gFunction for each class, storing a function pointer for each in an array. It is important to note that each gFunction takes only one argument: x, the instance to be classified. This implementation follows the logic presented in

the lecture slides, where the value of g is defined by the expression shown in Equation:

$$g_k(\mathbf{x}) = \mathbf{m}_k^T \mathbf{x} - 0.5\, \mathbf{m}_k^T \mathbf{m}_k = \mathbf{m}_k^T \mathbf{x} - 0.5\, \mathbf{m}_k^2$$

.

```
# set g function for each class
gFunctions = []
for mean in means:
    def gFunction(x, mean=mean):
        return mean.T @ x - 0.5 * mean @ mean.T
    gFunctions.append(gFunction)
```

Since we are dealing with only two classes, it is possible to define a decision function between them, which also takes only x as input. Following the same logic from the slides, the decision function is defined by applying the gFunction of the first class (in our case, `isNotReliable`) and the gFunction of the second class (i.e., `isReliable`) to the instance x. The final classification follows the rule: if d(x)0 then classify as isNotReliable, else classify as isReliable. As shown in Figure **??**, we use the gFunction pointers for each class, applied to the instance x, to define the return value of the decision function.

```
# set d function for two first classes
def d(x):
    # if d >= 0 then isNotReliable else isReliable
    return gFunctions[0](x) - gFunctions[1](x)
```

Finally, we create the dfPredictions dataframe by applying the decision function d to each instance in dfTest, following the logic described above. We then return both dfTargetTest and dfPredictions to perform the evaluation.

6.3 Mahalanobis Distance Classifier

The Mahalanobis Minimum Distance Classifier (Mahalanobis MDC) follows almost the same logic as the classifier described in Section **??**, receiving exactly the same input parameters and producing the same outputs. The decision function is applied in the same way, as the difference between the gFunctions of each class. Therefore, its implementation is very similar, with the only difference being how the gFunction is calculated, as it must follow the equation below:

$$g_k(\mathbf{x}) = \mathbf{m}_k^T \mathbf{C}^{-1} \mathbf{x} - 0.5\, \mathbf{m}_k^T \mathbf{C}^{-1} \mathbf{m}_k$$

For this classifier, we first compute the covariance matrix for each class. Then, we calculate the inversed pooled covariance matrix, which acts as

an averaged covariance matrix that can be used for both gFunctions. It is important to note that, when dfTrain has only one feature (is a scalar), the numpy.inv function fails to compute the inverse because the matrix is not square. Therefore, a safeguard is implemented to handle this case.

Finally, the gFunctions are defined according to the theoretical lecture slides' equation for the Mahalanobis MDC and the rest of the logic remains the same as Euclidean MDC.

```
# get covariance matrix for each class
covarianceMatrices = []
for classLabel in sortedClasses:
    classSamples = dfTrain[dfTargetTrain == classLabel]
    covarianceMatrix = utils.np.cov(classSamples, rowvar=
        False)
    covarianceMatrices.append(covarianceMatrix)
# get inversed pooled covariance matrix
pooledCovariance = sum(covarianceMatrices) / len(
    sortedClasses)
if (dfTrain.shape[1] == 1): # dfTrain with just one feature
    inversedPooledCovariance = utils.np.array([[1 /
        pooledCovariance]])
else:
    inversedPooledCovariance = utils.inv(pooledCovariance)
# set g function for each class
gFunctions = []
for mean in means:
    def gFunction(x, mean=mean):
        return mean.T @ inversedPooledCovariance @ x - 0.5 *
            mean.T @ inversedPooledCovariance @ mean
    gFunctions.append(gFunction)
```

6.4 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is an algorithm that classifies a data point based on the majority label among its `k` nearest neighbors in the feature space. The distance metric, usually Euclidean, determines which neighbors are closest.

The classifier was implemented using the `KNeighborsClassifier` module from `sklearn`. The model is trained on the input data and used to predict test labels by evaluating the majority class among the nearest neighbors. The choice of the hyperparameter `k` is detailed in Subsection **??**.

6.5 Naive Bayes

Naive Bayes is a probabilistic classifier based on Bayes' Theorem, which assumes independence between features. In the Gaussian variant, it also

assumes that the continuous features follow a normal distribution.

The implementation uses the `GaussianNB` module from `sklearn`. The model was trained on the training set and evaluated on the test set without requiring any hyperparameter tuning.

6.6 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised learning algorithm that seeks to find the optimal hyperplane that maximally separates data points from different classes in the feature space. When data is not linearly separable, kernel functions are used to map the input space into a higher-dimensional space where separation becomes possible.

The classifier was implemented using the `SVC` module from `sklearn`, with an RBF kernel. The model is trained on the training subset and then used to predict the labels of the test set. Since SVM requires setting the hyperparameters `C` and `gamma`, their selection is discussed in Subsection **??**.

7 Tests

It was decided to measure the following metrics for the models: accuracy, precision, recall and f1 score. The f1 score was decided to be the one to appear in the tables for comparison because it is the metric that is more important for this project, since it gives us the idea of how balance the precision and recall. This is important because not only is necessary to make sure we minimize the number of false positives (higher precision) but also the most dangerous ones, the false negatives (higher recall).

7.1 Hyperparameters Grid Search

Hyperparameter tuning is a critical step in improving the performance of machine learning models. Some classifiers are particularly sensitive to variations in their hyperparameters, which can significantly affect their predictive ability. In this study, two classifiers: Support Vector Machine (SVM) and K-Nearest Neighbors (KNN), require the selection of hyperparameters. For SVM, parameters such as `C` and `gamma` influence the decision boundary's flexibility and kernel behavior. For KNN, the number of neighbors `k` determines the balance between local and global decision structures.

To address this, we performed a grid search, a method that systematically explores a predefined range of hyperparameter values to identify the combination that yields the best model performance. Specifically, we applied

a holdout evaluation (70% training, 30% testing) to the KNN classifier, running each configuration 30 times with different random seeds, varying `k` from 1 to 20. The resulting boxplot (Figure **??**) illustrates the variation in F1 scores across combinations, highlighting how model performance can drop considerably when increasing `k`.

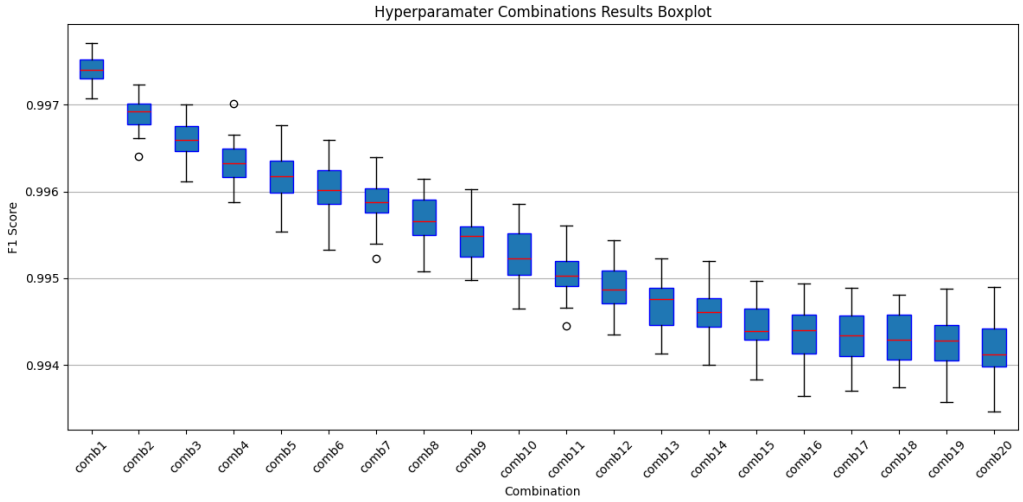


Figure 5: KNN Classifier Grid Search

To statistically validate the influence of hyperparameters on the KNN classifier, we performed a couple of tests. First, the Shapiro–Wilk test was applied to each combination to assess normality. All combinations showed $p > 0.05$, indicating that the data is normally distributed. Consequently, a one-way ANOVA test was used, confirming significant differences among combinations ($p < 0.0001$). To identify which combinations differ, pairwise independent t-tests with Bonferroni correction were applied. The results show that combinations with lower k values performed significantly better. Combination 1 consistently outperformed all others, confirming its statistical superiority. The most relevant results are summarized below.

k	p-value	Normal?
1	0.9483	Yes
2	0.1838	Yes
3	0.9172	Yes
4	0.2924	Yes
5	0.9840	Yes
6	0.9072	Yes
7	0.8956	Yes
8	0.6382	Yes
9	0.4708	Yes
10	0.8139	Yes
11	0.4289	Yes
12	0.6327	Yes
13	0.5908	Yes
14	0.9113	Yes
15	0.7315	Yes
16	0.6541	Yes
17	0.6347	Yes
18	0.2627	Yes
19	0.9044	Yes
20	0.4710	Yes

Table 4: Shapiro–Wilk test results

k	Wins
1	19
2	18
3	17
4	15
5	14
6	13
7	12
8	11
9	10
10	9
11	8
12	6
13	5
14	2
15–20	0

Table 5: Ranking of KNN hyperparameters

Following the same methodology, we conducted statistical tests to compare two SVM hyperparameter combinations. Due to the longer execution time of SVM models, we limited the analysis to two representative configurations, selected based on offline performance testing. The evaluated com-

binations were: (1) $C = 0.1$, $\gamma = 0.1$ and (2) $C = 0.1$, $\gamma = 1.0$. As with KNN, the Shapiro–Wilk test indicated that both combinations follow a normal distribution ($p > 0.05$). Consequently, a pairwise independent t-test was conducted, showing that Combination 1 significantly outperformed Combination 2 ($p < 0.0001$, $t = 54.47$). The summary of the tests is presented in the tables below.

Combination	**p-value**	**Normal?**
1	0.3804	Yes
2	0.3775	Yes

Table 6: Shapiro–Wilk test results

Combination	**Wins**
1 (C = 0.1, $\gamma = 0.1$)	1
2 (C = 0.1, $\gamma = 1$)	0

Table 7: Ranking of SVM hyperparameters

7.2 Critical Dimension

The curse of dimensionality describes the decline in model performance that can occur when too many features are used, often due to overfitting in high-dimensional spaces. While we have efficient tools to rank features, such as 4.2 and 4.3, they do not indicate the optimal number of features for a given classifier. To address this, we apply the concept of Critical Feature Dimension (CFD), which empirically estimates the ideal number of features by gradually adding features according to a ranking. In this case, based on the Kruskal–Wallis test.

In order to determine the optimal number of features we decided to analyze for each classifier the performance (in this case F1-Score), using different number of features ranging from 1 to the original number of features (50) and plotting the results. Therefore, we determined that using K-Nearest Neighbors with 12 features would achieve the highest performance of 0.999963 of f1 score.

It is important to highlight that, for this analysis, the best hyperparameters, previously determined in Subsection **??**, were used for classifiers that require them. Additionally, the results are based on a holdout evaluation approach, with 70% of the data used for training and 30% for testing.

The following table presents the highest achieved values for f1 score and the respective number of features for each classifier.

Classifier	Number of Features	F1 Score
KNN	12	0.99996
Naive Bayes	16	0.99993
SVM	50	0.99956
Mahalanobis MDC	50	0.99935
Fisher LDA	50	0.99894
Euclidean MDC	12	0.98672

Table 8: F1 scores obtained by CFD

In the following graphics we can analyze the evolution of the performance with different number of features for each of the classifiers.

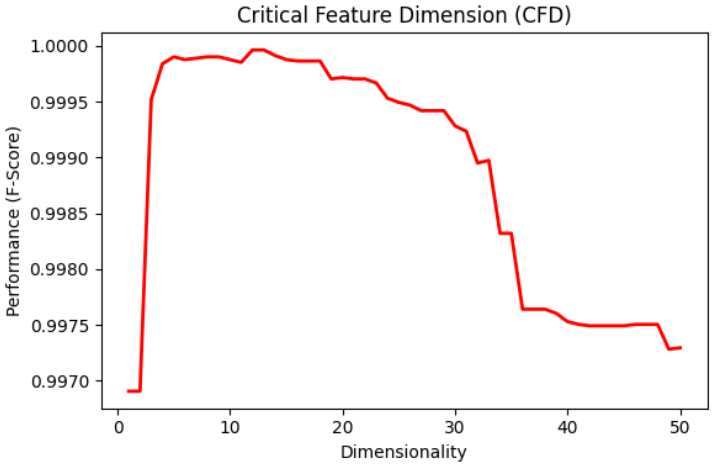


Figure 6: CFD KNN Classifier

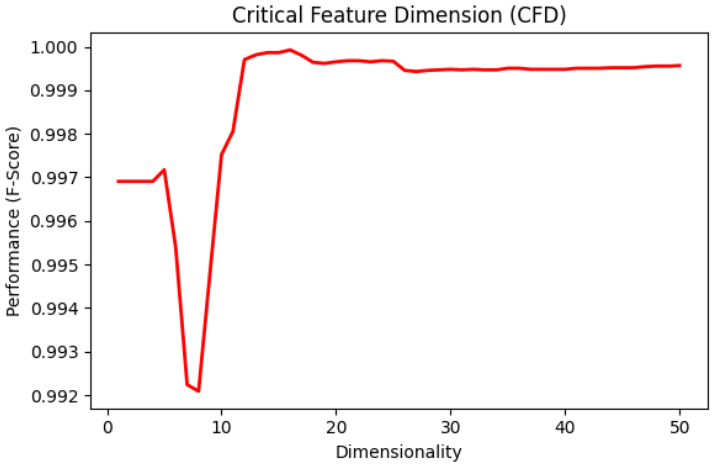


Figure 7: CFD Naive Bayes Classifier

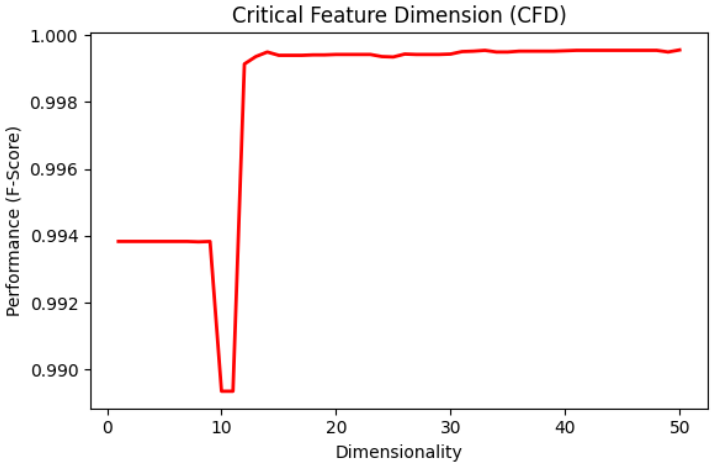


Figure 8: CFD SVM Classifier

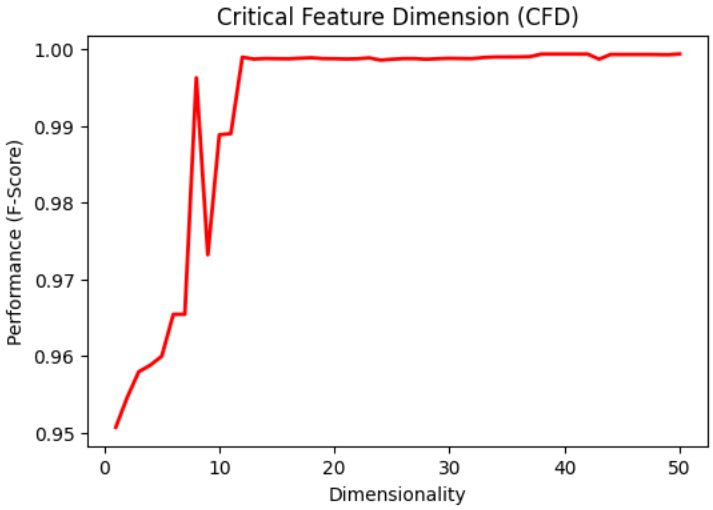


Figure 9: CFD Mahalanobis MDC

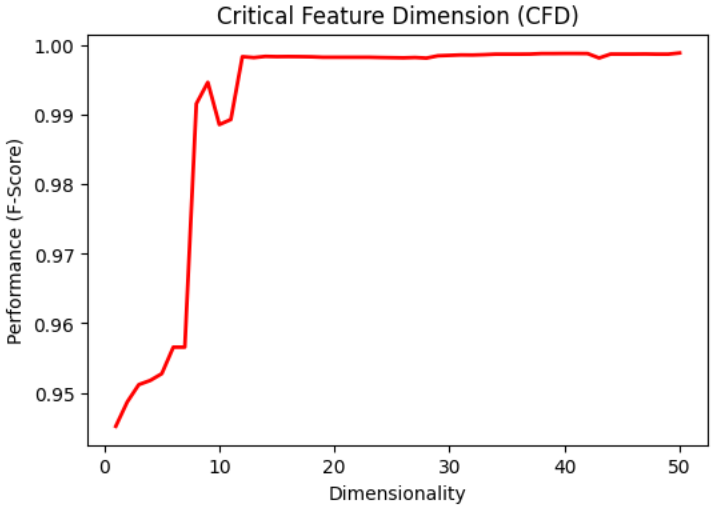


Figure 10: CFD Fisher LDA

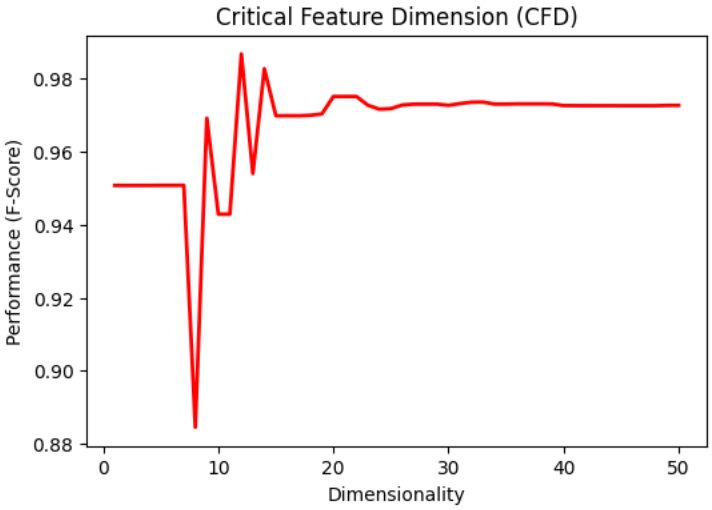


Figure 11: CFD Euclidean MDC

7.3 Classifiers Comparison

These tests were made 30 times with different seeds each time for the train test split, so that we would end up with different datasets for each training round. We also utilized for these set of tests the best classifiers' number of features (according to 7.2) for the selection phase (using the KS test) and we also measured the performance of the different classifiers using feature reduction as PCA.

It is important to mention that in PCA method, we reduced the dimension of the problem according to Kaiser Criterion. Another important observation is that **the only classifier to which PCA reduction was not applied was Fisher LDA**, since it already performs its own dimensionality reduction as discussed in Section 5.2. . The results of this comparative analysis are shown in Figure 12.

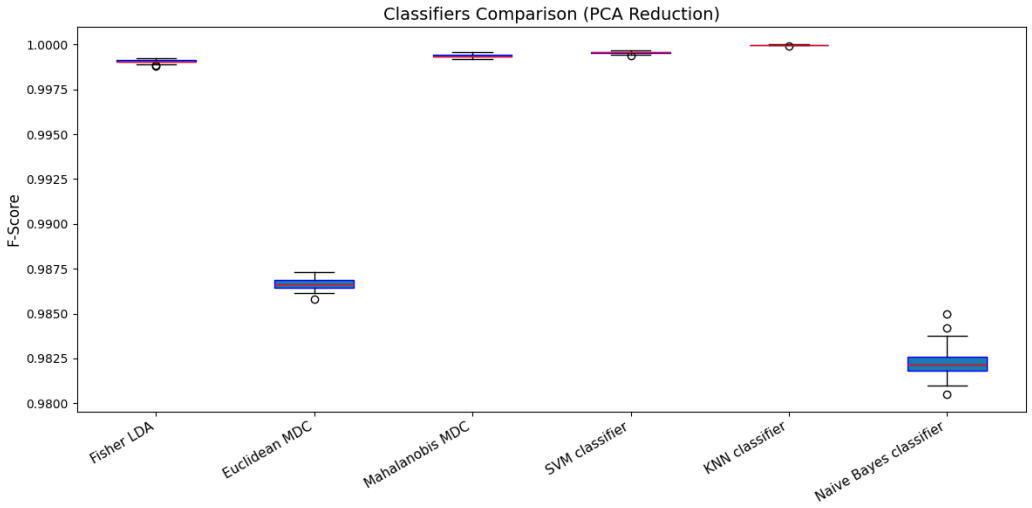


Figure 12: Classifiers Comparison

To assess whether the differences observed in Figure **??** are statistically significant, we applied a series of statistical tests. First, we performed the Shapiro–Wilk test to assess the normality of the F1-score distributions for each classifier. As shown in Table 9, at least one combination was found to be non-parametric ($p < 0.05$), namely the *KNN* and *Naive Bayes* classifiers. Consequently, we opted for the Kruskal–Wallis test to compare the classifiers. The result ($p = 0.0000$) indicates that there are statistically significant differences among them.

To identify which classifiers differ significantly from one another, we conducted pairwise Mann–Whitney U tests with Bonferroni correction. The results revealed that the classifier with PCA reduction that consistently outperformed all others was **KNN**, followed by **SVM** and **Mahalanobis MDC**, while **Naive Bayes** presented the weakest performance. The summary of test outcomes is shown below.

Classifier	**Shapiro–Wilk p-value**
PCA Reduction Euclidean MDC	0.8630
Fisher LDA	0.1983
PCA Reduction KNN classifier	**0.0059**
PCA Reduction Mahalanobis MDC	0.7152
PCA Reduction Naive Bayes classifier	**0.0111**
PCA Reduction SVM classifier	0.3804

Table 9: Shapiro–Wilk test results

Classifier	Pairwise Wins
PCA Reduction KNN classifier	5
PCA Reduction SVM classifier	4
PCA Reduction Mahalanobis MDC	3
Fisher LDA	2
PCA Reduction Euclidean MDC	1
PCA Reduction Naive Bayes classifier	0

Table 10: Ranking of Classifiers

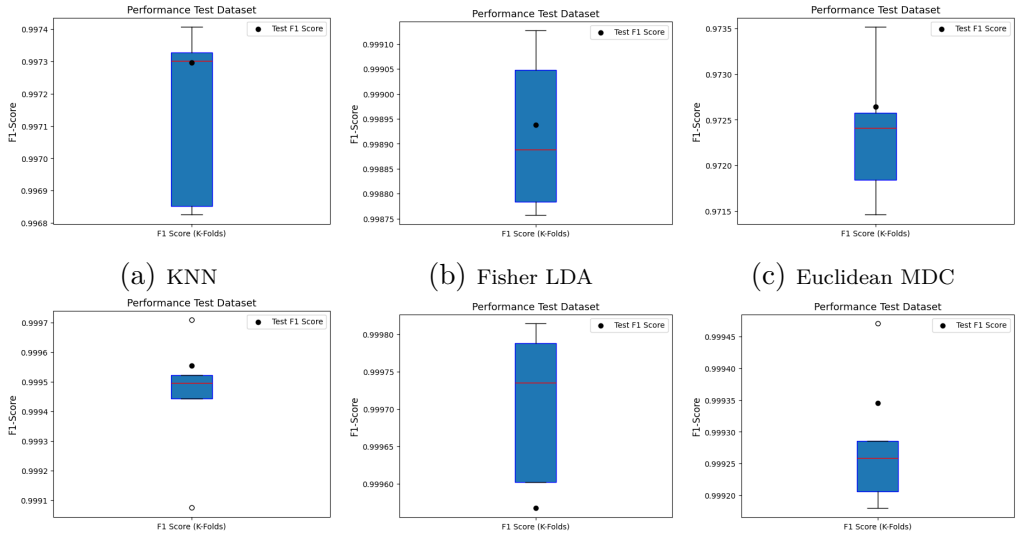
7.4 Performance Test Dataset

After selecting the best hyperparameter configuration for each classifier, determining the optimal number of features and comparing their performance on the development dataset, it is crucial to assess whether these results generalize to unseen data. Therefore, we evaluate the final performance of each classifier on the test dataset.

To ensure a robust evaluation, we apply k-fold cross-validation, a resampling technique that partitions the dataset into k equal subsets (folds). In each of the k iterations, one fold is used as the validation set while the remaining $k-1$ folds are used for training. This process helps mitigate the bias of a single train-test split and provides a more reliable estimate of model generalization. In our case, we used $k = 5$.

For each classifier, we compute the mean and standard deviation of the F1 Score across the 5 folds using k-fold cross-validation and we also evaluate the performance on the independent test dataset. This allows us to assess whether the classifiers generalize well to unseen data.

By visually comparing the results in Figure below, we observe that classifiers such as **KNN**, **SVM**, **Fisher LDA**, **Euclidean MDC**, and **Mahalanobis MDC** maintained their test performance within or even slightly above the interval defined by the mean $\pm$ standard deviation of the k-fold cross-validation results, confirming their good generalization capability. In contrast, **Naive Bayes** showed a lower performance on the test dataset, falling below the expected variability range, indicating that its generalization ability may be limited in this context.



(a) KNN (b) Fisher LDA (c) Euclidean MDC

(d) SVM (e) Naive Bayes (f) Mahalanobis MDC

Figure 13: Performance Test Dataset

8 Conclusion

In conclusion, this study systematically explored how hyperparameter tuning, feature selection and dimensionality reduction affect classifier performance. By evaluating multiple classification algorithms under controlled and comparable conditions, we were able to identify the most effective configurations for each model.

In Section 7.1, we investigated the impact of hyperparameters on KNN and SVM classifiers using grid search. The results showed that both classifiers are highly sensitive to these parameters, with statistical tests confirming significant performance differences across configurations. For KNN, lower values of k led to superior performance, while for SVM, a smaller γ provided better generalization.

Section 7.2 introduced the concept of Critical Feature Dimension (CFD) to empirically determine the optimal number of features for each classifier. This analysis highlighted that the ideal number of features varies across models, with KNN achieving its best F1 score with 12 features, while models like SVM and Mahalanobis MDC performed better with the full feature set. This reinforces the importance of tailoring feature selection to each classifier.

Following this, Section 7.3 compared all classifiers using their optimal number of features and dimensionality reduction via PCA. Statistical analysis with Kruskal–Wallis and pairwise Mann–Whitney U tests (with Bonfer-

roni correction) confirmed significant differences between classifiers. Among them, KNN stood out as the best-performing classifier, closely followed by SVM and Mahalanobis MDC. Naive Bayes presented the weakest performance in this scenario.

Lastly, Section 7.2 assessed the generalization capabilities of each model by evaluating their performance on an independent test dataset. Most classifiers, including KNN, SVM, Fisher LDA, Euclidean MDC, and Mahalanobis MDC, maintained performance within the range observed in cross-validation, demonstrating robust generalization. However, Naive Bayes failed to replicate its development performance on the test data, indicating possible overfitting or sensitivity to data variability.

In summary, for our dataset and evaluation, the KNN classifier with 12 features and PCA reduction achieved the most consistent and statistically significant performance. This demonstrates the importance of combining careful hyperparameter tuning, feature selection, and appropriate dimensionality reduction to develop reliable classification models.

9 Acknowledgments

Chatgpt was utilized to help to generate code and also correct the english in the report.

A Options Menu

The menu presented below was implemented for the first project milestone and reflects the classifier options and functionality available at that stage. Although it was not updated to include the newly added classifiers introduced in later sections, the overall execution flow remains the same.

To execute the program manually, we created an options menu that appears when running the main script. This menu guides the entire execution of our implementation.

The first prompt asks the user to choose whether the execution will be manual or if it is intended for the generation of the Critical Feature Dimension (CFD), detailed in Section 7.2, or for the Classifiers Comparison, detailed in Section 7.3.

Next, the user is asked whether to apply any feature selection algorithm. The available options are the Kruskal-Wallis Test (KS Test) or no selection at all—in which case, the entire dataframe will be used. If the KS Test is selected, the user is also prompted to specify the number of features to retain.

After that, the user is asked whether they want to apply any type of feature reduction. The available options are: PCA, LDA, or none (i.e., no dimensionality reduction). If PCA is selected, the user is then asked if they would like to apply a criterion to determine the number of features to retain—either the Kaiser Criterion or the Scree Test. The user also has the option to manually specify the number of features.

Finally, the user is asked to choose which type of classifier to use in the execution. The available options are: Fisher LDA, Euclidean Minimum Distance Classifier, and Mahalanobis Minimum Distance Classifier. An example of the menu execution used to run the program is shown in Figure 14.

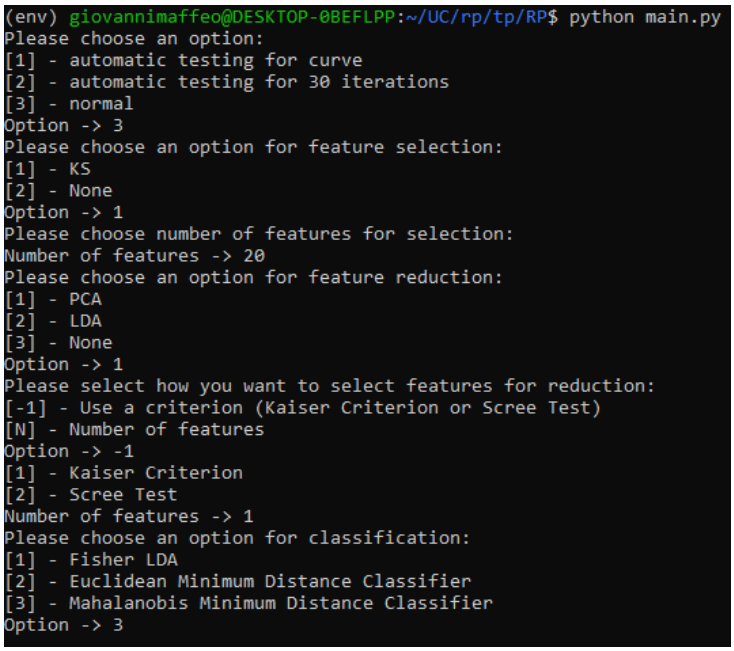


Figure 14: Options Menu Example

At the end of the program execution, a .csv file is generated containing the execution results. This file stores the following information related to the program run: featureSelection, numberFeaturesSelection, featureReduction, numberFeaturesReduction, classifier, TP, TN, FP, FN, accuracy, precision, recall, and fScore.