Part 1 - Data Exploration

In this part, we first need to distinguish between continuous and categorical. To do this, a number of ways exist, like using the **type of value** (in our case all were float), **visualization** and **statistics analysis** (mean, variance). In our case, we used the unique values and the range (minimum and maximum) of the values. The steps followed were:

1. Count unique values for each feature (column)
2. Find Minimum and Maximum Values for each feature (column)
3. Check if the range (max - min) is close to the count of unique values. We will use a threshold of 1 difference (heuristic) between unique and max-min. *We could also check for differences between the values for example, like if there is a certain step in the values (1,2,3,4,...) or if it is random (1,6,13,25,...). But since this serves as a starting analysis, we can use the method arbitrarily and then check the plots and values as well.*
4. Classify as Categorical or Continuous
   i) If the range is approximately equal to the count of unique values (difference is +/- 1) or if we only have 2 sets of unique values, we assign them as categorical.
   ii) If the range does not meet the aforementioned requirements, the feature is likely continuous.

While doing these steps, we also calculate mean, variance (although they don't mean anything for categorical) and store them along with unique values and the maximum, minimum values in a dictionary, indicating the type of data. An example format is the following:

{0: {'Type': 'Categorical',
'Unique_Val': 10,
'Max-Min': (10.0, 1.0),
'Mean': 5.452933151432469,
'Variance': 8.058607813175652},
3: {'Type': 'Categorical',
'Unique_Val': 16,
'Max-Min': (16.0, 1.0),
'Mean': 4.2792178262846745,
'Variance': 11.253551730177902},
4: {'Type': 'Continuous',
'Unique_Val': 6,
'Max-Min': (10.0, 1.0),
'Mean': 4.697135061391542,
'Variance': 8.868709391035363},}

First we have the feature as a key, and then as values we include different metrics which can be later viewed to decide for a feature.

Since we will be using classification algorithms later on, such as SVM and KNN, we would like features that are informative, which means we prefer features with high variance, which we could later scale or normalize if necessary. For this purpose, we calculate the variance and sort it in descending order. Another interesting metric we could check is correlation with the variable ***y***. After calculating the correlation, we sort it based on the variance order, so the first correlation shown will be the feature with the highest variance and so on. From these, features with highest correlation as well as relatively high variance were chosen. Four of those seemed interesting, which are features 0, 4, 10 and 12, having high positive correlation (when compared to the rest) as well as high variance. Different features were selected as the 5th, but after looking at plots and statistics, feature 5 was chosen as the fifth, since it had the highest negative correlation with our target variable, just so that there will be some variability in our features. Out of these, 3 are categorical and 2 are continuous.
In Table 1 we can see each feature, along with its type, unique values, distribution, and different statistics. This table can be viewed in the delivered files: ***Feature_table.csv***

Feature	Type	Unique	Class Distribution	Mean	Variance	Correlation
0	Categorical	10	{1.0: 214, 2.0: 229, 3.0: 223, 4.0: 245, 5.0: 186, 6.0: 213, 7.0: 249, 8.0: 235, 9.0: 211, 10.0: 194}			0,267
4	Continuous	6		4,697	8,868	0,299
5	Categorical	7	{1.0: 1034, 2.0: 301, 3.0: 703, 4.0: 64, 5.0: 67, 6.0: 29, 7.0: 1}			-0,407
10	Categorical	2	{1.0: 2014, 10.0: 185}			0,272
12	Continuous	7		4,486	8,481	0,236

Table 1. Selected features 0,4,5,10,12, along with different metrics and information. Class distribution indicates the unique value along with the number of counts belonging to that value

To assess the features, we can also check the histograms of distributions (basically how many values correspond to certain unique values). After looking closer at the values, the plots were personalized, like showing the unique values through ***xticks()***, which would reduce readability in case of numerous continuous values but in our case brings the plots to a similar form. At first, if we classified feature 10 based on the range of max-min, we would consider it continuous. This is the reason we also need to check the histograms to account for these events. Overall, it seems the way we classified the features as continuous or categorical offers some sort of reasoning. While all of the features could be seen as categorical with some missing cases, we consider here that if there is a gap between unique values, the feature has a nature of continuous data (like we notice for Feature 4 in Figure 1).

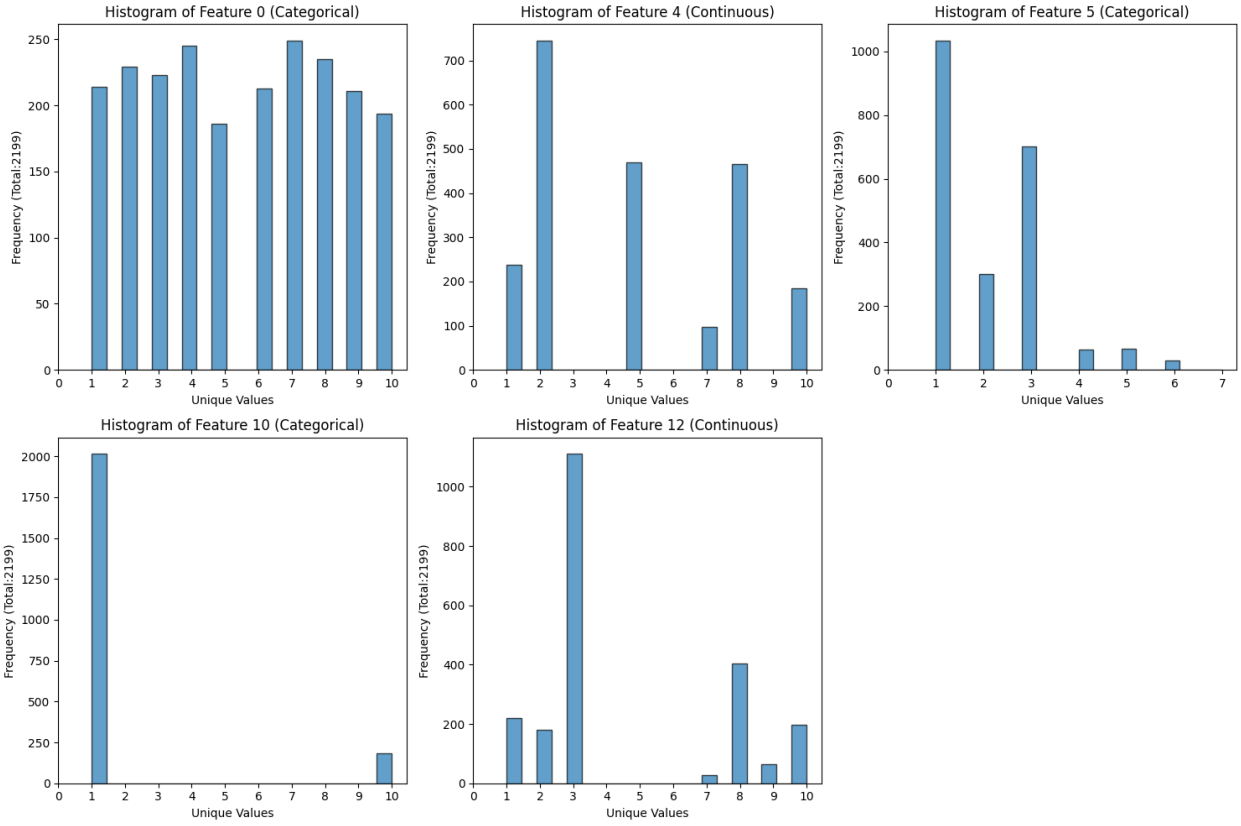

Figure 1. Histograms of the selected features. On x-axis we see the unique values and on the y-axis we see the amount of values that fall into each one (distribution).

Both Table 1 and Figure 1 were produced in a single loop, each time taking a feature and calculating different statistics, inserting them into our table and producing the plots.

Part 2 - Model Selection

For this part, 3 classifiers were chosen along with different hyperparameters from scikit-learn:

1. **SVM (Support Vector Machine) – LinearSVC**
   Hyperparameters
   i) L1: L1 norm, makes some weights zero, encourages sparsity of coefficients
   ii) L2: L2 norm, penalizes sum of the square of coefficients
   iii) C (Regularization Parameter): Strength of regularization (1/C). ***C = [0.1, 1, 10].*** In accordance to previous exercises, these C values seemed to be the best, so they were chosen again. Too high values will remove features as seen.
   iv) Extra parameter from scikit-learn is ***dual*** which determines the formulation used in the optimization problem. We set True for L2 and False for L1
2. **KNN (K-Nearest Neighbors) – KNeighborsClassifier**
   Hyperparameters
   i) K (n-neighbors): How many neighbors the algorithm considers for classification. ***k = [3, 5, 7]***
   ii) Uniform weights: All points in each neighborhood are weighted the same
   iii) Weighted Average (distance): Weights by the inverse of their distance. Closer neighbors have greater influence.
3. **RF (Random Forest) – RandomForestClassifier**
   Hyperparameters
   i)Number of Trees (n_estimators): Number of trees in the forest. ***N = [50, 100, 150]***. Higher values increase robustness, but also complexity.
   ii) Criterion gini: Uses gini impurity, which measures how often a randomly chosen elements would be incorrectly classified
   iii) Criterion entropy: Uses information gain (entropy), which is the information provided by a split.

The different combinations of these hyperparameters, gave us a total of 18 different configurations, which can be seen in the delivered files, along with the later calculated ROC AUC score, for K-Folds 5 and 10:
k_5_all_feat_configuration_table.csv (K-Folds = 5 for cross validation)
K_10_all_feat_configuration_table.csv (K-Folds = 10 for cross validation)

For the function ***create_folds(data, k)*** we used the ***StratifiedKFold()*** function from sklearn.model_selection, with *shuffle = True,* in order to shuffle while keeping the distribution of classes the same. This will return a list with a length equal to the number of folds. Each fold contains equally split indices from the samples. In our case, we had 2199 samples, so for a k_folds = 5, we will have a list with 5 lists, each containing 440 samples indices, and the last one 439.

The function ***run_model(train_data, test_data, configuration)*** uses as input a configuration in the form of a dictionary, a train and test data and depending on the value of the key ‘Type’ within our configuration dictionary, it performs SVM, KNN or RF accordingly, using the specified. First it trains the model using the train data, then predicts based on the test data. It returns both the model (not required for the exercise)

and the performance (model, performance). Performance is calculated using the ***roc_auc_score*** function from sklearn.metrics, after the prediction is done, using the original classes along with the predicted ones.

The next function, ***performance_estimation(data, validation_indices, configurations_list)***, will use the list of all configurations (18) and for each, it will calculate the performance of each fold. After it has calculated that, it will take the average of those performances, and store them as the overall configuration performance. This means, our output will be a list of 18 values, each corresponding to a configuration.

The last function ***best_model(performance_list, configurations_list)*** will simply find the maximum value (indicating the highest ROC AUC score) within the aforementioned list and through it find the corresponding configuration, which will show the parameters and classifier used to obtain that score.

The above steps were performed for K-folds k=5 and k=10, since we do not need more, due to the limited nature of our data. More folds means less samples in each fold, which could potentially reduce robustness.

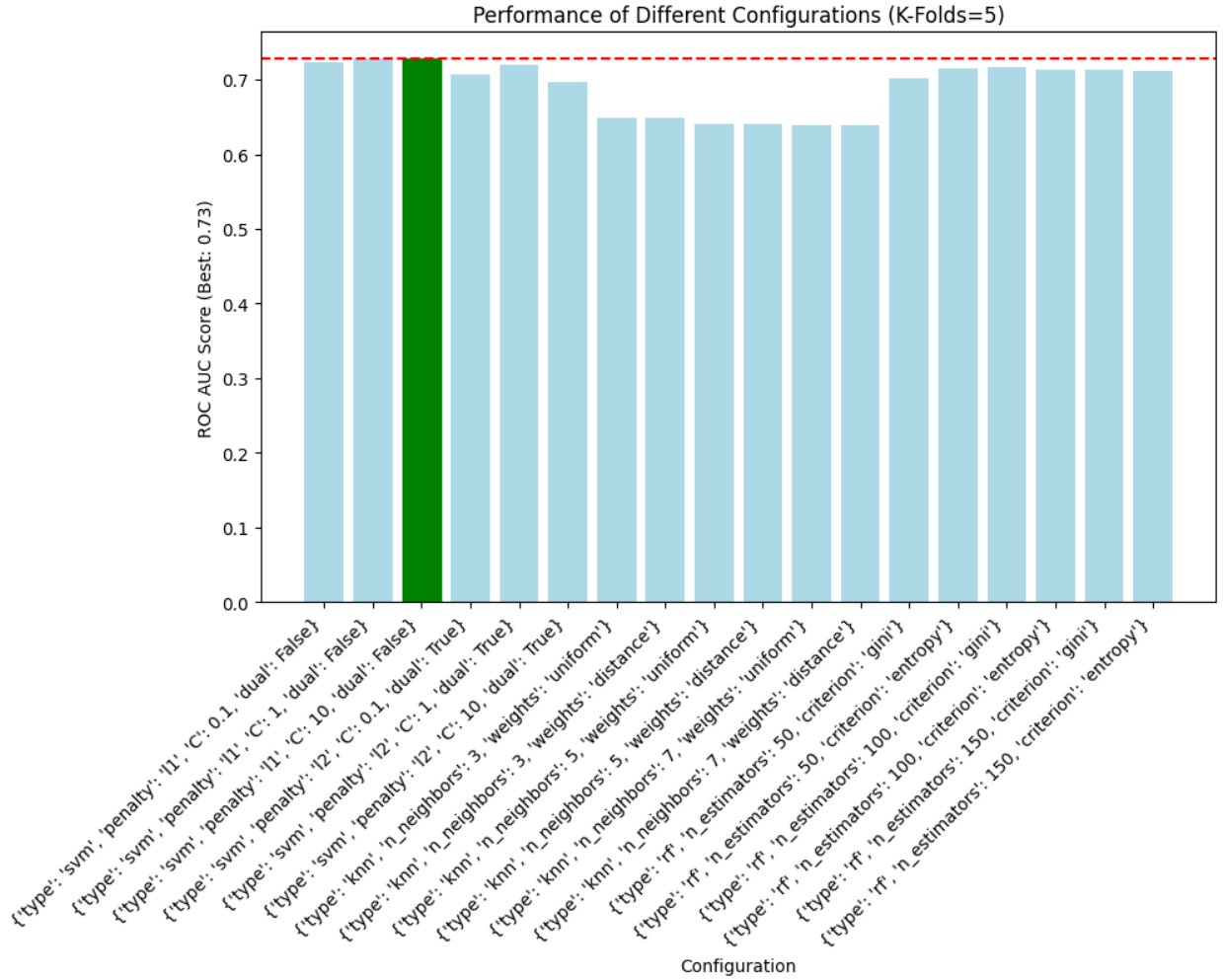

Figure 2. Bar plot indicating the ROC AUC score on the y-axis, and the configurations (parameters) used for the model on the x-axis. K-Folds = 5. Green highlights the best score.

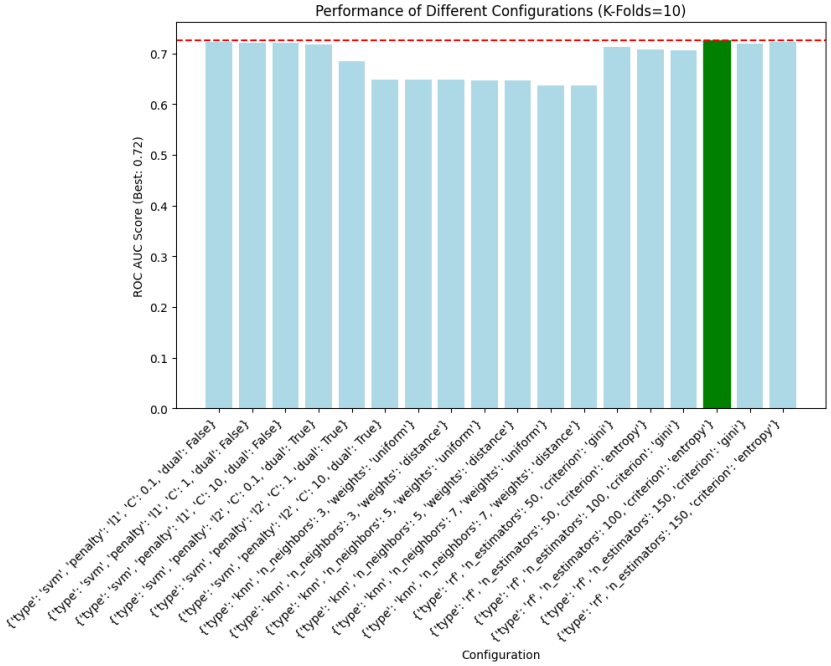

Figure 3. Bar plot indicating the ROC AUC score on the y-axis, and the configurations (parameters) used for the model on the x-axis. K-Folds = 10. Green highlights the best score.

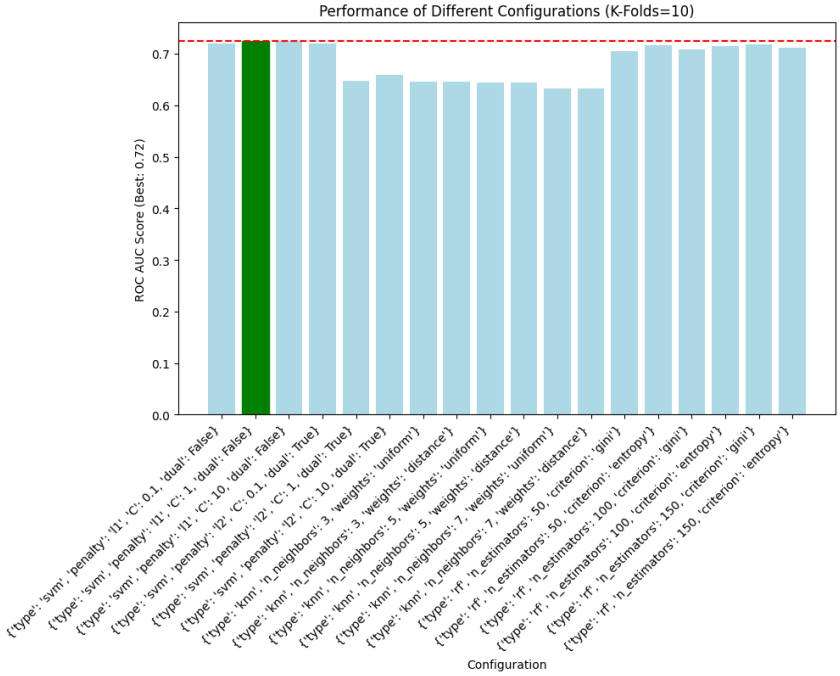

Figure 4. Bar plot indicating the ROC AUC score on the y-axis, and the configurations (parameters) used for the model on the x-axis. K-Folds = 10, different iteration. Green highlights the best score

We can see all the ROC AUC scores for each configuration in Figures 2-4. Figure 2 is performed with K-Folds = 5. We notice that the configuration with the best score is SVM with L1 and C=10. As for Figures 3,4, they indicate different iterations with K-Folds = 10, since the scores were similar. The alternation was between Random Forest with entropy and SVM with L1 and C=1. More information can be seen in the tables aforementioned. Overall, KNN seems to perform the worst and the number of folds does slightly increase the Random Forest ROC AUC scores, but the hyperparameters do not seem to affect our scores by a lot. Perhaps having more extreme values could provide us with a more clear indication. While the parameters chosen have high differences between them (regarding each model), perhaps the nature of the data is such that they do not affect performance and on a large scale. For the purpose of exploration, the same was performed for K-Folds = 5, but only using the 5 selected features, which interestingly gives the advantage to KNN as seen in Figure 5. While this was only done for exploration, we can see that feature selection might play a big role in how we also choose our model. The final model was trained using SVM with L1 and C=10, based on the results of K-Folds = 5.

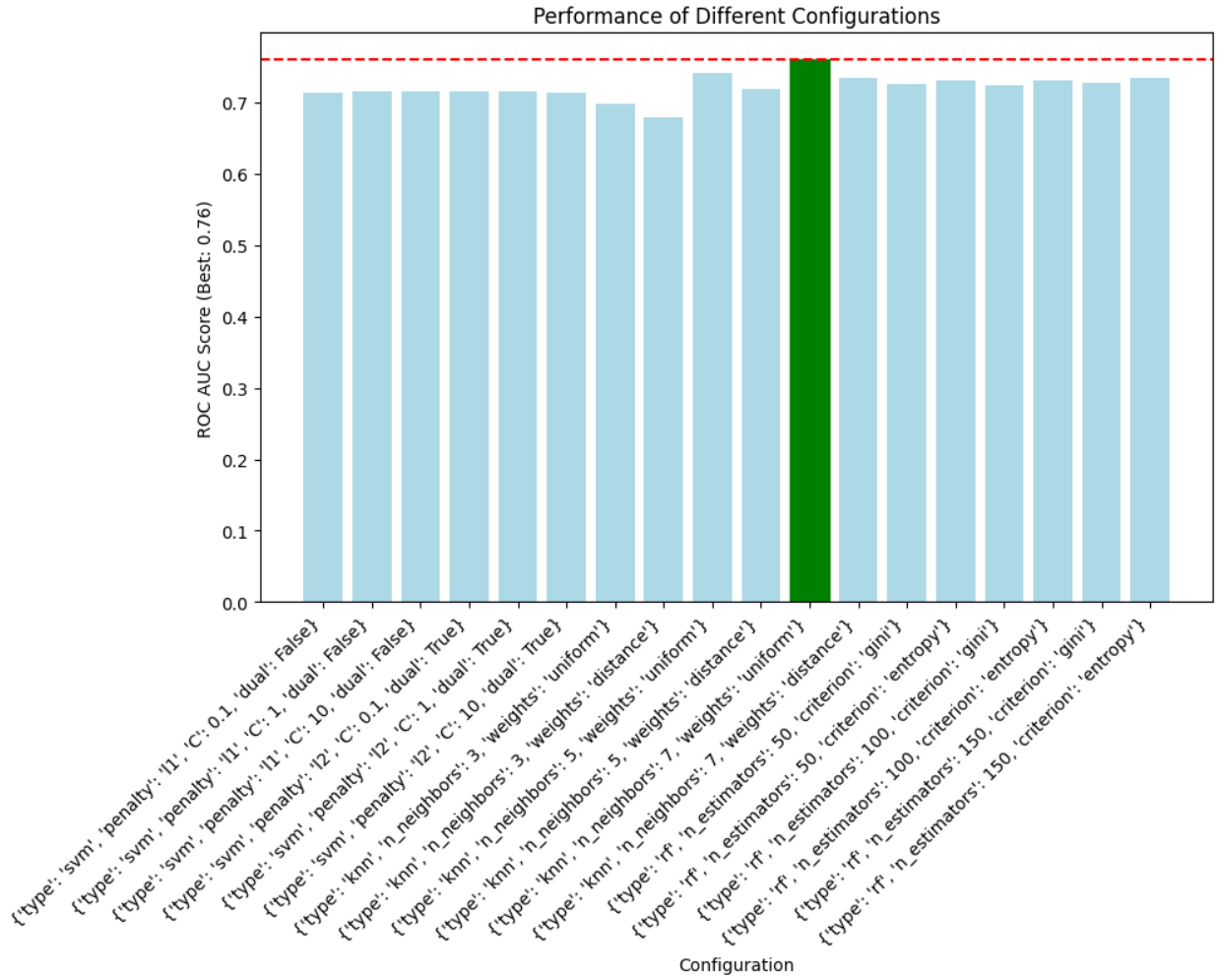

Figure 5. Bar plot on the selected features from part 1, indicating the ROC AUC score on the y-axis, and the configurations (parameters) used for the model on the x-axis. K-Folds = 5, different iteration. Green highlights the best score.

Part 3 - Computing the out-of-sample performance, ROC curve

Now that we have our trained model on all of our original data, we can use the other dataset as a test data to predict. The out-of-sample ROC AUC score is calculated in the same manner as before, using the ***roc_auc_score*** function. We notice that the score is slightly larger (~0.73 vs ~0.77). We do not notice any major differences, but the slightly higher score could indicate that our model generalizes well to new data and/or that the data used is a representative test set. But the score does not have that much of a difference to exclude the fact that it could just be simple variability in performance.

In order to produce the ROC curve, we need to calculate FPR (False Positive Rate), TPR (True Positive Rate) and the probabilities. The former are computed through the ***roc_curve*** function. Since we need probability scores, and LinearSVC does not provide them, we use ***CalibratedClassifierCV*** and re-fit our original dataset, then we can extract probabilities using the second test dataset. Now we have everything to plot (Figure 6).

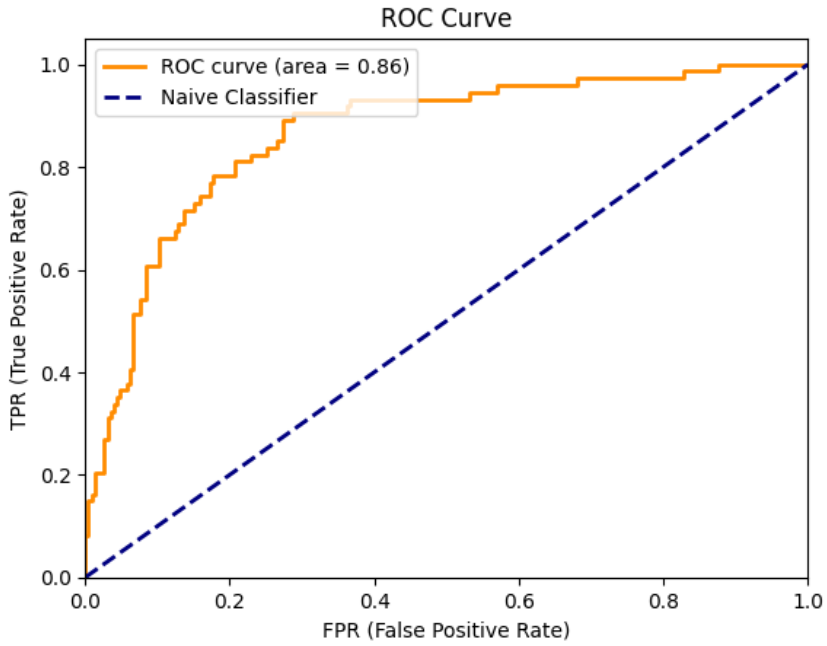

Figure 6. ROC curve compared to a Naive Classifier (trivial classifier).

We notice an area of 0.86 which is a good indicator that our model performs well, meaning it can distinguish well between our classes, and we compare it to the trivial classifier which assigns classes randomly (area = 0.5).

Part 4 -

For the last part of the assignment, we will bootstrap the holdout dataset for a number of 1000 iterations, and predict using our final model, which is the SVC used previously. For each iteration, we make a prediction on the bootstrapped data using the final model, compute the ROC AUC score and append it in a list. This list will eventually contain the distribution of performances for said dataset for the specific model. This interval gives us an estimate of the range within which the true performance of the model lies, with 95% certainty. If the confidence interval is narrow, it suggests that the model's performance is consistent and not highly sensitive to changes in the data sample. A wider confidence interval indicates more variability in performance, suggesting that the model's performance might depend significantly on the specific sample of data it's tested on. By comparing the original ROC AUC score (obtained on the entire holdout dataset without bootstrapping) with the confidence interval, we can assess how representative this original score is, which as we see (Figure 7), is in the middle of our confidence interval, where most of the values are. This means our original performance was expected when compared to the distribution of these performances on the bootstrapped dataset.

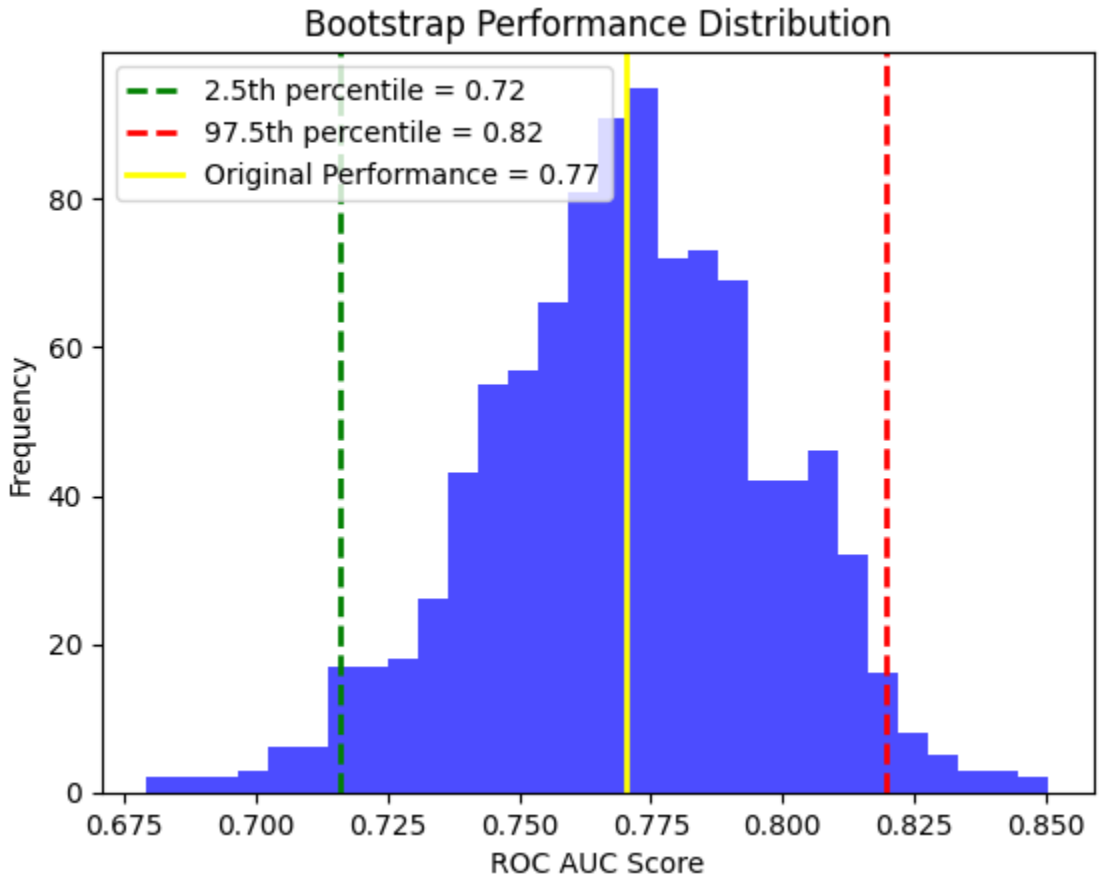

Figure 7. Performance distribution (ROC AUC scores) for the holdout dataset, using the best model (SVC, L1, C=10).