# Machine Learning - Milestone 2 - Supervised Learning

Raúl Barba Rojas[1,1*], Diego Guerrero Del Pozo[1,1†] and Marvin Schmidt[1,1†]

[1*]Department of Computer Science, Universidad de Castilla-La Mancha, Paseo de la Universidad, 4., Ciudad Real, 13071, Castilla-La Mancha, Spain.

*Corresponding author(s). E-mail(s): Raul.Barba@alu.uclm.es;
Contributing authors: Diego.Guerrero@alu.uclm.es;
Marvin.Schmidt@alu.uclm.es;
†These authors contributed equally to this work.

## 1 Introduction

The task to be completed within this assignment is the prediction of damages dealt to buildings in Nepal during the 2015 Gorkha earthquake. To do this, we use an evolutionary approach for supervised learning techniques, in which a baseline with basic techniques is established first and improved using more advanced optimization techniques.

The assignment is based on the competition "**Richter's Predictor: Modelling Earthquake Damage**" held by DrivenData.org. The dataset we are being provided consists of 39 features, from which the feature `buildingid` uniquely identifies a specific building in the region. Among various other descriptors of each building, the features, for instance, specify the geographical region or the building materials used. The label to be predicted, `damagegrade`, represents the level of damage dealt during the earthquake.

The competition provides a training dataset, a testing dataset and a submission template which can be used to create a submission file. The submission file can then be uploaded to the website to produce a score of the achieved submissions.

# 2 Baseline

## 2.1 Feature Selection

In this section, we describe a really important part of the development of our solution, which is the feature selection [1]. Although we performed feature selection in an incremental way, we decided to group all the different improvements performed on the feature selection, from scratch until the last version of the feature selection used in the final solution.

Essentially, we first used all the features to obtain an initial result, to gain some understanding of how important feature selection was going to be. As the result felt miserable, we decided to perform feature selection, which was first performed using domain knowledge, and it was later improved by making use of decision trees, which allowed us to gain information on the relevance of the features themselves.

### 2.1.1 Feature Selection using Domain Knowledge

The first step in order to improve the results was to perform an in-depth read of the different features. Their description can be found in the challenge's website [**?** ]. As a result of an initial analysis, it was decided to keep the following characteristics:

- `age`: we understood that the age of the building would have an impact in its damage degree. Thus, we decided to include it.
- `area_percentage`: we considered that the damage degree of a building somehow depended on how big the building was. Thus, we also included this characteristic.
- Variables related to the building material were also included, e.g. `has_superstructure_adobe_mud`, `has_superstructure_mud_mortar_stone`, `has_superstructure_stone_flag`. The reason why they were included is that, using domain knowledge, we can understand that the material must have a real impact when determining the damage degree of the building, thus it was also included.
- `geo_level_x_id`: the 3 variables associated with the location of the building were also added. This is because we understood that the distance from the earthquake's epicentre could play an important role when determining the damage degree of the building.

As a result of this domain-knowledge-based feature selection, we obtained a direct impact in the different algorithms, thus proving that feature selection is key to maximize the performance of our classifier.

### 2.1.2 Feature Selection using Decision Trees

As we've learned from the lectures, implementations of decision tree classifiers will order each feature, and therefore a decision, based on its importance and amount of information to be gained from making it. More specifically, the most important decisions will be made in a lower depth of the tree, reducing the overall amount of decisions that have to be made. After a decision tree classifier has been trained, the feature importance list can simply be exported and used for analysis and feature selection purposes.

[1] Refer to: `Task02_SupervisedLearning_Baseline_Barba_Guerrero_Schmidt.ipynb`.

It is important to note that we preprocess the input dataset to the classification process using the OneHot-Encoding for compatibility reasons, rolling out the amount of features from 38 to 68. In a first attempt of narrowing down the most important features using decision trees, we started training the classifier algorithms with the feature selection from our previous step already applied. Therefore, the amount of input features to the classifier is 17. By doing this, we were able to further decrease the amount of input features by eliminating those which don't contribute much information value.

However, thanks to the feedback provided, performing a feature selection process before training the decision tree classifier turned out to be an unnecessary generalization. Assuming that we forgot an important feature in our preliminary feature selection using domain knowledge, we would not be able to detect this error. Furthermore, since the f1-score of the classifier itself does not turn out to be higher than other techniques in the baseline work, the feature selection from decision trees should be improved as much as possible. So, in a second attempt, the decision tree classifier is trained with all the 68 features. Figure 1 shows the feature importance of the twenty most relevant features. We decide to pick the seven most relevant features, which sum up to describe 85% of the most important features of the dataset.


**Fig. 1** Plot of most important features, extracted from trained `DecisionTreeClassifier`.

Taking a look at the domain-based feature selection, we notice that the 4th most important feature "`foundation_type_r`", which is a product of the OneHot-Encoding, was overlooked. We extract the seven most important features (see figure 1), until `height_percentage`, as the most important values to be used in our reduced set of features.

## 2.2 Baseline Development

In order to develop the baseline of the project, we decided to apply three simple algorithms to create initial classifiers that we could test to see how good or bad they perform[2]. The baseline would be determined by the algorithm with the best performance in the competition, i.e. real data.

Thus, the idea is to obtain a baseline that can be improved with more complex algorithms to improve in a more fine-grained manner the performance of the classifier.

[2]Refer to: `Task02_SupervisedLearning_Baseline_Barba_Guerrero_Schmidt.ipynb`

### 2.2.1 Naive-Bayes

The first algorithm that we used in order to obtain a classifier is Naive-Bayes. In this case, the algorithm does not require parametrization, thus its application is simpler when compared to other algorithms.

However, Naive-Bayes does have different possible models that can be used, and the model to choose greatly depends on the features that were selected.

In our case, the first execution used all the different features and ComplementNB as a model. The reason why we decided to choose ComplementNB over the rest of the models is that we performed an exploratory data analysis that showed unbalanced data. ComplementNB is a good model when managing such data, thus it looked like a decent decision when it was performed. However, the result was not that good, as we only reached an f1-score of 0.2648 (which is actually worse than randomly determining the labels – 0.33).

As a result, we decided to apply improvements. The first improvement we came up with is to perform a smarter feature selection. Thus, it is here when we applied the domain-knowledge-based feature selection explained in section 2.1. With that feature selection, we applied again ComplementNB as the model to be used, obtaining an f1-score in the test dataset of 0.2561.

That last result proved that using the ComplementNB model was not such a good choice, because although data selection seemed to be just fine, the results were in fact worse. If we pay attention to the domain-knowledge-based feature selection, we can easily check that almost all variables are categorical and actually binary variables. BernoulliNB is a model that is known for being most effective when working with binary variables, as a consequence, we chose to apply the Bernoulli NB model keeping the same feature selection.

This last improvement led to a model that provided an f1-score of 0.5690 on the test dataset and 0.5665 of the driven data competition, which was already a good starting point for our baseline.

### 2.2.2 KNN

The second classification algorithm used is KNN. Contrary to Naive-Bayes, this algorithm requires a bit of parametrization, which will be discussed in the following paragraphs.

Implementing a basic KNeighborsClassifier with a k value of 8, taken randomly, the result obtained in the competition is a huge 0.7013, which at first sight seems like a big improvement from the previous model. It is remarkable to say that the first version of our KNN model used the features obtained via domain knowledge.

The first parameter we decided to tune is the number of neighbours. For this purpose, the easiest way to do this is to use the model to predict the output label using different numbers of neighbours, and comparing the accuracy of those trials. We compared the accuracy of both uniform and inverse distance, obtaining the result shown in figure 2.

And, while the best decision could be choosing uniform and a number of neighbours of around 64, we know that, from our empirical observations, the optimal result is provided by using a value for k of 32. This could be most likely due to the phenomena of overfitting. With this change, we increase the accuracy up to 0.7060. Another change included in the model is the use of Manhattan distance instead of

the default Euclidean distance, which is less appropriate for our model. Thus, the accuracy reaches 0.7121.

However, there are a couple more changes that can be included in our model: Better feature selection: with the most important features obtained using decision trees, the accuracy of the model slightly raises to 0.7128. Normalization of values: some numerical columns, such as the geo id columns, the age, or the area and height percentage, present disparate values that can be normalized to make every one of them fit in a range from 0 to 1. With this easily applied technique, we reach a final accuracy value of 0.7215, the highest obtained in our baseline, which makes us barely reach the top 1000 submissions.

Figure 3 shows the confusion matrix of the last model.

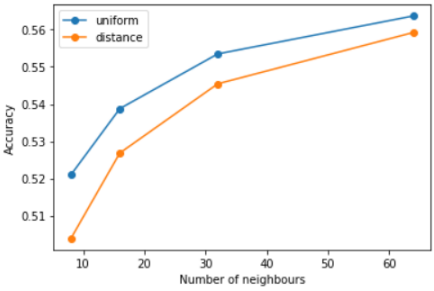


**Fig. 2** KNN: Manual tuning of parameters.

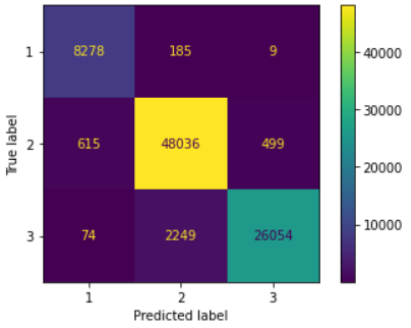


**Fig. 3** KNN: Confusion matrix.

As we can observe, our model is very robust at this point, but still fails to classify some 3rd grade earthquakes, as it considers them as 2nd grade, which is the main weakness the Naive-Bayes model also had.

### 2.2.3 Decision Trees

The third and last classification algorithm we applied to the dataset are decision trees. As already stated in the above section, the classifiers main advantages for this task are found in the feature selection capabilities, at least within the baseline step. In this section, we'll be taking a closer look into the parametrization and training procedure for the decision tree classifier.

As mentioned, the classifier is trained with the full set of features of our dataset. As an initial step, we train the DecisionTreeClassifier without any parameters, which yields an f1-score of 0.6528, which is already a great result. However, it is not as good as the best f1-score we got from KNN.

For decision trees, we are able to make adjustments to the parameters `class_weight`, `criterion`, `max_depth`, `min_samples_split` and `min_samples_leaf`.

From the exploratory data analysis, we know that the labels of our dataset are unbalanced (see figure 4). Therefore, we define the class weights accordingly in a ratio of 5:30:18.

The criterion, as well as the maximum depth, aren't trivial to determine. Therefore, we use an experimental approach to find good values for these parameters. For every possible depth, ranging from 1 to twice the amount of available features and

the two available criteria, we train the classifier and plot the resulting f1-score (see figure 5). Using this technique, we found that the criterion is not a critical parameter here, and the score stops improving after a depth greater than 20.

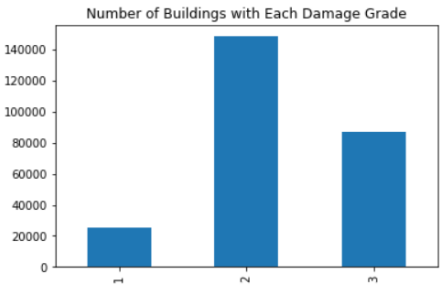


**Fig. 4** Exploratory Data Analysis: Distribution of labels.

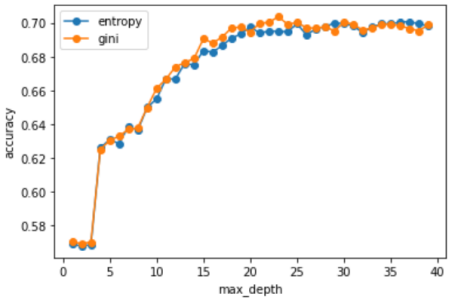


**Fig. 5** Decision Trees: Manual tuning of parameters.

For the two remaining parameters, the minimum samples split and minimum samples leaf, we use the values provided from the lecturers resources, as they seem to improve the f1-score the most. With all of these parameters applied, the achieved f1-score of the classifier is 0.6966, which is worse than the score obtained by KNN.

# 3 Optimizations

## 3.1 Random Forests

The developed decision tree classifier within the Baseline didn't turn out to be a good choice in terms of achieving the best accuracy. So, using Ensemble techniques, we'll be using this section to explain the training process of a Random Forest Classifier, as well as the hyperparameter tuning process we've performed[3].

As a starting point, we ran the `RandomForestClassifier` model provided by sklearn without any complex parameters to create a baseline F1 score. The only parameter we use is the amount of estimators `n_estimators` determining the amount of Decision Trees in our Random Forest. Using 100, we achieve a score of 0.7020, which is interesting, considering that our manually tuned decision trees achieved a score 0.77% worse.

Generally, comparing Random Forests to Decision Trees, there are even more parameters which are also even less intuitive to understand. For instance, instead of five parameters for the final classifier, we use eight. These, adding to the problem, each interact with each other. This makes choosing the right combination of parameters quite difficult. The problem of choosing the combination yielding the most accurate results can be found using specialized search tools. Within the project, we used both Randomized Search, and Grid Search.

Randomized search is a technique where a range of hyperparameter values is defined, and a random sample of these values is selected and used to train and evaluate the model. This technique is not exhaustive and was used to gain a general

[3] Refer to: `Task02_SupervisedLearning_RandomForests_Barba_Guerrero_Schmidt.ipynb`.

idea of the ideal hyperparameter ranges. In multiple cycles, we modified a list of parameters to test, and evaluated the outcomes.

After about four to five hours of code executions, we compiled a list of insights during four testing cycles. Figure 6 shows the final values within cycle 3 and 4.

| Cycle | #n_iter, #cv | #fits | Best parameters | Best F1 | execution time | insights for next cycle |
|---|---|---|---|---|---|---|
| 3.2 (ran locally) | 50,3 | 150 | {'n_estimators': 475, 'min_samples_split': 2, 'min_samples_leaf': 8, 'max_features': 'auto', 'max_depth': 32, 'bootstrap': True} | 0.7081 | 40min | - fix `min_samples_leaf` to 8<br>- try `max_depth` around the range of 30-60<br>- replace `n_estimators` = 250 with 550<br>- try cv = 2 to run higher `n_iter` |
| 4 (ran locally) | 96,2 | 192 | {'n_estimators': 475, 'min_samples_split': 4, 'min_samples_leaf': 8, 'max_features': 'sqrt', 'max_depth': 40, 'bootstrap': True} | 0.704 | ~44min | - ideal `max_depth` seems to be somewhere between 30-40 |

**Fig. 6** Iterative insights during Random Forest Hyperparameter Tuning.

Up until now, the best F1 score we could achieve is 0.7081 (0.7123 as evaluated on DrivenData.org) suggesting that there might not be as much potential to this technique as we initially hoped for.

In order to find the final set of hyperparameters, we use the Grid Search technique. Grid search is a technique where a grid of hyperparameter values is defined, and the model is trained and evaluated using all possible combinations of these values. This can be an exhaustive and time-consuming process, but it is effective for finding the optimal hyperparameter values for a model. The outcome values of the Randomized Search and those very close to them are chosen for the grid of possible parameters. By doing this, we unfortunately weren't able to improve our F1 score in comparison to the Randomized Search approach and yielded a score of 0.704.

Taking a look at the improvements related to Decision Trees and its Ensemble version, the Random Forest, we see that this type of classifier is not suited enough for the classification task at hand - Other classifiers developed in this project had achieved a score greater than 0.74. Figure 7 shows all results side-by-side, including a comparison to the best result of the competition at the time of writing, the 12th of December 2022.

| Technique used | F1-Score | Comparison w/ best possible result |
|---|---|---|
| DT without Tuning | 0.6545 | 86,59% |
| DT w/ manual Tuning | 0.6910 | 91,42% |
| DT + Bagging | 0.7091 | 93,82% |
| RFs without Tuning | 0.7020 | 92,88% |
| RFs w/ tuned Hyperparameters | 0.7123 | 94,24% |

**Fig. 7** Random Forest: Achieved improvements from Hyperparameter Tuning.

## 3.2 XGBoost

After developing a proper baseline model that gave us results of up to 0.7215 accuracy, the next idea is to start implementing more and more complex models to try to reach the top spots. Using XGBoost, a priori, we expect better results (with the same features obtained from the decision trees), as it is known for being a better model that could be good when trying to make these kinds of predictions that we are trying to do. Historically, it has been good to make predictions in this competition, so we try to use it to improve our results in it [4].

Using the base model without further hyperparameter tuning, we get a f1-score of 0.7183, which seems pretty good at first. Regarding hyperparameter tuning, it can be done in these ways:

1. We will perform random search to obtain possible good values for the hyperparameters.
2. We will perform grid search to obtain the "optimal" values for the hyperparameters.

Using the first option, we can get rank 695, as we got a result of 0.7364. On the other hand, the second option gets us to rank 668, with a score of 0.7378. However, we still believe that it can be improved, either by normalizing the data or using Bayes search. Unfortunately, none of these seem to produce better models, most likely because it did not finish the execution (it did not converge) in the latter case.

In this section, AdaBoost and GradientBoost were also tested, but failed to show improvements versus the previous models.

## 3.3 CatBoost

After the previous tests, CatBoost was used, in order to look for a better model than the previously mentioned [5]. The idea in this section is to take advantage of CatBoost virtue to work with categorical variables and apply it to the 3 geo level columns.

Since it requires very little parametrization, it is quite easy and fast to implement, providing a noticeably high f1-score.

## 3.4 LightGBM

Another good strategy to improve the model accuracy is to use LightGBM for gradient boosting as well[6]. For the cross validation, we used 5 folds of 100 candidates each, equalling 500 fits in total.

This model, along a proper parametrization, also seems promising at first, so we will include it in other complex models.

The link to the Google Colab document for LightGBM is the following:

## 3.5 Basic Stacked Model

Having considered all the previous models, we can dedicate ourselves in this section to construct a stacked model containing several of the previously developed ones[7].

[4] Refer to: `Task02_SupervisedLearning_XGBoost_Barba_Guerrero_Schmidt.ipynb`.
[5] Refer to: `Task02_SupervisedLearning_CatBoost_Barba_Guerrero_Schmidt.ipynb`.
[6] Refer to: `Task02_SupervisedLearning_LightGBM_Barba_Guerrero_Schmidt.ipynb`.
[7] Refer to: `Task02_SupervisedLearning_BasicStackedModel_Barba_Guerrero_Schmidt.ipynb`.

In our case, the model providing the best results consists in the combination of the XGBoost one, the baseline KNN model and the baseline DT model. All values here are normalized for improved performance after testing the not normalized ones.

Regarding the stacked model, we will use stratified K-fold cross validation with 5 splits, and logistic regression. Testing the basic stacked model gives us a f1-score of 0.7499, representing a huge 0.7423 value in the competition, for a rank of 532. As always, this difference may be caused by a bit of overfitting in the model.

Here, one would think that including the boosted model would significantly help to achieve a better result, but it turns out it is not the case. We tried adding both the LightGBM model and the CatBoost model, only for us to get a f1-score of 0.7483, versus the previous 0.7499.

# 4 Extensions

## 4.1 Outlier Feature Clustering

Another great idea could be to add new features using Laplace's rule[8]. In our case, we will only use these features obtained from conditional probabilities, each of them representing the probability of each earthquake to present a damage grade of 1, 2 or 3, given its geo level 1, 2 and 3, as shown in figure 8.

| building_id | prob_d1_g1 | prob_d2_g1 | prob_d3_g1 | prob_d1_g2 | prob_d2_g2 | prob_d3_g2 | prob_d1_g3 | prob_d2_g3 | prob_d3_g3 |
|---|---|---|---|---|---|---|---|---|---|
| **802906** | 0.086461 | 0.665354 | 0.248185 | 0.003704 | 0.251852 | 0.744444 | 0.000000 | 0.162162 | 0.837838 |
| **28830** | 0.034277 | 0.446174 | 0.519549 | 0.010050 | 0.492462 | 0.497487 | 0.062500 | 0.812500 | 0.125000 |
| **94947** | 0.021627 | 0.393378 | 0.584996 | 0.082386 | 0.316477 | 0.601136 | 0.029412 | 0.360294 | 0.610294 |
| **590882** | 0.129718 | 0.739603 | 0.130678 | 0.019512 | 0.853659 | 0.126829 | 0.032258 | 0.838710 | 0.129032 |
| **201944** | 0.046959 | 0.568370 | 0.384672 | 0.029865 | 0.591522 | 0.378613 | 0.008197 | 0.614754 | 0.377049 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **688636** | 0.083215 | 0.779516 | 0.137269 | 0.172414 | 0.724138 | 0.103448 | 0.000000 | 0.928571 | 0.071429 |
| **669485** | 0.013066 | 0.179388 | 0.807546 | 0.003831 | 0.061303 | 0.934866 | 0.000000 | 0.020408 | 0.979592 |
| **602512** | 0.013066 | 0.179388 | 0.807546 | 0.024024 | 0.057057 | 0.918919 | 0.045455 | 0.090909 | 0.863636 |
| **151409** | 0.354986 | 0.559142 | 0.085872 | 0.507429 | 0.452947 | 0.039624 | 0.220339 | 0.766949 | 0.012712 |
| **747594** | 0.021627 | 0.393378 | 0.584996 | 0.000000 | 0.249249 | 0.750751 | 0.000000 | 0.097561 | 0.902439 |

**Fig. 8** Outlier Feature Clustering: Table of features.

Since we have 9 features now, we can apply PCA to do dimensionality reduction as it will save us time and computational cost. Choosing 2 components explains 87% of the total variance. By using k-means++ and trial and error, we decided to use 6 different components to cluster all the records in the dataset, providing us with the result seen in figure 9.

Finally, by using different boosting algorithms previously mentioned, we can obtain accuracy values of up to 0.74, which are slightly worse than the previous ones, but still very good. However, there is a final strategy to try to reach the top spots in the ranking.

[8] Refer to: `Task02_SupervisedLearning_OutlierFeatureDetection_Barba_Guerrero_Schmidt.ipynb`.

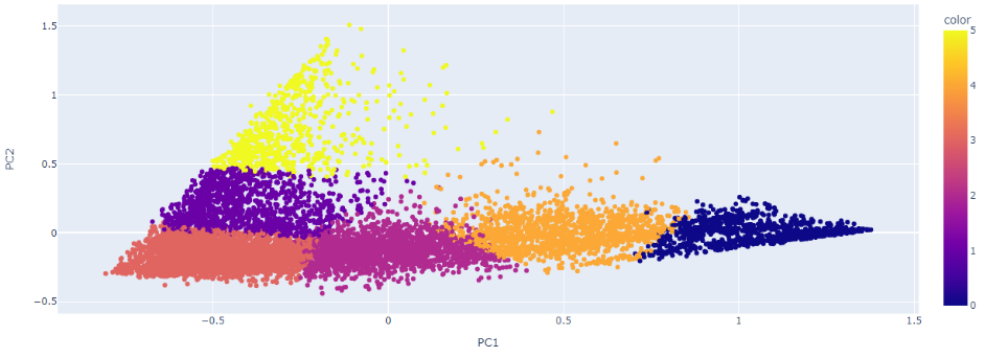


**Fig. 9** Outlier Feature Clustering: Visualization.

## 4.2 Two-phase classification stacked model

Finally, our last idea was to divide the classification part into two differentiated phases, since there are more samples from damage grade of 2 than from the other two [9]. Thus, our aim is to first classify each record into one of the two following classes: damage grade of 2 or class 0 (containing all records from the other damage grade records).

After this first classification, if we do not include the record into class 2, we must classify it into damage grade of 1 or damage grade of 3. This strategy prevents us from one of the main problems we used to have in the baseline: the model wrongly classifying records from the third class into the second one.

For both phases, we will use a stacked model with both a CatBoost model and an XGBoost model, since including LightGBM here does not help us improve the results. This implementation will provide us a confusion matrix shown in figure 10 for the first and second phase, respectively.

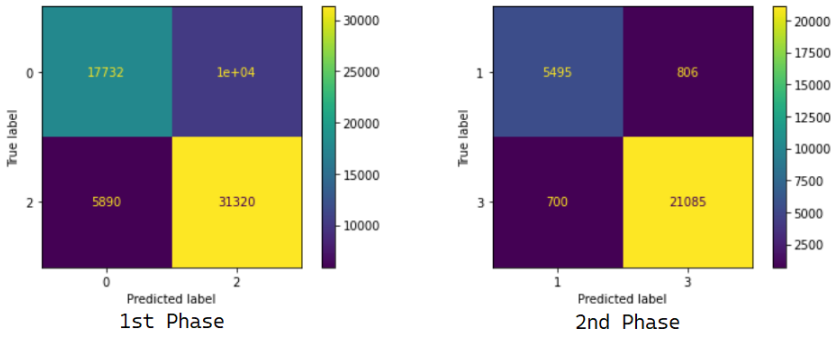


**Fig. 10** Two-phase classification stacked model: Confusion Matrices for both phases.

This model has shown to be the best out of the ones tried, with a value of 0.7438, meaning a rank of 459[10].

[9] Refer to: `Task02_SupervisedLearning_TwoPhaseClassificationStackedModel_Barba_Guerrero_Schmidt.ipynb`.
[10] Rank 459, as of 13.12.2022, uploaded by user `RaulBarbaRojas` on 2022-12-11 19:48:05.