Report Homework – Machine Learning

Multi-UAV conflict prediction

Paolo Ferretti - 2039579

Executive Summary

Over recent years, machine learning methods, with their ability to extract information from a vast amount of data and automate processing, have found applications in multiple fields. This success is based on algorithms and on mathematical models that have been developed in the past decades. Moreover, excellent libraries built around Python like NumPy and Scikit-learn have contributed increasing the popularity of Machine Learning.
Beside it, we have whitnessed the birth of Deep Learning, which has become extremely popular due to the concept of neural network. For the scope of this report, I will not go into details on this part. In fact, the aim of the work presented here, is to solve a supervised learning task without the use of neural networks.
In supervised learning we want to learn a model from labeled training data to make predictions about unseen data. We talk about classification when we deal with a supervised learning task with discrete class labels, while we talk about regression for supervised learning task where the outcome is a continuous variable.

In particular, the task is to perform a multi-class classification and a regression problem on a severely imbalanced dataset using domain independent techniques. The presence of imbalance may negatively influence the performance of the model, leading rapidly to overfitting.

Since the data rarely comes in the form and shape that is necessary for the optimal performance, the preprocessing of the data will be one of the most important step. Many machine learning algorithms require that the selected features are on the same scale. This is achieved by transforming them with feature scaling techniques. Hence, I will present different methods to scale features.

Note that dimensionality reduction techniques will not be addressed in this report since they are out of the scope of the homework. In any case, these techniques are useful and very effective when the dataset contains features that are highly correlated which I will show you that this is not really the case.

Then, my goal will be to analyse how to train different models effectively on the imbalanced dataset and to compare them. But before comparing different models, I will have to decide upon a metric to measure performance. I'll show that the commonly used accuracy, will not be the most accurate metric to evaluate the models. Thus, I will select other performance metrics.

Finally, the default parameters of the learning algorithms will not probably be optimal. Therefore, I will perform hyperparameter optimization techniques to fine tune the performance of the models.

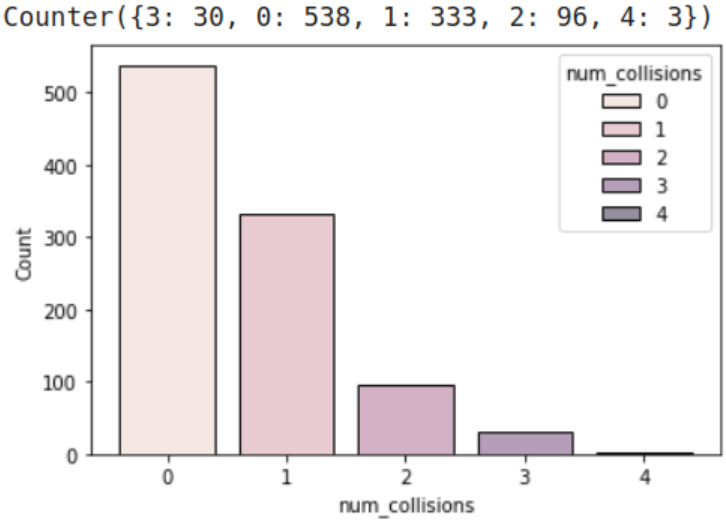
1. The dataset

The dataset is one of the most important part of a machine learning problem. It contains all that we need to train and validate our model. The main problem is that the majority of the datasets do not have the shape we want. Some may contain noise and outliers or do not have enough data. Thus, with little and poor information, the performance of the learning algorithm would be severely affected.

Other classification datasets may not have equal number of instances in each class. There are problems where this class imbalance is the normality. Think about fraud detection or medical analysis, where the number of negative samples are commonly higher than the number of positives.

Here, I'm dealing with a similar case. I have to work with an extremely imbalanced dataset. It contains 1000 samples,. Each of them it's composed by 35 continuous input features and two output columns, namely '*num_collisions*' and '*min_CPA*'. For the classification problem, '*num_collisions*' represent the belonging class of each sample and it can have 5 different discrete values, while the '*min_CPA*' column contains the continuous values for the regression problem.

As I said, the dataset is very imbalanced. The number of samples for each class are distributed in the way reported below:



As you can see, there are 538 samples belonging to class zero, 333 to class one, 96 to class two, 30 to class three and only three samples belong to class four. This imbalance must be handled to obtain acceptable performances.

To conclude the overview of the dataset, let's say that it is stored in a tabular separated value file. Hence, it will be read using the read_csv function in pandas library and it will be stored into a pandas DataFrame.

2. Libraries

For this task, I will mostly refer to the scikit-learn library, which is one of the most popular and complete open-source libraries of machine learning. I will also use the Matplotlib and Seaborn libraries to plot useful graphs. Finally, I will use NumPy which offers good functions to handle data and Imblearn for oversampling techniques like SMOTE and ADASYN.

3. Data Analysis and Preprocessing

3.1 Data extraction

As I said before, the quality and amount of data are key factors to determine how well a machine learning algorithm can learn. Therefore, it is critical to examine and preprocess a dataset before feeding it to a learner.

Obviously, the first thing I did is to load the dataset into a DataFrame using pandas. A pandas DataFrame is a data structure containing labeled axes and it disposes of good functions to manipulate the data. Each row in a dataframe represents one sample and each column a different feature.

Then, I've divide input and output extracting the columns from the DataFrame. The input can be extracted taking the first 35 columns of each row. The output for the multi-class classification problem is contained in the 36th column, while the output for the regression problem is in the last column.

I've also wanted to add some lines of code to extract only part of the input features. I've done it to be able to test different configurations and for visualization purposes. This was easily done with the method '*query*' which evaluates a boolean expression and passes the result to DataFrame.loc.

3.2 Missing values and correlations

3.2.1 Missing values

Since it is common to work with datasets with missing values due to error in the collection process or just due to unavailable information, I've first checked my dataset to eventually take care of them. In fact, many tools are unable to handle such missing values or will produce unpredictable outputs. Thus, to identify possible missing values I've used the '*isnull*' method. As, you can see, there aren't missing values, so I've proceeded with further analyses.

```
Number of null cells in df: 0
```

3.2.2 Correlations

I've also displayed correlations between features in the dataset to see if there were visibly redundant features that could potentially be removed. I've done this using the '*corr*' method which computes pairwise correlation of columns. Here, I will not report the correlation table, it could be visualized on the provided Colab Notebook.

What it is easily visible is that there are almost no correlations between features. For this reason, I left the them as they were.

3.3 Data Visualization

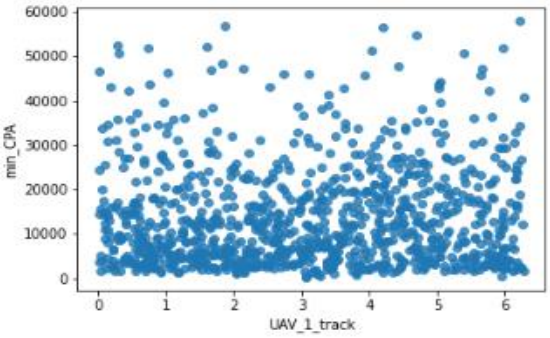
For what concerns data visualization, I have to distinguish between the classification and regression tasks.

For classification, I've plotted the histogram reported above, a pie plot showing the percentage of examples belonging to each class, a two-features scatter plot and a one-feature density plot.

Regression – Regular Plot

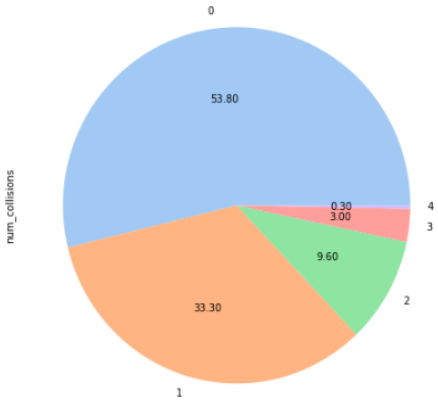
It displays the relation between the input column and the target value. With this function it is also possible to plot a linear regression model fit. In this case, I've decided to set the '*fit_reg*' parameter to False to visualize only the data.

Obviously, it is impossible to visualize a 35-dimension input data without feature reduction. So, I've decided to plot just one feature. This will not probably give much information.



Classification - Pie plot

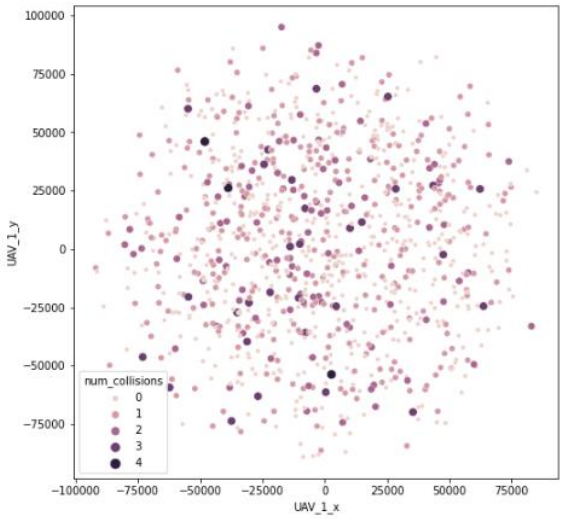
Just to visualize dataset imbalance and the percentage of samples belonging to each class in a straight-forward way.



Classification - Two-features scatter plot

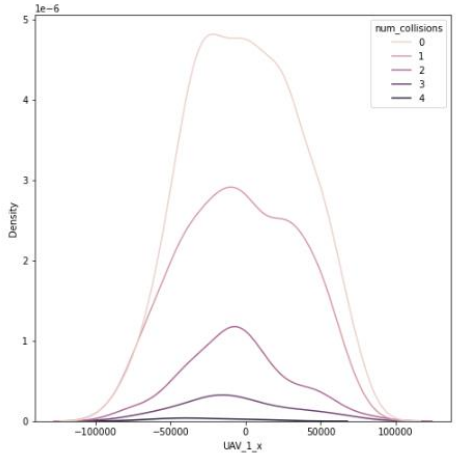
It takes two selected features from the input and it plots them in a scatter plot with different scatter dimensions and colors.

It is useful to see how the features are related one to the other considering also the belonging class. With this function it is also possible to plot the relation between one input features and the output value.



Classification - One-feature density plot

It displays the distribution of one selected feature of the samples with respect to the belonging class. From the distributions it is possible to say whether a feature is discriminant or not for the target class. Below, I've reported the plot of the distribution of one feature just as an example. As you can see, this feature alone will not be very discriminant to classify the samples.



3.4 Feature Scaling

Since the majority of machine learning algorithms behave much better if features are on the same scale, feature scaling is a crucial step in the preprocessing pipeline.

For this reason, I wrote a function to take care of this. I've added the possibility to choose the preferred feature scaling technique among four possible options.

Below, I report the four options with a brief description for each of them:

- **Maximum Absolute Scaling**: it returns values of the input data between -1 and 1. It takes the input and it divides it by the maximum absolute value on that column.
- **Normalization via Min-Max Feature Scaling**: it scales the feature between 0 and 1. It's computed by subtracting from the input the minimum value in the column and subsequently dividing by the difference between the maximum and minimum value.
- **Standard Scaler**: it scales the data into a distribution with zero mean and variance 1.
- **Robust Scaling**: it removes the median and scales the data according to the quantile range. It is recommended when working with small datasets.

Personally, i've used the Standard Scaler which can be more practical for many machine learning algorithms. In fact, models like SVM initialize the weights to zero or to small random values. With Standard Scaler, I've centered the feature columns at mean zero with standard deviation 1 so that the columns have the same parameters as a standard normal distribution.

3.5 Partitioning the dataset

Before training a model, it is important to divide the input data in two subset, one for training and one for testing. This is very important because comparing predictions to true labels in the test set can be seen as the unbiased performance evaluation of the model. A straightforward way to randomly partition the dataset into separate test and training sets is to use the '*train_test_split*' function.

This function randomly split x and y into separate training and test dataset. I've assigned 30 percent of the examples to *X_test*. The test size influences the performance of the model. With too many samples in the test set, the model will not be able to train properly. On the other hand, using too many samples in the training set will probably lead to better performance of the model but it is possible that the evaluation on the test set will not be accurate.

Since i have an imbalanced dataset, providing the class label array y as an argument to the '*stratify*' parameter (in classification) ensures that both training and test datasets have the same class proportions as the original dataset. I'm sure that at least one example belonging to the last class will be in the *X_test* set.

3.6 Imbalanced dataset and resampling techniques

Since the dataset is imbalanced, it is important to try techniques that could help improving performances.

In fact, class imbalance influences a learning algorithm during model fitting. Typically, machine learning algorithms optimize a reward or loss function computed as a sum over the training examples that it sees. Thus, the decision rule is going to be biased toward the majority class.

One way to deal with imbalanced class proportion is to assign a larger penalty to wrong predictions on the minority class. This can be easily done with the parameter '*class_weight*' set to 'balanced', which is implemented for most classifier.

Other popular strategies to treat class imbalance include upsampling the minority class, downsampling the majority class or doing both of them. Due to the fact that I did not have a huge dataset, I've decided just to upsample the minority classes trying different techniques and evaluating the results. The upsampling has been applied after the dataset split just to be sure that the test data will be un unbiased estimator. Note that I've evaluated both the models trained with the original training set and the models trained with the resampled training set.

Below, I report the three options with a brief description for each of them:

- **SMOTE**: or Synthetic Minority Oversampling TEchnique works by selecting examples that are close in the feature space, drawing a line between them and drawing a new sample at a point along that line.
- **ADASYN**: this method is similar to SMOTE, but it generates different number of samples depending on an estimate of the local distribution of the class to be oversampled.
- **RandomOverSampler**: it's the naivest strategy to generate new samples by randomly sampling with replacement of the current available samples.

At this point, I can proceed to describe the models used for classification and regression.

4. Classification

4.1 Model selection

The first thing I did, was to reason on which models were available to solve this multi-class classification task. Since it is always recommended to compare performance of different learning algorithms to select the best model, I've decided to consider 3 of the most popular and powerful methods. The following models are available on the scikit-learn library.

Below, I briefly describe each of them:

- **Support Vector Machine Classifier**: it is a powerful learning algorithm based on maximizing the margin, namely the gap between the decision boundary and hyperplanes. It tends to have a lower generalization error, whereas models with small margins are more prone to overfitting. While dealing with a nonlinearly separable dataset, it introduces slack variables leading to the so-called soft-margin classification, to allow convergence of the optimization in the presence of misclassifications. The use of slack variables introduces the variable C. This parameter controls the penalty for misclassification. Large value of C corresponds to large error penalties. Therefore, it can be tuned to control overfitting. This model can be easily kernelized to solve nonlinear classification problems. The idea behind kernel methods is to create nonlinear

combinations of the original features to project them in an higher-dimensional space where the data becomes linearly separable. This is done using a kernel function, that can be interpreted as a similarity function between pair of examples.

- **Decision Tree Classifier**: this model breaks down our data making a decision at each step. The decision tree model learns a series of questions to infer class labels of the examples. It starts at the tree root and split the data on the feature that results in the largest information gain. It can build complex decision boundaries. The deeper the decision tree, the more complex it becomes, leading rapidly to overfitting.
- **Random Forest Classifier**: It is an ensemble method which combines multiple decision trees. The idea behind random forest is to average multiple decision trees that individually overfit the data to build a more robust model with a better generalization error. It draws a random bootstrap sample and grows a decision tree from this. Then it aggregates the prediction to assign the class label. This method is robust to noise and the larger the number of trees, the better the performance at the expense of the computational cost.

4.2 Performance Metrics

Generally speaking, accuracy is a useful metric to measure the performance of a model. But in this case, since the dataset is extremely imbalanced (538 samples belonging to the first class), accuracy can return a value of 53,8% while the model is always classifying samples as they belong to the majority class. This means that the model hasn't learned anything useful from the provided features. The situation described is the so-called Accuracy Paradox.

This is not what I want, so I've decided to tackle the problem considering other performance metrics. More specifically, I've used '*balanced_accuracy*' and '*F1_macro*'.

- **Balanced accuracy**: it is necessary when dealing with imbalanced dataset. It is calculated as (sensitivity+specificity)/2, where the sensitivity is the true positive rate, and the specificity is the true negative rate.
- **F1 Macro**: the macro-average F1 score is computed using the arithmetic mean of all the F1 scores per each class. This method treats all the classes equally regardless of their support values.

When working on an imbalanced dataset that demands attention on the negatives, balanced accuracy does better than F1. In the end, I've considered both to evaluate the results.

4.3 Baseline Performances

In this section, i analyse the main metrics for each model (from the previous section) trained with the training set. I've created a function that does exactly this, it displays the metrics of the models (fit with training set) on train and test set in a compact way. Remember that all the models have been trained using the default values for the parameters. In the Colab Notebook i've displayed other metrics which are not so relevant. I did not report them here to have a more comprehensible report.

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	RandomForestClassifier	1.000000	0.198820	1.000000	0.154031
1	SVC	0.283262	0.193093	0.269758	0.149081
2	DecisionTreeClassifier	1.000000	0.214069	1.000000	0.210705

Figure 1: Metrics of models trained with the original training set and then evaluated on training and test sets

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	RandomForestClassifier	1.000000	0.192099	1.000000	0.177478
1	SVC	0.974706	0.201124	0.974676	0.199125
2	DecisionTreeClassifier	1.000000	0.168193	1.000000	0.170704

Figure 2: Metrics of models trained with the resampled training set and then evaluated on training and test sets

As you can see, the metrics are very bad, and the models tend to overfit. This is even more visible if we look at the confusion matrices.

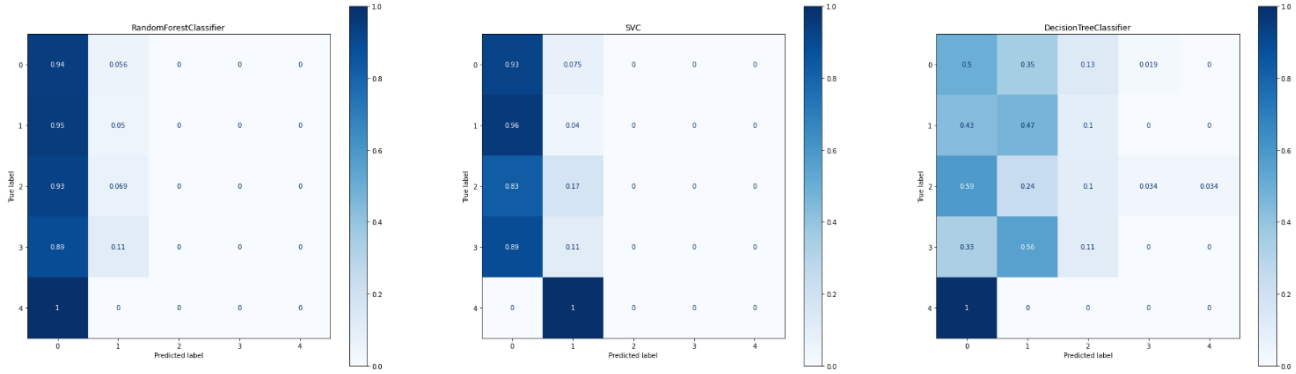


Figure 3: Confusion matrices of models trained with the original training set

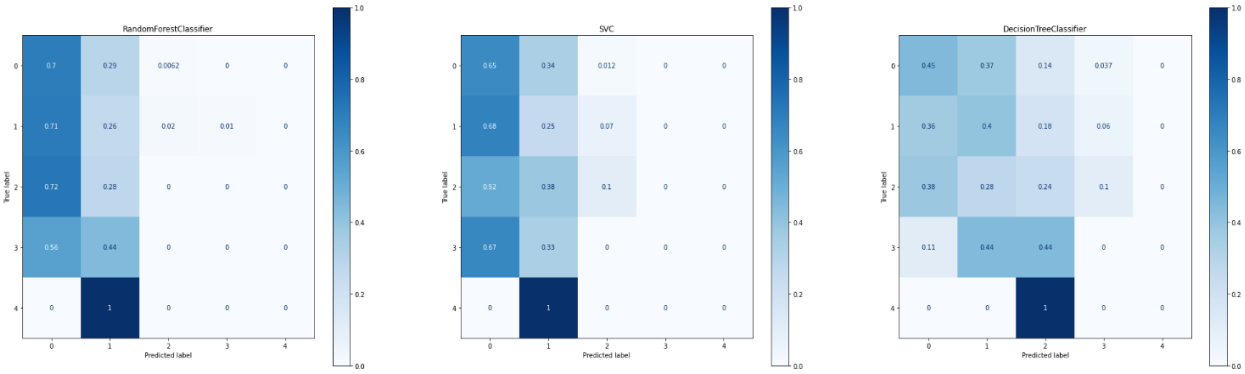


Figure 4: Confusion matrices of models trained with the resampled training set

The models tend to weight more the majority class and classify everything as belonging to the first two classes. I started from these performances, trying to work on the parameters of the models to improve the classification of the critical classes.

From these preliminary analyses, I've observed how resampling the minority classes slightly improves the performance.

4.4 Hyperparameter Optimization

Since the performances were bad, i've analysed these classifiers to see if they can be improved using GridSearch. This approach is quite simple, it's a brute-force exhaustive search where I've specified a list of values for different hyperparamaters. It performs k-fold cross-validation and selects the best parameters according to a scoring measure. I've used as scoring balanced accuracy, which i thought to be one of the most representative performance metrics. GridSearch is available on scikit-learn.

For each learning method, I've selected some parameters and search in the space of possible combinations. For Support Vector Classifier I've selected C which is a regularization parameter that defines the trade-off between misclassification and margin, the kernel which specifies the type of kernel to apply to obtain a kernelized SVC, gamma which is the kernel coefficient, the degree in the case of polynomial kernel and the '*class_weight*' parameter.

For what concerns Decision Tree Classifier, I've selected as parameters the max depth of the tree, the type of split (random or best) and the '*class_weight*' parameter.

For Random Forest Classifier, I've selected as parameters the max_depth of the trees, the number of trees to build, the '*class_weight*' parameter and the number of features to consider when looking for the best split (max_features).

The possible values that I put in the dictionary '*param_grid*' were obtained after reading the documentation of the classifiers and after several trials.

All the documentation about these models can be found here:

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

4.5 Evaluation of optimized models

After the Grid search it was necessary to train the new models on the train set and to evaluate it on both the test and train set. This is done to visualize the performance and to deduce if the models overfit the data. Here I've reported the best models found for the original dataset and for the resampled dataset.

For the original dataset, the best models found are:

- SVM classifier with polynomial kernel of degree 2. The parameter '*class_weight'* is set to 'balanced', as expected. It decreases the weight of records in the common class to balance the weight of the whole class.
- DecisionTreeClassifier with max depth equal to 5, '*class_weight*' set to 'balanced' and random splitting to increase the randomness of the process.
- RandomForestClassifier that builds 50 estimators with max depth equal to two and '*class_weight*' set to 'balanced'.

```
svc = SVC(C=0.7, class_weight ='balanced', degree = 2, kernel = 'poly', gamma = 'scale')
decision_tree = DecisionTreeClassifier(max_depth = 5, class_weight = 'balanced', splitter = 'random', random_state=2)
random_forest = RandomForestClassifier(n_estimators = 50, class_weight='balanced',  max_depth = 2, random_state = 4)
```

For the resampled dataset the situation is slightly different, and after the grid search, I've performed some hyperparameter tuning by hand. The best models found are:

- SVM classifier with polynomial kernel of degree 2. The parameter '*class_weight'* is set to 'balanced', as expected. Gamma is set to 0.2 and C to almost 0.01
- DecisionTreeClassifier with max depth equal to 10, and with best splitting.
- RandomForestClassifier that builds 12 estimators with max depth equal to 4. Adding classifier leads to better performances on the first classes.

```
svc_res = SVC(C=0.0094, class_weight = 'balanced', degree = 2, kernel = 'poly', gamma = 0.2)
decision_tree_res = DecisionTreeClassifier(max_depth = 10, splitter = 'best', random_state=2)
random_forest_res = RandomForestClassifier(n_estimators = 12, max_depth = 4, random_state =4)
```

4.5.1. Metrics and confusion matrices

Here, I've reported the main metrics obtained at the end.

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	SVC	0.815674	0.209388	0.698083	0.203313
1	DecisionTreeClassifier	0.624182	0.232347	0.429021	0.154902
2	RandomForestClassifier	0.624756	0.232808	0.524293	0.210994

Figure 5: Metrics of models trained with the original training set and then evaluated on training and test sets

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	SVC	0.866437	0.228361	0.863366	0.222807
1	DecisionTreeClassifier	0.938366	0.188882	0.938329	0.187676
2	RandomForestClassifier	0.756135	0.227317	0.747892	0.198398

Figure 6: Metrics of models trained with the resampled set and then evaluated on resampled and test sets

After this, I've plotted the confusion matrices of the models both on the test set and on the training set. I've done this to visualize better overfitting.

Test set

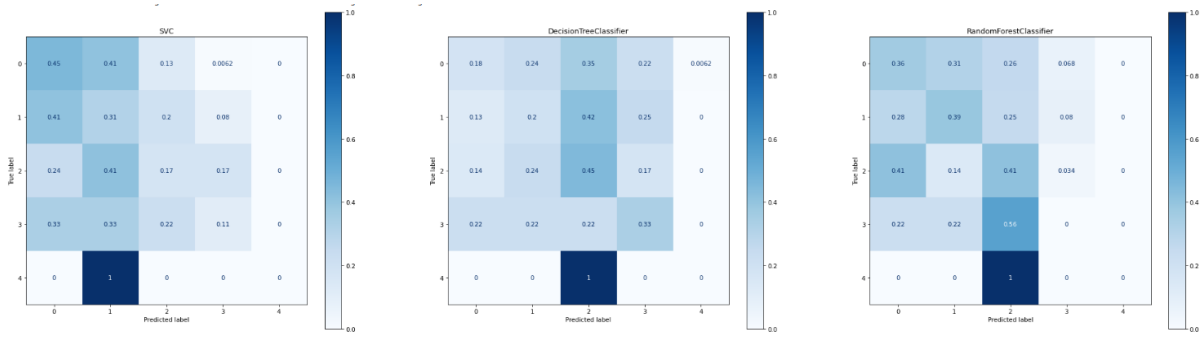


Figure 7: Confusion matrices of models trained with the original training set

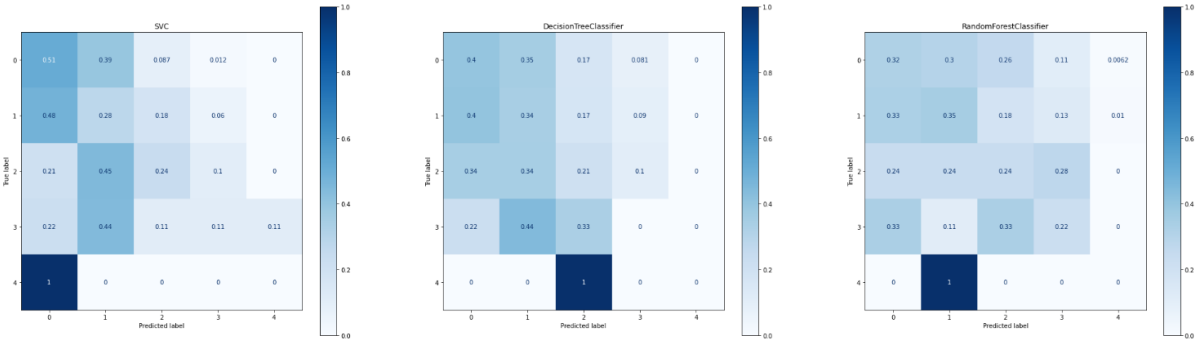


Figure 8: Confusion matrices of models trained with the resampled training set

Training set

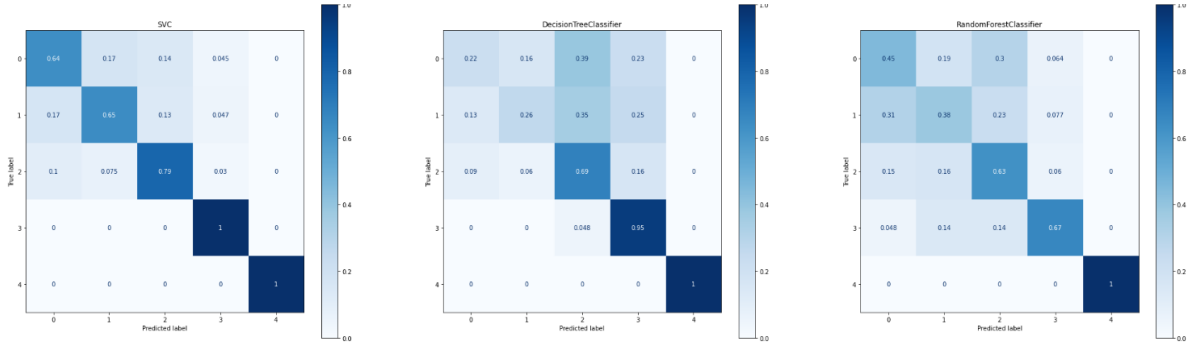


Figure 9: Confusion matrices of models trained with the original training set (training set)

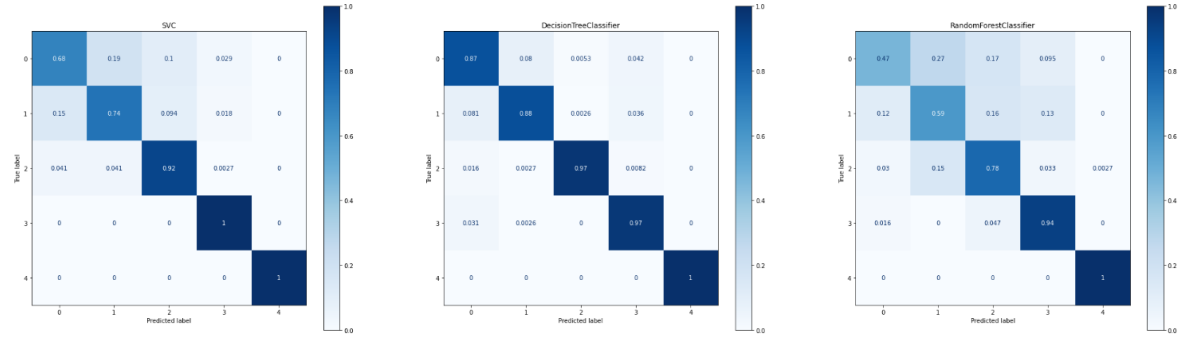


Figure 10: Confusion matrices of models trained with the resampled training set (on resampled train)

4.6 Considerations

These models slightly increase the performance on the classification task with respect to the baseline. Still all of them tends to overfit very quickly and they are not very accurate. The performances with the resampled dataset tend to be worse except for random forest. Surely, with the resampled set they tend to generalize better the least populated classes. On the other hand, overfitting is very high and applying regularization techniques results in a drop on the performance metrics. I've tried multiple combinations of parameters by hand and using RandomSearch with poor results. In my opinion, the best models are random forest and kernelized svm with the parameters reported above and trained on the resampled dataset.

4.7 Bagging

Bagging is an ensemble learning technique where each estimator receives a random subset of examples from the training dataset. Once the individual estimators are fit to the bootstrap samples, the predictions are combined. I've used BaggingClassifier from the scikit-learn library with SVC with polynomial kernel as base estimator.

Metrics

Here the main metrics are reported:

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	BaggingClassifier	0.862439	0.238506	0.812173	0.248946

Figure 11: Bagging with kernelized SVC metrics (original dataset)

	Model	balanced_accuracy_train	balanced_accuracy_test	F1_macro_train	F1_macro_test
0	BaggingClassifier	0.879734	0.2445	0.876764	0.236944

Figure 12: Bagging with kernelized SVC metrics (resampled dataset)

Confusion Matrices

These are the confusion matrices of the predictions on the test set of the two bagging classifiers. One trained with the original dataset, one trained with the resampled dataset.

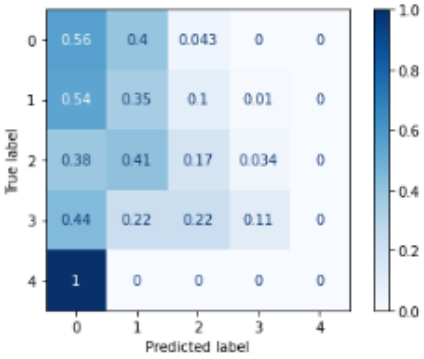


Figure 13: Confusion matrices of predictions of bagging classifier trained on the original dataset

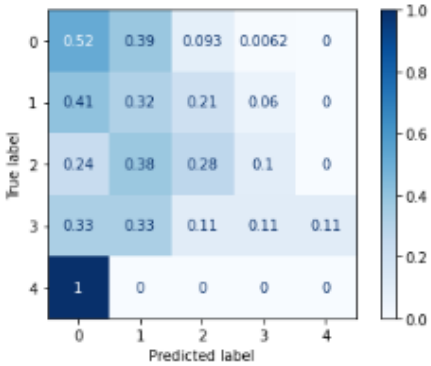


Figure 14: Confusion matrices of predictions of bagging classifier trained on the resampled dataset

The performances of bagging with kernelized SVC seems to be better. It has higher accuracy on the most populated class and better-balanced accuracy, mantaining at least some information about the last two classes. Obviously, it requires more time to train when increasing the number of estimators.

5. Regression

In this part, I present what I've done for regression analysis. The goal of regression is to model the relationship between one or multiple features and the continuous target variable. In contrast to classification, regression analysis aims to predict outputs on a continuous scale rather than categorical class labels.

In my case, the outputs are stored in the 'min_CPA' column. The preprocessing part is the same of the classification part except for resample. I've performed feature scaling using StandardScaler and I did not oversample the input data.

5.1 Model selection

The first thing I did, was to reason on which models were available to solve this regression task with multiple features. Since it is always recommended to compare performances of different learning algorithms to select the best model, I've decided to consider 3 of the most popular and powerful methods. The following models are available on the scikit-learn library.

Below, I briefly describe each of them:

- **Decision Tree Regressor**: it divides the input space into smaller regions that become more manageable. It works with arbitrary features and does not require any transformation of the features when dealing with nonlinear data because it analyzes one feature at a time, rather than taking weighted combinations into account.
- **Random Forest Regressor**: it is an ensemble technique that combines multiple decision trees. It usually has better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Moreover, they are less sensitive to outliers in the dataset and do not require much parameter tuning.
- **SVR**: it's the same concept of SVM applied to regression

5.2 Performance Metrics

This is not what I want, so I've decided to tackle the problem considering other performance metrics. More specifically, I've used '*r2 score*' and '*Mean Squared Error*'.

- **R squared score**: also called coefficient of determination. It is the proportion of the variation in the dependent variable that is predictable from the independent variables. It normally ranges between 0 and 1 but it can have negative values when the performance is worse than average prediction without considering the features.
- **Mean Squared Error (MSE)**: it measures the average of the squares of the errors, that is the average squared difference between the estimated values and the actual values.

5.3 Baseline Performances

In this section, i analyse the main metrics for each model (from the previous section) trained with the training set. I've created a function that does exactly this, it displays the metrics of the models (fit with training set) on train and test set in a compact way. All the models have been trained using the default values for the parameters.

	Model	R2_train	R2_test	MSE_train	MSE_test
0	RandomForestRegressor	0.860770	-0.092462	1.744869e+07	1.235262e+08
1	SVR	-0.057682	-0.019155	1.325521e+08	1.152372e+08
2	DecisionTreeRegressor	1.000000	-0.937373	0.000000e+00	2.190614e+08

Figure 1R: Metrics of models evaluated on training and test sets

The models work badly. Random Forest and Decision Tree overfit the data while SVR seems to ignore the features in input at all. The mean squared error is very high. I've performed hyperparameter optimization by hand and using gridsearch. In the following section I present the results.

5.4 Hyperparameter Optimization

For grid seach, for each learning method, I've selected some parameters and search in the space of possible combinations. For Support Vector Regressor I've selected C which is a regularization parameter that defines the trade-off between error and margin, the kernel which specifies the type of kernel to apply, gamma which is the kernel coefficient and the degree in the case of polynomial kernel.

For what concerns Decision Tree Regressor, I've selected as parameters the max depth of the tree, the type of split (random or best) and the criterion, which is the function to measure the quality of a split. I've selected among the possible criteria the "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node, "absolute_error" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node, and "poisson" which uses reduction in Poisson deviance to find splits.

For Random Forest Classifier, I've selected as parameters the max_depth of the trees, the number of trees to build and the criteria.

The possible values that I put in the dictionary '*param_grid*' were obtained after reading the documentation of the regressors and after several trials.

All the documentation about these models can be found here:

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

5.5 Evaluation of optimized models

After the Grid search it is necessary to train the new models on the train set and to evaluate it on both the test and train set. This is done to visualize the performance and to deduce if the models overfit the data. Here I've reported the best models found:

- SVR with polynomial kernel of degree 6 with C equal to 1 and gamma equal to 0.2.
- DecisionTreeRegressor with '*max_leaf_node*' equal to 2, best splittingand the poisson criterion.
- RandomForestRegressor that builds 90 estimators with max depth equal to three and poisson as criteria.

```
# polynomial svr
poly_svr = SVR(kernel='poly',C= 1, degree=6,  gamma=0.2)

# decision tree regressor
dt = DecisionTreeRegressor(criterion='poisson',
                           splitter='best',
                           max_depth=None,
                           max_leaf_nodes=2)
# random forest regressor
ran_forest = RandomForestRegressor(n_estimators = 90, max_depth =3, criterion ='poisson')

best_models = [poly_svr, dt, ran_forest]
```

The corresponding metrics are reported. The performances are slightly better.

	Model	R2_train	R2_test	MSE_train	MSE_test
0	SVR	0.998981	0.071102	1.277305e+05	1.050317e+08
1	DecisionTreeRegressor	0.016844	-0.012001	1.232123e+08	1.144283e+08
2	RandomForestRegressor	0.060839	-0.031164	1.176987e+08	1.165952e+08

Figure 2R: Metrics of optimized models evaluated on training and test sets

These models slightly increase the performance on the regression task with respect to the baseline. With SVR with polynomial kernel, i've got still a bad r2 score but visibly better with respect to the others, even the MSE is lower. Still the SVR model tends to overfit while the other 2 have lower R2 score on the train set but higher R2 score on the test set.

In my opinion, the best model is kernelized SVR with the parameters reported above.

With these results in mind, I've tried bagging.

5.6 Bagging

I've used BaggingRegressor from the scikit-learn library with SVR with polynomial kernel as base estimator.

The results are reported here:

	Model	R2_train	R2_test	MSE_train	MSE_test
0	BaggingRegressor	0.274796	0.059084	9.088490e+07	1.063929e+08

Applying bagging to kernelized SVR does not increase the general performance on the dataset. However, it tends to reduce overfitting and the R2 score and the MSE are similar with respect to the single SVM regressor.