University of Rome "La Sapienza"
Department of Ingegneria Informatica, Automatica e Gestionale

Machine Learning
Homework 1

Multi-UAV Conflict Prediction

Gianmarco Scarano

Matricola:
2047315

December 9, 2022

Contents

Chapter 1

Introduction

1.1 Multi-UAV Conflict Risk Analysis

The main topic about this first homework is related to UAVs conflicts.

Given the dataset, we do have a:

- Classification problem: Compute and estimate the total number of conflicts between UAVs.
- Regression problem: Predict the minimum Closest Point of Approach (CPA) among all the possible pairs of UAVs.

1.2 Goal

Our goal for this homework is to show and demonstrate the differences between the results given by different algorithms.

1.3 Code writing

The Jupyter Notebook related to this report has been written through Visual Studio Code and tested afterwards multiple times also on Google Colaboratory where cells did run perfectly fine on both machines.

Chapter 2

Classification

2.1 Dataset

The dataset is given in a form of .tsv file where we have:

- 35 features as input features
- 1 column representing the 5 classes: 0, 1, 2, 3, 4
- 1000 samples in total

Through the use of `Pandas` library, we read and store the .tsv file in a DataFrame as follows:

ID	**UAV_1_track**	**UAV_1_x**	**UAV_1_y**	**UAV_1_vx**	**...**	**UAV_5_target_y**	**num_collisions**
0	0.027	-62300	-59305	6.705	...	33187	3
1	4.023	-17220	47439	-167.65	...	32972	0
2	1.841	-19900	59030	208.71	...	32929	0
3	3.621	-48565	-11986	-113.51	...	32872	0
4	2.318	52665	-47498	177.79	...	32453	0

where we store only the `num_collisions` value, as we are dealing with a Classification problem.

2.2 Working scheme

As we know, the dataset here is highly imbalance, meaning that the percentage distribution of the classes over the Y vector has the following scheme:

- 538 samples of class 0 (53.8%)
- 333 samples of class 1 (33.3%)
- 96 samples of class 2 (9.6%)
- 30 samples of class 3 (3.0%)
- 3 samples of class 4 (0.3%)

We could then, possibly, think of splitting our work in 3 big parts, which involve:

- Using the original dataset without optimizations
- Apply pre-processing techniques
- Balancing class weights directly into the models

2.3 Using original dataset

In this section, we are going to use the .tsv dataset as it is, untouched and without any further optimization techniques. We expect the results to be very inaccurate, as we are about to give a highly imbalance dataset to our models.

For this test and also further ones, we split the X set and Y set in two different sets of Training (66,7%) and Test (33,3%) using the `train_test_split()` function present in the `Sklearn` library, generating different numpy arrays as: UAV_X_train and UAV_y_train & UAV_X_test and UAV_y_test.

2.3.1 First experiments

As for the very first experiments, our plan is to run different tests on the `UAV_X_train` set, in order to check which method works better than another.

In order to evaluate these different tests, we utilize the `cross_validate()` function from Sklearn, which evaluate metrics by cross-validating the training set as follows:

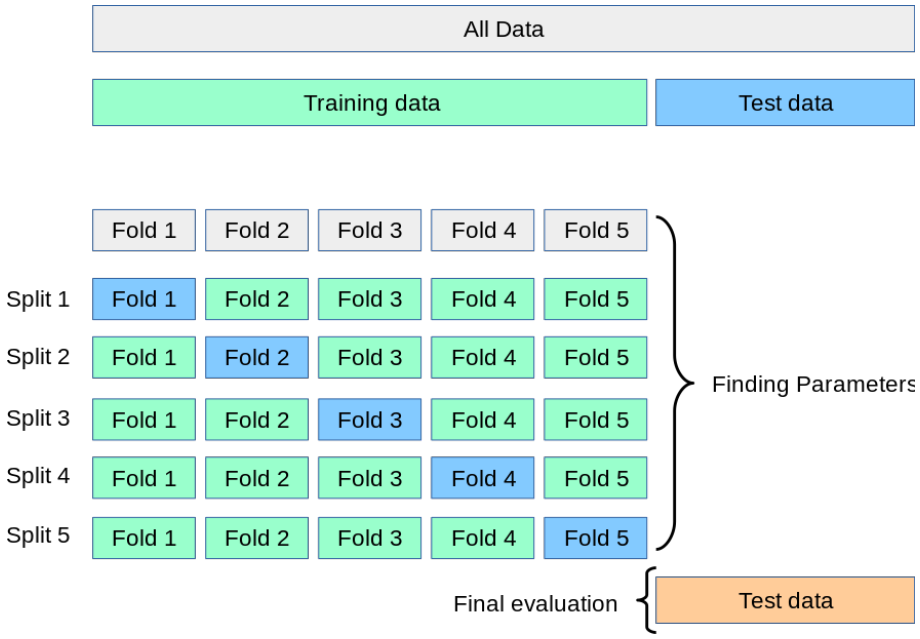

Figure 1: Impact of the Cross-validation scheme over the Training and Test set.

Further, in order to test the robustness of the classifiers, we also fit and evaluate them on the Test set (as per above).

The metric used for this kind of task is `balanced_accuracy_score()` from Sklearn, which basically returns the average of recall obtained on each class. We need this because, again, we are dealing with an imbalanced dataset, otherwise we would have used a simply accuracy score function.

As for the folds, we pass 2 folds into the function, since there are just 2 samples of class 4 in the Training set (while just 1 sample in the Test set).

So, we test the following Classification methods:

- Logistic Regression Classifier
- Random Forest Classifier
- Gaussian Naive Bayes Classifier
- Gradient Boosting Classifier
- SVM Classifier
- VotingClassifier (which includes the previous 5 classifiers)

From our tests, the results are the following:

Classifier	Accuracy Test Set	Accuracy CV
Logistic Regression	0.18569	0.19998
Random Forest	**0.21232**	0.20826
Naive Bayes	0.20414	0.21672
Gradient Boosting	0.18736	0.20065
SVC	**0.21070**	0.20742
VotingClassifier	0.20940	0.21790

This simple test shows how SVC and Random Forest Classification can perform better than others on our dataset, so we stick with those and keep running our experiments using these two classifiers.

2.3.2 Support Vector Machine

The main process of running our "further" experiment is basically following the workflow for the final evaluation, as shown here:

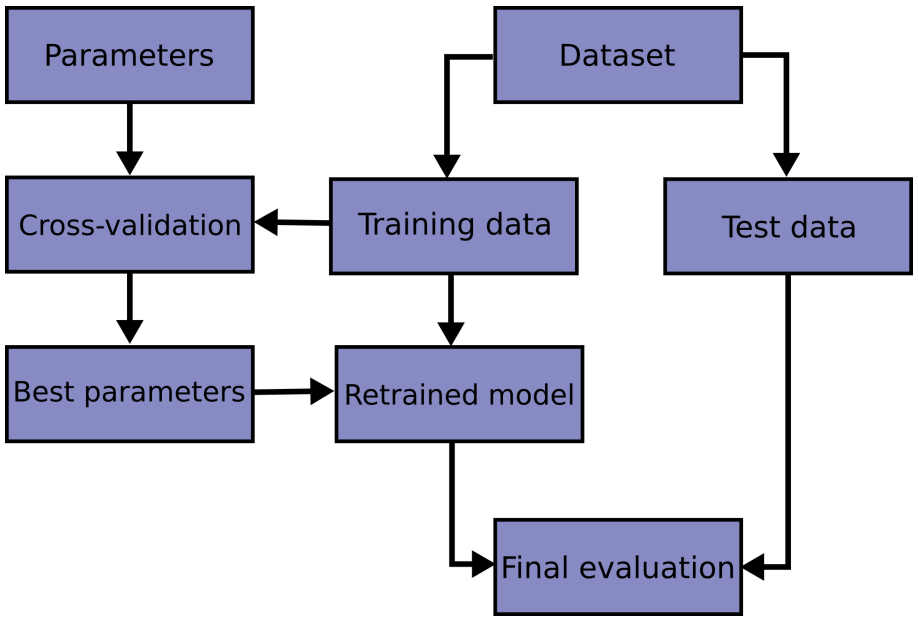

Figure 2: Use of Grid Search workflow

We already executed the first 2 rows of this workflow, so it's time for us to find the best parameters for our model, re-train it and finally testing it on the Test set.

Grid Search for SVM Classifier

In order to get the best hyper-parameters of the SVM, we deploy the Grid Search Algorithm from the `Sklearn` library. This will also be done for RFC later on. The parameters passed to the GS algorithm are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':('scale', 'auto')`
- `'kernel': ('rbf','sigmoid')`
- `CV: 2`

The GS algorithm returns that the best parameters are `C = 3.6`, `gamma = 'scale'`, `kernel = 'rbf'` and that the best accuracy is `0.226`. Note that this accuracy, as well as the one that we'll try later on the GS algorithm for RFC, is still the balanced accuracy over the mean scores extracted by the 2 folds.

We also run a different GS for the 'poly' version of SVM Classifier. This is because SVM Classifier would then uselessly fit the model with degree parameter values, which are ignored in all kernel but 'poly' itself. The parameters passed to the GS algorithm for this test are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':['scale']`
- `'kernel': ['poly']`
- `'degree': np.arange(1,5,1)`
- `CV: 2`

The best hyper-parameters for SVM Classifier with 'poly' kernel are: `C = 5.9, Degree = 2` with an accuracy of `0.249`

Final evaluation

Concluding our test on the SVM Classifier, we evaluate the Test set with the best parameters found through the Grid Search. For computational purposes, we assign a value of 2000 to the `cache_state` variable. The accuracy returned is of `0.22549`, which is way better than the one we had on the 'First experiments' subsection.

2.3.3 Random Forest Classifier

Continuing with the Random Forest Classifier, we use the same `random_state` variable (my matricola code - `2047315`) used for the SVM Classifier (in order to have homogeneity between the twos).

In RFC, we utilize two more variables called `Bootstrap` and `Warm Start` which respectively set the use of the whole dataset for building each tree and the reuse of the solution provided by the previous calls to `fit()`.

Grid Search for Random Forest Classifier

As with SVM Classifier, we deploy a Grid Search in order to retrieve the best parameters of the RandomForestClassifier. The metric used is basically the same as the one we used before in SVM Classifier.

The Grid Search returns us this parameters: `bootstrap: False`

`criterion: log_loss, n_estimators: 150, warm_start: True`, where accuracy is `0.216`.

Final evaluation

If we pass these parameters to a fresh new RFC model and evaluate this on our test data, we'd get `0.20699` as accuracy with 173 samples correctly predicted out of 333.

2.3.4 Results & Conclusion

Plotting the results

For visualization purposes, we plot the results we had from previous tests in a simple bar plot.

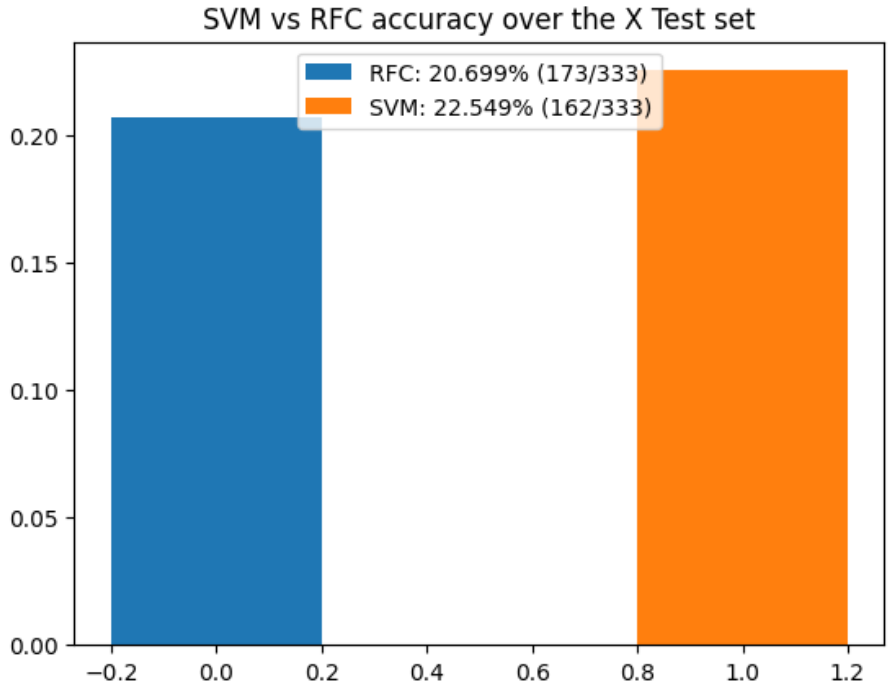

Figure 3: This plot, shows the difference in terms of accuracy of the two models tested on the Test Set, using the original/untouched dataset.

We can clearly see that even though the SVM Classifier has 2 points of difference from the RFC model, what happens is that RFC has correctly identified more samples in the Test set.

This happens because we used the `balanced_accuracy_score`, where it doesn't take into consideration the simple accuracy, but as we said, it relies on the the average of recall obtained on each class.

This could be easily demonstrated through a simple Confusion Matrix plot, which we will discuss in the next subsection.

Confusion Matrix

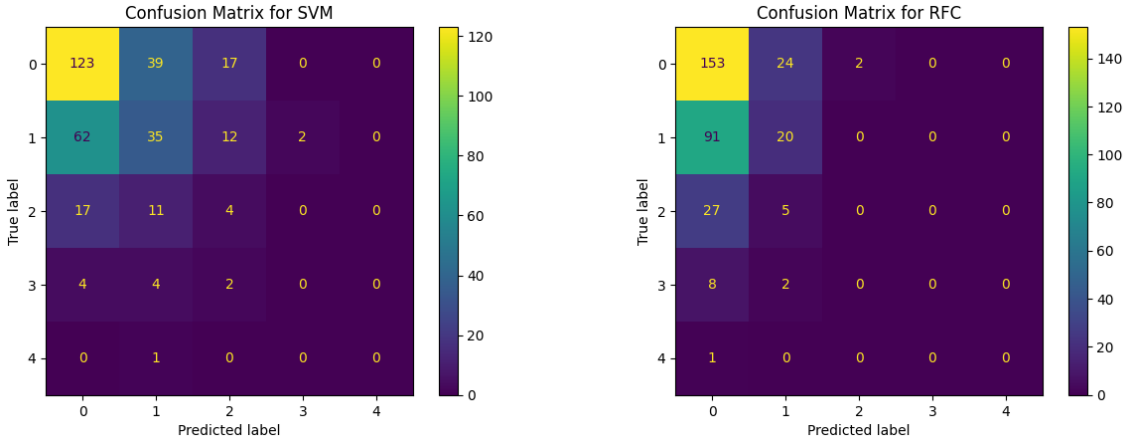

Figure 4: Confusion Matrix for both SVC and RFC on the Test Set, using the original/untouched dataset.

Class	Precision	Recall	F1-Score	Support
0	0.597	0.687	0.639	179
1	0.389	0.315	0.348	111
2	0.114	0.125	0.119	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.486	333
Macro avg	0.220	0.225	0.221	333
Weighted avg	0.462	0.486	0.471	333

Table 2.1: SVM Classifier - Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.546	0.855	0.667	179
1	0.392	0.180	0.247	111
2	0.000	0.000	0.000	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.520	333
Macro avg	0.188	0.207	0.183	333
Weighted avg	0.424	0.520	0.441	333

Table 2.2: RFC - Classification Report

As we said, in the SVM Classifier we have 4 correctly predicted samples for the class 2, differently from the RFC model which predicted 0 samples correctly for class 2-3-4. From these plots, we can also see how the predictions for the SVM Classifier are more "spread out" between the classes while RFC keeps a consistency (classes are mistaken between the real one and the one just after). We could also think that RFC performs better since it relies just on class 0 and 1, due to the fact that the accuracy on the classification report is based on a simple accuracy. In fact, if we look at the weighted avg, we see that SVM performs better (`0.471` vs `0.441`).

Test with AdaBoost have been carried out, but without any further improvement in the overall accuracy, which swings between `0.19` and `0.20` for both models.

2.4 Normalized & Balanced Dataset

In this section, we run another test starting all over again from the .tsv file. Here we do normalize our set of features, since it has values between `-104075.61` and `101510.15`, so using the `MinMaxScale()` class from the `Sklearn` library could be a great option for scaling all feature values between 0 and 1.

Indeed, our new X set has the following values:

ID	UAV_1_track	UAV_1_x	UAV_1_y	UAV_1_vx	...	num_collisions
0	0.003	0.167	0.161	0.508	...	3
1	0.640	0.422	0.741	0.165	...	0
2	0.292	0.407	0.804	0.907	...	0
3	0.576	0.245	0.418	0.271	...	0
4	0.368	0.817	0.225	0.846	...	0

We then split the X set and Y set in two different sets of Training (66,7%) and Test (33,3%) using the `train_test_split()` function present in the `Sklearn` library, generating different numpy arrays as: UAV_X_train and UAV_y_train & UAV_X_test and UAV_y_test.

2.4.1 Random Over Sampler

We then try to find a nice, simple and effective way of pre-processing our data.

As for the imbalance problem, we introduce the use of RandomOverSampler (*ROS*) technique by the Imbalanced Learn Team.

This simple tool is a multi-class re-sampling function, which balances the data by oversampling the samples in the class with less samples (by picking samples at random with replacement), re-sampling each class independently.

One important thing about the RandomOverSampler technique is that it needs to be carried only on the training test, otherwise we would introduce bias inside the Test set with bad consequences such as overfitting (the model would recognize only the classes that he has already 'seen' during the Training phase and so return high overall accuracy, but poor F1-score for the classes).

With this being said, the percentage distribution of the classes over the Y vector has now changed drastically in order to have more compact and distributed values.

Figure 5: Impact of ROS over the X set. Now all classes have 20% of features each.

2.4.2 First experiments

As we did in the first test, we run multiple experiments in order to check which method works better than another. In order to evaluate these different tests, we utilize the metric `accuracy_score()` from Sklearn, which is a simple accuracy metric (differently from the balanced one we used before) as we are not dealing with a imbalanced problem anymore.

As for the folds, we now pass 15 folds into the function, since there are more than 2 samples of class 4 in the Training set (which forced us to use 2 folds in the previous test).

So, we test the following Classification methods:

- Logistic Regression Classifier
- Random Forest Classifier
- Gaussian Naive Bayes Classifier
- Gradient Boosting Classifier
- SVM Classifier
- VotingClassifier (which includes the previous 5 classifiers)

From our tests, the results are the following:

Classifier	Accuracy Test Set	Accuracy CV
Logistic Regression	0.24324	0.58994
Random Forest	**0.51652**	0.90978
Naive Bayes	0.27027	0.52754
Gradient Boosting	0.40841	0.86241
SVC	**0.41141**	0.83730
VotingClassifier	0.44144	0.86630

This simple test shows how SVC and Random Forest Classification (again) can perform better than others on our dataset, so we stick with those and keep running our experiments using these two classifiers. Also both algorithms return a higher accuracy than the previous test.

2.4.3 Support Vector Machine

It's time for us to find the best parameters for our model, re-train it and testing it on the Test set.

Grid Search for SVM Classifier

The parameters passed to the GS algorithm are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':['scale']`
- `'kernel': ('linear','rbf','sigmoid')`
- `CV: 15`

The GS algorithm returns that the best parameters are `C = 5.5`, `gamma = 'scale'`, `kernel = 'rbf'` and that the best accuracy is `0.886`. Note that this accuracy, as well as the one that we'll try later on the GS algorithm for RFC, is still the accuracy over the mean scores extracted by the 15 folds.

We also run a different GS for the 'poly' version of SVM Classifier here. The parameters passed to the GS algorithm for this test are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':['scale']`
- `'kernel': ['poly']`
- `'degree': np.arange(1,5,1)`

- `CV: 15`

The best hyper-parameters for SVM Classifier with 'poly' kernel are: `C = 0.2, Degree = 4` with an accuracy of `0.877`

Final evaluation

Concluding our test on the SVM Classifier, we evaluate the Test set with the best parameters found through the Grid Search. For computational purposes, we assign a value of 2000 to the `cache_state` variable. The accuracy returned is of `0.450`, which is higher than the accuracy returned from the first test.

2.4.4 Random Forest Classifier

Grid Search for Random Forest Classifier

As with SVM Classifier, we deploy a Grid Search in order to retrieve the best parameters of the RandomForestClassifier. The metric used is basically the same as the one we used before in SVM Classifier.

The Grid Search returns us this parameters: `bootstrap: False`
`criterion: log_loss, n_estimators: 200, warm_start: True`, where accuracy is `0.922`.

Final evaluation

If we pass these parameters to a fresh new RFC model and evaluate this on our test data, we'd get `0.50751` as accuracy with 169 samples correctly predicted out of 333.

2.4.5 Result Conclusions

Plotting the results

For visualization purposes, we plot the results we had from previous tests in a simple bar plot.

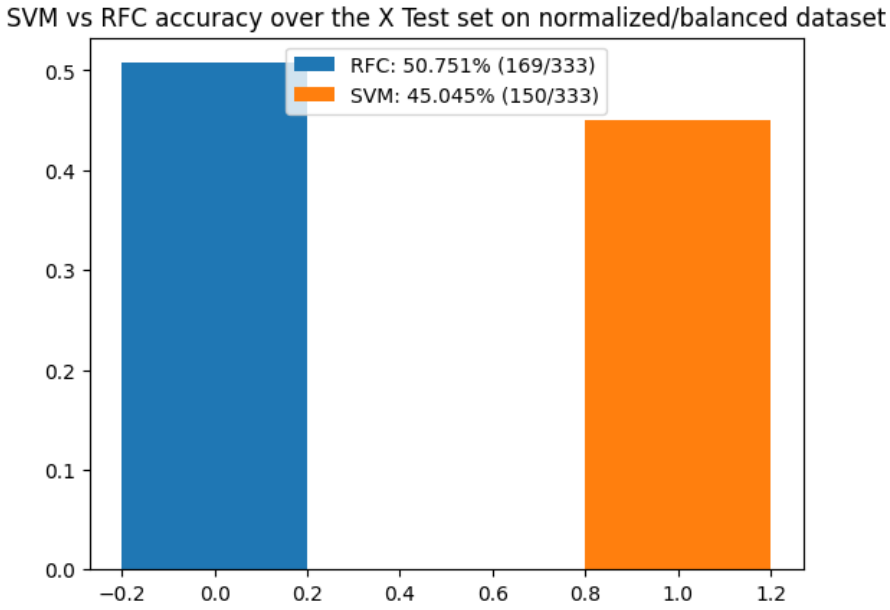

Figure 6: This plot, shows the difference in terms of accuracy of the two models tested on the Test Set, using the normalized & balanced dataset.

Here we used the `accuracy_score`, where it does take into consideration just a simple accuracy. We see how the SVM performs less better than the RFC model with a difference of 5 points. Further explainations are made in the next subsection.

Confusion Matrix

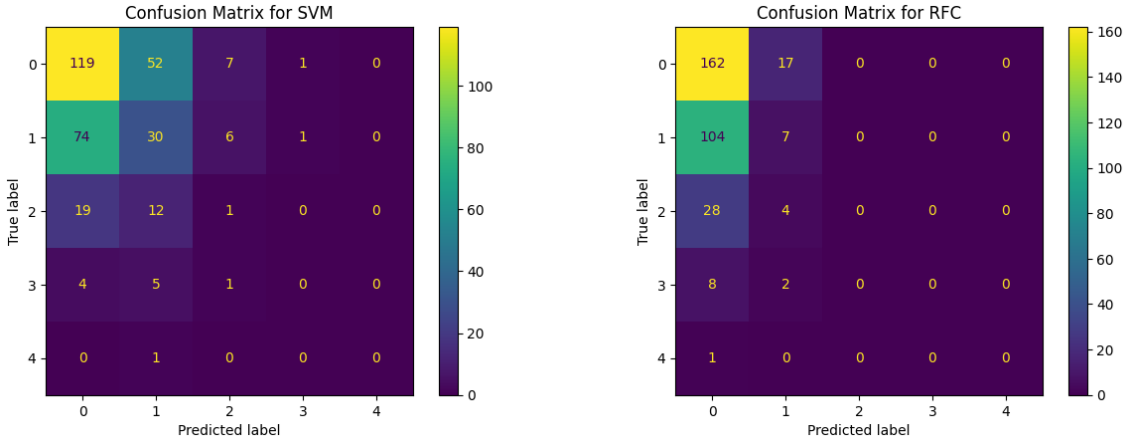

Figure 7: Confusion Matrix for both SVC and RFC on the Test Set, using the normalized & balanced dataset.

Class	Precision	Recall	F1-Score	Support
0	0.551	0.665	0.603	179
1	0.300	0.270	0.284	111
2	0.067	0.031	0.043	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.450	333
Macro avg	0.184	0.193	0.186	333
Weighted avg	0.403	0.450	0.423	333

Table 2.3: SVM Classifier - Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.535	0.905	0.672	179
1	0.233	0.063	0.099	111
2	0.000	0.000	0.000	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.508	333
Macro avg	0.154	0.194	0.154	333
Weighted avg	0.365	0.508	0.394	333

Table 2.4: RFC - Classification Report

By looking at the confusion matrix and classification report, we highlight the fact that the SVM has a F1-Score of `0.04` for class 2, something that RFC really doesn't predict (having `0.00` of F1-Score). Aside from that, for class 0 (which has not been oversampled), RFC performs way better with a F1-Score of `0.67` against `0.60` of SVM.

We can clearly see that, as the previous experiment, SVM Classifier recognizes samples from class 2, while the RFC model seems blind for samples in class 2-3-4 of the Test set.

Test with AdaBoost have been carried out, but without any further and substantial improvement in the overall accuracy.

2.5 Normalized & Balanced through class weights

In this section, we are going to analyze our third and last experiment for this classification problem, without the use of the RandomOverSampler. What we do, instead, is providing a weight for the classes to every model we testt through a parameter called `class_weight` inside each model.

The weights for the classes are calculated as follows:

$$w_i = \frac{n\_samples}{n\_classes * n\_samples\_class\_i}$$

In this way, we'll have that the class 4 (which has just 2 samples in the Training Set and 1 in the Test Set) will have more "weight" than class 0, 1, 2 and 3 (which have all less samples).

2.5.1 Representing the Dataset

It's time for us to plot the dataset. Since our dataset has 35 features, we are not able to plot all of them in a hyperplane. What we could do, though, is plotting (for every sample), the second and the third feature using the `seaborn` library, as shown:

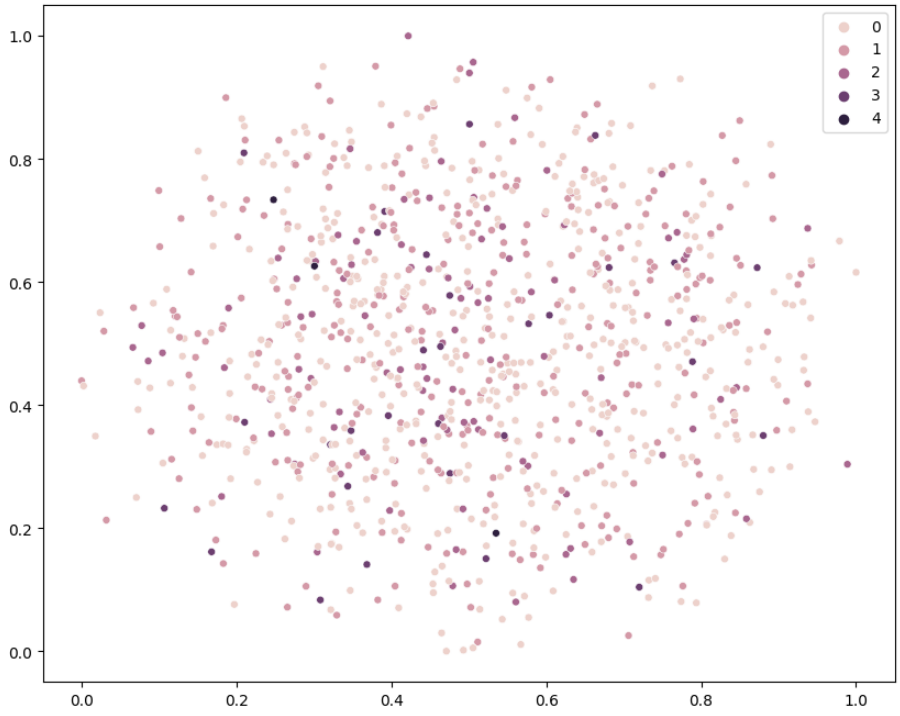

Figure 8: Feature 0 and 1 over the 1000 samples with their relative classes

2.5.2 First experiments

As we did in the first and second test, we run multiple experiments in order to check which method works better than another. In order to evaluate these different tests, we utilize the metric `accuracy_score()` from Sklearn as we are not dealing with an imbalanced problem anymore.

As for the folds, we pass 2 folds into the function, since there are just 2 samples of class 4 in the Training set (while just 1 sample in the Test set).

So, we test the following Classification methods:

- Logistic Regression Classifier
- Random Forest Classifier
- Gaussian Naive Bayes Classifier
- Gradient Boosting Classifier
- SVM Classifier
- VotingClassifier (which includes the previous 5 classifiers)

From our tests, the results are the following:

Classifier	**Accuracy Test Set**	**Accuracy CV**
Logistic Regression	0.22523	0.28036
Random Forest	**0.52553**	0.53822
Naive Bayes	0.45045	0.45729
Gradient Boosting	0.44745	0.47529
SVC	**0.35736**	0.39431
VotingClassifier	0.46246	0.48127

This simple test shows how Random Forest Classification (once again) can perform better than others on our dataset. We also take into consideration the SVM Classifier in order to stick with the same models for different experiments.

2.5.3 Support Vector Machine

It's time for us to find the best parameters for our model, re-train it and testing it on the Test set.

Grid Search for SVM Classifier

The parameters passed to the GS algorithm are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':('scale','auto')`
- `'kernel': ('linear','rbf','sigmoid')`
- `CV: 2`

The GS algorithm returns that the best parameters are `C = 5.9`, `gamma = 'scale'`, `kernel = 'rbf'` and that the best accuracy is `0.4648`. Note that this accuracy, as well as the one that we'll try later on the GS algorithm for RFC, is still the accuracy over the mean scores extracted by the 2 folds.

We also run a different GS for the 'poly' version of SVM Classifier here. The parameters passed to the GS algorithm for this test are the following:

- `'C' : np.arange(0.1,6,0.1)`
- `'gamma':['scale']`
- `'kernel': ['poly']`
- `'degree': np.arange(1,5,1)`
- `CV: 2`

The best hyper-parameters for SVM Classifier with 'poly' kernel are: `C = 0.3, Degree = 4` with an accuracy of `0.451`

Final evaluation

Concluding our test on the SVM Classifier, we evaluate the Test set with the best parameters found through the Grid Search. For computational purposes, we assign a value of 2000 to the `cache_state` variable. The accuracy returned is of `0.426`, which is way higher than the previous test tried in this chapter (+7 points).

2.5.4 Random Forest Classifier

Grid Search for Random Forest Classifier

As with SVM Classifier, we deploy a Grid Search in order to retrieve the best parameters of the RandomForestClassifier. The metric used is basically the same as the one we used before in SVM Classifier.

The Grid Search returns us this parameters: `bootstrap: True`

`criterion: gini, n_estimators: 300, warm_start: True`, where accuracy is `0.541`.

Final evaluation

If we pass these parameters to a fresh new RFC model and evaluate this on our test data, we'd get `0.537` as accuracy with 179 samples correctly predicted out of 333, which is slightly higher than the one we had on our first experiments (+1 point).

2.5.5 Result Conclusions

Plotting the results

For visualization purposes, we plot the results we had from previous tests in a simple bar plot.

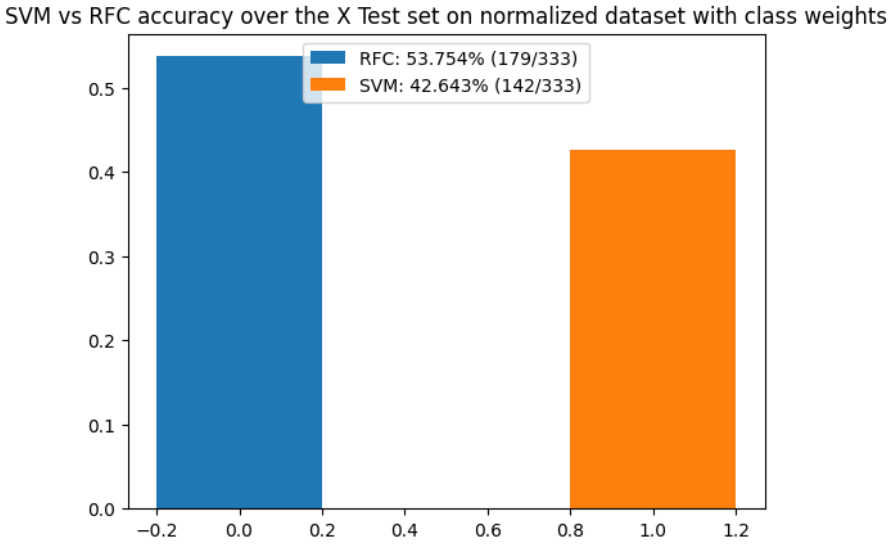

Figure 9: This plot, shows the difference in terms of accuracy of the two models tested on the Test Set, using the normalized & dataset with class weights.

Here we can see how easily the RFC performs way better than RFC (+11 points in accuracy / +37 samples predicted correctly).

Confusion Matrix

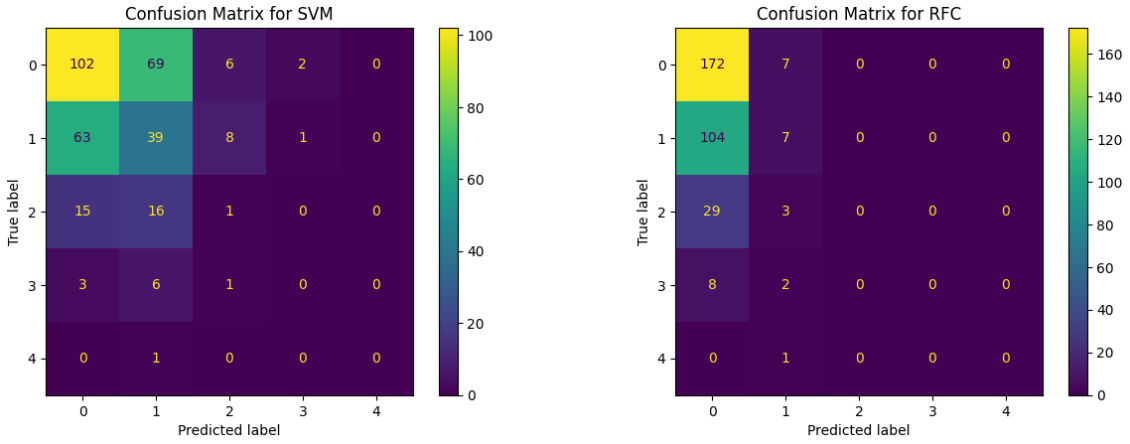

Figure 10: Confusion Matrix for both SVC and RFC on the Test Set, using the normalized dataset with class weights.

Class	Precision	Recall	F1-Score	Support
0	0.557	0.570	0.564	179
1	0.298	0.351	0.322	111
2	0.062	0.031	0.042	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.426	333
Macro avg	0.184	0.190	0.186	333
Weighted Accuracy	0.405	0.426	0.414	333

Table 2.5: SVM Classifier - Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.550	0.961	0.699	179
1	0.350	0.063	0.107	111
2	0.000	0.000	0.000	32
3	0.000	0.000	0.000	10
4	0.000	0.000	0.000	1
Accuracy			0.538	333
Macro avg	0.180	0.205	0.161	333
Weighted Accuracy	0.412	0.538	0.411	333

Table 2.6: RFC - Classification Report

By looking at the confusion matrix and classification report, we still see how RFC goes strong in classifying class with a F1-Score of `0.70 ~ (0.699)`, differently from the SVM Classifier (`0.564`). The SVM model, though, classifies correctly more samples of class 1-2 instead of the RFC model which recognizes very few samples of class 1 and 0 samples for class 2.

Here we use the `weighted avg` metric, where it's shown how both models perform the same in terms of weighted F1-Score (`0.414 (SVM) / 0.411 (RFC)`).

We now plot the decision surface of the SVM Classifier.

We see that Poly and RBF best show the different classes, while we see a "red" trend for the Sigmoid function. It's interesting looking at the various curves of the Poly function, due to the degree of the polynomial function.

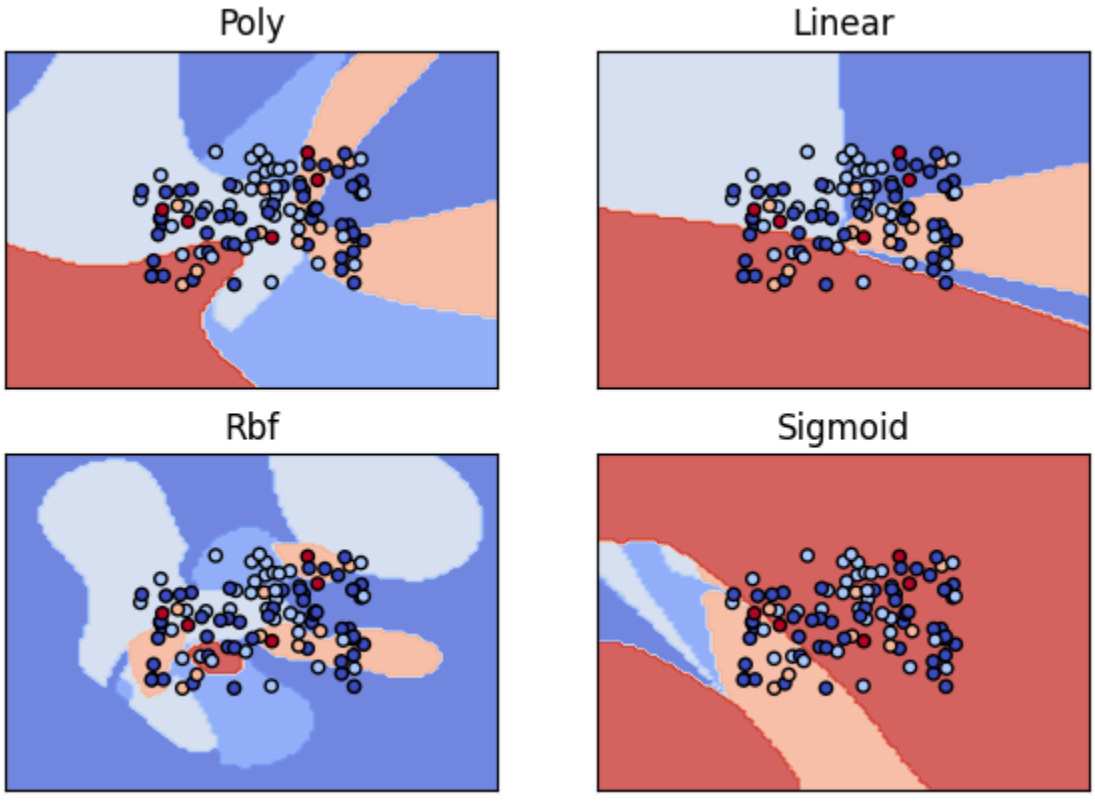

Figure 11: Decision surface for the various kernels of SVM

2.6 Final result comparisons & discussion

Before discussing about overall and final results, we deploy a one last accuracy plot:

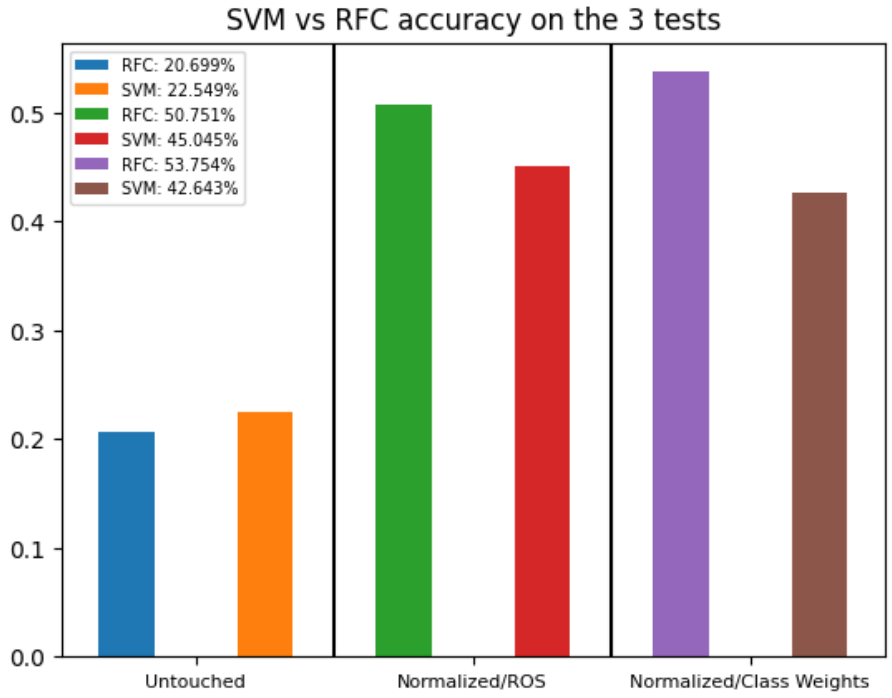

Figure 12: Difference in terms of accuracy of the two models tested on the Test Set over the 3 experiments carried out.

From this final plot, we can see how poor the performances are when using the untouched version of the dataset, while things start to get better in the second and the third experiment. This is because we used the ROS and class weight technique.

We want to highlight the fact that the Class Weight technique performs better than the ROS algorithm due to the +3 points of accuracy on the RandomForestClassifier, since it relies directly on the various weights of the class, without considering random over-sampled samples, since that's the case of ROS, which randomly draws samples along the feature space.

There is no need to oversample the dataset, risking to draw samples which do not really belong to the feature space of a certain class. It's true, though, that ROS has overall good performances.

Chapter 3

Regression

3.1 Dataset

We read and store the .tsv file in a DataFrame as follows:

ID	UAV_1_track	UAV_1_x	UAV_1_y	UAV_1_vx	...	min_CPA
0	0.027	-62300	-59305	6.705	...	1673.734
1	4.023	-17220	47439	-167.65	...	51230.547
2	1.841	-19900	59030	208.71	...	18668.177
3	3.621	-48565	-11986	-113.51	...	10159.624
4	2.318	52665	-47498	177.79	...	22110.623

where we store only the `min_CPA` value, as we are dealing with a Regression problem.

3.2 Pre-processing techniques

Later on, we normalize our X and y set between 0 and 1 through a simple mathematical formula of normalization for every element $x_i \in D$ thanks to the `MinMaxScaler()` function from the `sklearn` library:

$$\frac{x_i - min(set)}{max(set) - min(set)}$$

Now our dataset looks as the following:

ID	UAV_1_track	UAV_1_x	UAV_1_y	UAV_1_vx	...	min_CPA
0	0.003	0.167	0.161	0.508	...	0.025
1	0.640	0.422	0.741	0.165	...	0.884
2	0.292	0.407	0.804	0.907	...	0.320
3	0.576	0.245	0.418	0.271	...	0.172
4	0.368	0.817	0.225	0.846	...	0.379

3.3 First experiments

As we did in Classification, we first try various Regressors in order to check which one overall performs better. In order to evaluate the regressors, we do use the R2-Scoring and the MSE (Mean Squared Error). We test the following regressor methods:

- Stochastic Gradient Descent
- KernelRidge
- ElasticNet
- Bayesian
- Lasso
- GradientBoostingRegressor
- SVR
- KNeighborsRegressor
- VotingClassifier (which includes the previous 8 regressors)

From our tests, the results are the following:

Regressor	**R2-Score**	**MSE**
Stochastic Gradient Descent	-0.00602	0.03610
KernelRidge	-0.04066	0.03735
ElasticNet	-0.00375	0.03602
Bayesian	-0.00165	0.03595
Lasso	-0.00375	0.03602
Gradient Boosting Regressor	-0.08588	0.03897
SVR	**0.11634**	**0.03235**
K-Neighbors	-0.01501	0.03643
VotingRegressor	0.04892	0.03413

Through these sample tests, we can affirm that SVR performs way better than other regressors, so we do stick with it and do other appropriate tests, like hyper-parameters tuning and trying different configurations. It's interesting to see that most of these scores, apart from SVR, return a negative R2-Score, which means that the model predictions are worse than a constant function that always predicts the mean of the data. dd

3.3.1 Scoring

We do know there is no best metric for regression. What is interesting though, is that MSE returns us the average of the squares of the errors (average squared difference between the estimated values and the true values).

This could be taken into consideration, since our true values for y (minimum CPA) could be represented by a line in the space, so we want to approximate this line in the best way possible through points.

So once our model returns us these points, we can use MSE to decide whether it was a good approximation of the minimum CPA or not.

In the end we will evaluate the SVR performances through the MSE metric.

3.4 SVR (1st configuration)

Starting with Support Vector Machine for Regression, we try a first configuration of hyper-parameters such that `C = 1.0` with `kernel = RBF`.

The MSE for this specific model is `0.03235`, with a fit time of just `0.034s`, which is impressively fast.

Here follows a simple representation of the model approximation:

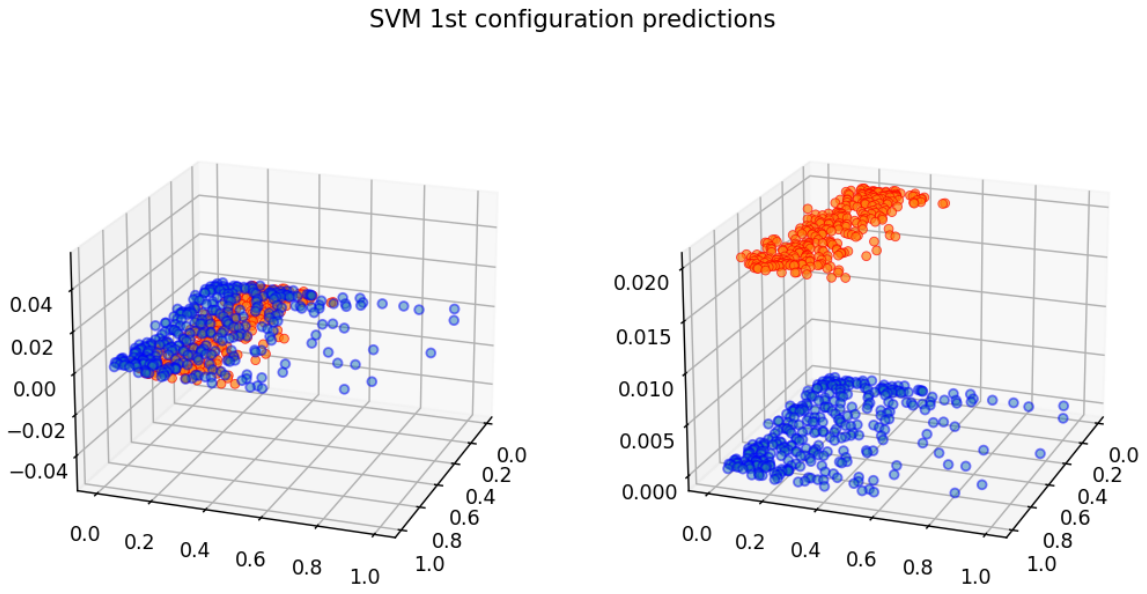

Figure 1: SVR predictions. The left-hand picture shows the actual predictions (red) and the true values (blue). The right-hand picture is only for visualization purposes - basically the same plot but on a different axis in order to have a better view of the predictions.

3.4.1 Further tests

We then run a GridSearch on the model previously explained with the following parameters:

- `'C' : np.arange(0.1,5,0.1)`
- `'gamma':['scale']`
- `'kernel': ('rbf','sigmoid', 'linear')`
- `'epsilon': np.arange(0.1,3,0.1)`
- `CV: 10`

which returns us a set of best hyper-parameters of the model: `C = 0.2`, `Epsilon = 0.1` and `kernel = rbf`. In fact, replacing these parameters to the model we previously validated, results in a very slighly better MSE (`0.03171` vs `0.03235`) which is still a good result.

We then run a K-Fold algorithm on the previous model, receiving an MSE of `0.03518`, which is higher than the one we got in the tests above.

Lastly, we use Bagging and Boosting algorithms with various parameters on the SVR, which do not improve our MSE any further, since we obtain an error of `0.03216` on Boosting and `0.03286` on Bagging.

3.5 SVR (2nd configuration)

We now try a different configuration of the Support Vector Machine for Regression, using initial values for the hyper-parameters such that `C = 1.0`, `degree = 4` and `kernel = Poly`.

The MSE for this specific model is `0.03704`, with a fit time of just `0.063s`. The R2-Score is `-0.03227`, which means that our model can't really explain the relation between data and predictions (and this is the reason why we use the MSE).

Here follows a simple representation of the model approximation:

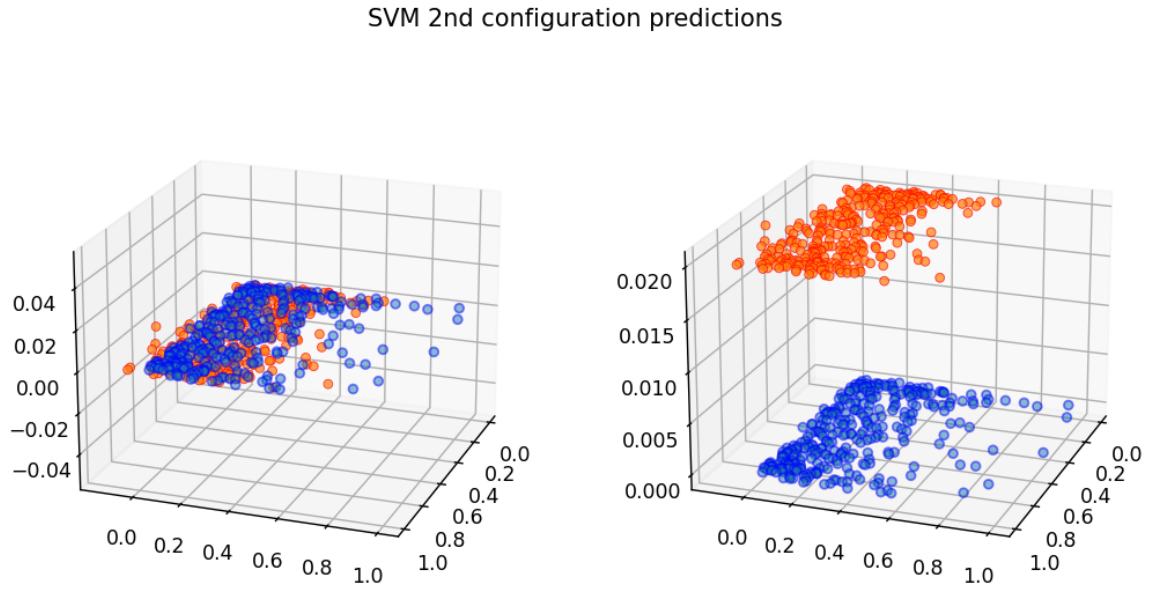

Figure 2: SVR predictions. The left-hand picture shows the actual predictions (red) and the true values (blue). The right-hand picture is only for visualization purposes - basically the same plot but on a different axis in order to have a better view of the predictions.

3.5.1 Further tests

We then run a GridSearch on the model previously explained with the parameters for the 'poly' kernel:

- `'C' : np.arange(0.1,5,0.1)`
- `'gamma':['scale']`
- `'kernel': ['poly]`
- `'degree': np.arange(1,6,1),`
- `'epsilon': np.arange(0.1,3,0.1)`
- `CV: 10`

which returns us a set of best hyper-parameters of the model: `C = 0.1`, `Epsilon = 0.1` and `Degree = 2`. Trying these parameters on the previous model return a better solution, with an MSE of `0.03209`, which is way less than the one we got with the default settings (`0.03704`).

In the end, we use Bagging and Boosting algorithms with various parameters on the SVR. The Boosting algorithm does not seem to improve our model, giving us an error of `0.03506`, while (surprisingly) the Bagging method returns us what seems like the best MSE for this 2-nd configuration: `0.03199`.

This is the final plot of our predictions with the bagging regressor:

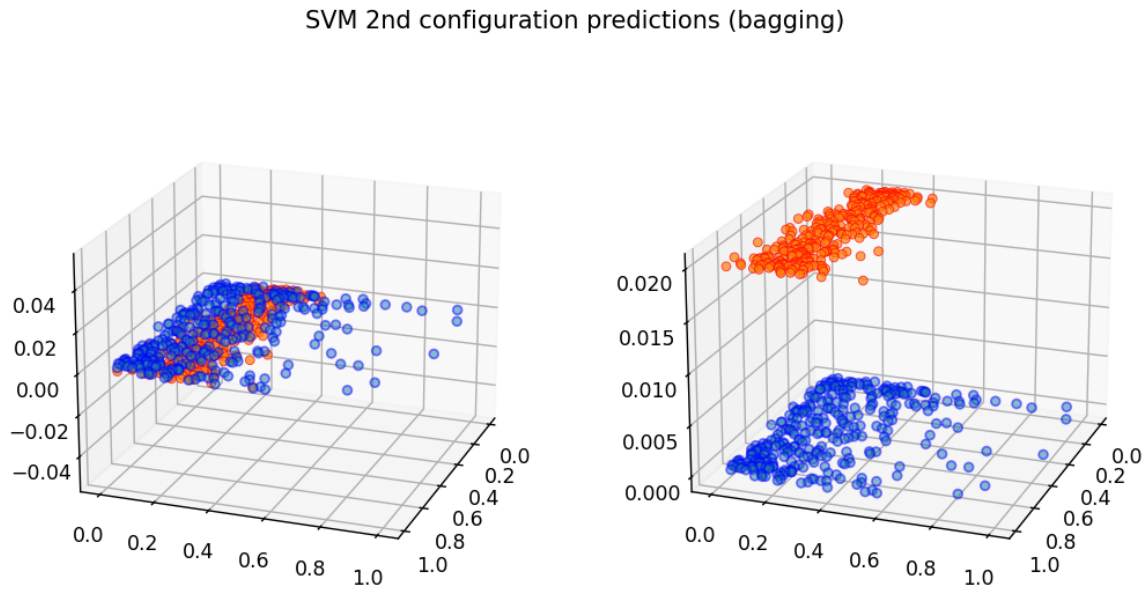

Figure 3: SVR predictions with Bagging. The left-hand picture shows the actual predictions (red) and the true values (blue). The right-hand picture is only for visualization purposes - basically the same plot but on a different axis in order to have a better view of the predictions.

3.6 Result comparisons & discussion

Discussing briefly about results, something curious can be seen in the various test and that is the way that the regression points are plotted inside each graph.

For SVR with `rbf` kernel and SVR with `poly` kernel on a Bagging regressor, we see that the points are all concentrated towards the end, while the SVR with only `poly` kernel seems to better approximate points that are also far away from the central-bottom mass (meaning we have also plots on the right of the 3D space). This is maybe the key-point of how both kernels work and how much they rely on their support vectors (which maximize the margin of the regressor).

Talking about scores and errors, from what we have seen, most of these methods we have tested have more or less the same range of error (MSE) oscillating between `0.03199` and `0.03704`.

There might still be some problems in the dataset though, which indeed bring some overfitting issues. This could be resolved through an exploration of the best-relations between the various features in the .tsv file.

We might also turn this task into a dimensionality reduction, where we reduce the features to the ones that are actually well-connected, hoping to bounce back some overfitting problems.