Multi-UAV conflict risk analysis

Author: **Leonardo Colosi**

SAPIENZA
UNIVERSITÀ DI ROMA

AIRO, ML
December 9, 2022

Contents

1 Introduction

This is the report for the first homework of the Machine Learning course of 2022 at Sapienza University. The goal of the homework is to make predictions and risk analysis over a set of data. The data-set provided for the assignment is a tab separated file (*tsv*) containing information about 5 different UAV. The information, organized as 7 columns for each drone for a total of 35 features, describes a set of UAVs positions and velocities, them orientations with respect to some targets and the targets positions. Additionally there are two output columns for two different problems of classification and regression. The first output column contains integer values, from zero to four, which are the labels for the classification problem, and represent the number of collisions between the 5 drones. The second column contains real values, the output for the regression problem, which represents the *min*-CPA between all the pairs of drones. CPA or Closest Point of Approach is an estimated point in which the distance between two objects, of which at least one is in motion, will reach its minimum value.As one aspect of the assignment was to operate in a domain independent way, all this information are useful to describe the problems to be solved but are not actually relevant for the solution itself. In fact all the given data has been treated just as numbers during the implementation of the ML algorithms. The specific goal of the homework is discussed in the next sub-sectionwhich also describes the limitations of the data-set.

1.1 Assigned tasks

The assignment of the homework, as anticipated, consist into solving two distinct problems:

1. The first is a classification problem of estimating the total number conflicts between UAVs given the provided features in the data-set. Since 5 drones are represented in the data-set there could be from 0 up to 4 collisions for each row of the train-set.

2. The second is a regression problem of predicting the minimum Closest Point of Approach (CPA) [1] among all the possible pairs of UAVs. So to implement a machine learning algorithm that best approximates the *min*-CPA function expressed in the train-set.

For each feature or output column there are 1000 rows plus one the title, so the available data are 37000. Also the data-set is unbalanced, in fact there are just 3 samples for the minority class label 4 and 30 for the second to last class label 3. On the other hand the majority class contains 538 samples. Given the nature of the data-set it is reasonable to assume that a machine learning model trained over it would reach a great performance level. In Section 1 and Section 2 I will outline the strategy that I have adopted to complete these two tasks.

2 Classification

The first problem presented is a multi-class classification problem over the given data-set. As anticipated in the introduction the data-set is widely unbalanced, thus a simple classification strategy would not work very well, leading to an inaccurate model with poor performances. In the next sub-section I will outline the strategy that I have adopted step by step. In the following sub-sections I will discuss the most significant aspect of each step to then show and comment on the experimental results.

2.1 Strategy outline

The strategy that I have chosen to complete this classification task goes as follows:

- Step 1: loading and reading of the data-set with panda;
- Step 2: normalizing and splitting the data-set;
- Step 3: taking a first look at the data;
- Step 4: establishing the evaluation metrics;
- Step 5: testing different model on the raw data-set;
- Step 6: comparing different models;
- Step 7: establishing an oversampling strategy;
- Step 8: testing the best model on new data-set.

Each step has a corresponded *Text* block in the Jupiter Notebook linked to the project.

2.2 Data acquisition

To acquire the data in the first place I have used the built-in function of panda *pd.read_csv* to read the given .tsv file. The data-set is structured such that the first 35 columns contain the sample features while the 36th column contains the output for the classification task and in the 37th column there is the output for the regression task. For this task I have loaded the features in a panda data-fame X and the classification output in a one-dimensional labeled array y.

2.3 Normalization and split

After the acquisition process I have performed a normalization of the features columns. I have done this to transform data in a way that they are either dimensionless and have similar distributions so that no single variable has more weights on the model. Otherwise different scales would not contribute equally to the model fitting and might end up creating a bias. I have used a modified version of *Min-Max* to scale everything between -1 and 1:

$$2 * \frac{x - min(x)}{max(x) - min(x)} - 1.$$

The next step was to split the original data-set to obtain two distinct subsets, one for training and the other for testing. This was achieved by the sikit-learn function *train_test_split*, which allows to specify the relative size of the subset.This is a fundamental aspect since the splitting size does affect the final performance of the model. For this particular problem it is also important to take into account that the size of the minority class is extremely small. This means that, working with the raw[1] data-set, it is important to choose a reasonable split size to avoid the case of having not enough samples of the minority class for training the model, of course this means leaving few samples (actually either two, one or zero) for the testing phase, an inevitable trade off due to the nature of the data-set. To avoid the case of having no samples in the training I have decided to split the data-set in such that the size of the train set is 60% of the total while the size of the train just 40%. I have also used the option *stratify* of the function over y, to keep the proportion between the classes after the splitting.

2.4 Plots of train and test sets

To visualize the classes of the data-set, both train set and test set, I have made a scatter plot with *matplotlib*. Since there are 35 columns in X I have reduced the size just taking the first 2 features to be able to make the 2D plot.

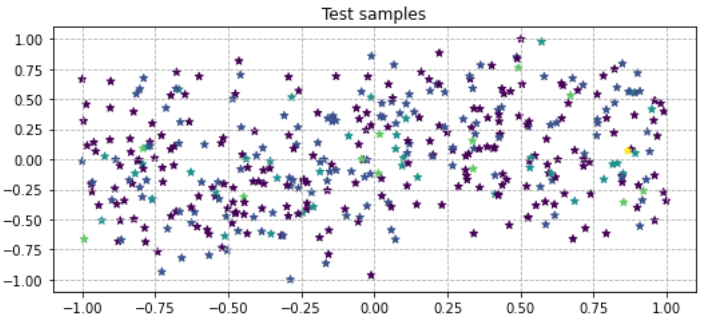


Figure 1: Scatter distribution of reduced X_train set, just the first two features of X are taken in account. The classes are represented with different colours.

[1] In this context I intended 'raw' as an unbalanced on which no data augmentation was performed.

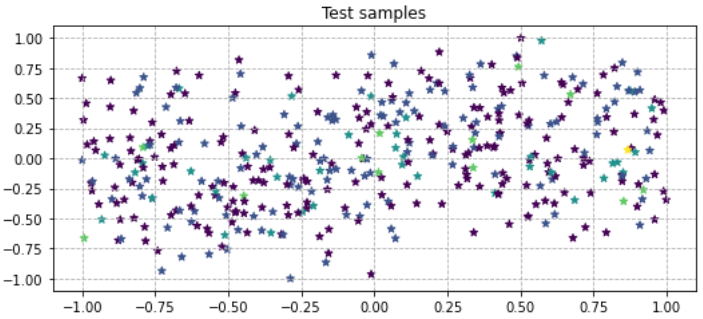


Figure 2: Scatter distribution of reduced X_test set, just the first two features of X are taken in account. The classes are represented with different colours.

2.5 Classification metrics

To evaluate the models performance I have taken in account the results given by different metrics, and then extracted the most useful information according to the problem.

Confusion matrix
A table of test samples with two dimensions where the rows represent the True Label of the samples while the columns represent the prediction made by the model.

Accuracy
Hence the number of correct predictions made by the model over all kinds of predictions made. Given: true positives as TP, False Positives FP and the same for true and false negative we have that:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision
Is a metrics to tell what proportion of samples predicted as positive, actually are true positive.

$$Precision = \frac{TP}{TP + FP}$$

Recall
Is a measure that tells us what proportion of samples that are actually true positive was predicted as positive by the algorithm.

$$Recall = \frac{TP}{TP + FN}$$

F1Score
This score is the harmonic mean between Precision and Recall, so if one of the two is really small between precision and recall, the F1 Score is more closer to the smaller number than the bigger one.

$$F1Score = \frac{2 * Precison * Recall}{Precision + Recall}$$

Evaluation
Accuracy is a good measure when the target variable classes in the data are nearly balanced. Since this is not the case, accuracy alone can be considered a good enough. If, for example, the model is very bad and predict all samples as zero or one the resulting accuracy would still be high since zero and one are dominant classes. On the other hand a combination of precision and recall can be used as a valid metric also in the case of unbalanced data-sets. These two metrics can be combined in one parameter, as anticipated before. So the weighted average of F1-scores for each class represents the most relevant metrics for this problem.

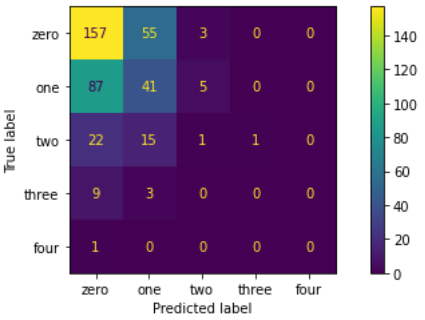
2.6 Models train and fit

After the first phase of preprocessing I have chosen three different models to train and test. Each model has been evaluated according to the metrics expressed in Sub-section 2.5. Also for each model I have performed a grid search with the goal of tuning the parameters and find the most suitable ones. When evaluating different "hyperparameters" for estimators there is a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. To avoid this problem I have applied the k-fold cross-validation procedure where the training set is split into k smaller sets and a model is trained using $K - 1$ of the folds as training data and the resulting model is validated on the remaining part of the data. By default, the score computed at each iteration is the score method of the estimator.

K Nearest Neighbors
First I have tried with the KNN model, setting the parameters for the grid search as:

- n_neighbors : list(range(2,10));
- algorithm: kd_tree ;
- p: list(range(2,5));
- metric : ['minkowski', 'euclidean']

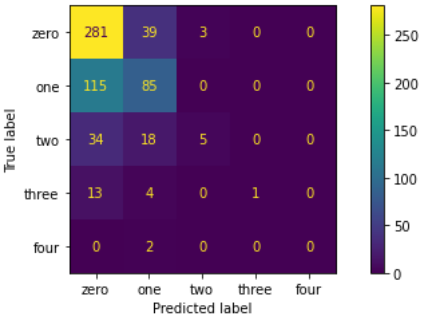
The best parameters found by the search are: algorithm: 'kd_tree', metric: 'minkowski', n_neighbors: 6, p: 2.




```
Test results on the test set:

Classification report for classifier KNeighborsClassifier:

              precision    recall  f1-score   support

           0       0.57      0.73      0.64       215
           1       0.36      0.31      0.33       133
           2       0.11      0.03      0.04        39
           3       0.00      0.00      0.00        12
           4       1.00      0.00      0.00         1

    accuracy                           0.50       400
   macro avg       0.41      0.21      0.20       400
weighted avg       0.44      0.50      0.46       400
```

Figure 3: KNN performances measured on the test set.

KNN overfitting/underfitting check



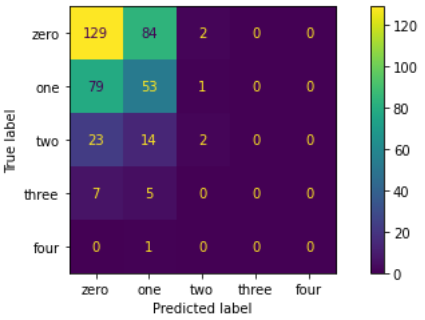

```
Test result on the Train set:


Classification report for classifier KNeighborsClassifier:

              precision    recall  f1-score   support

           0       0.63      0.87      0.73       323
           1       0.57      0.42      0.49       200
           2       0.62      0.09      0.15        57
           3       1.00      0.06      0.11        18
           4       1.00      0.00      0.00         2

    accuracy                           0.62       600
   macro avg       0.77      0.29      0.30       600
weighted avg       0.63      0.62      0.58       600
```

Figure 4: KNN performances measured on the train set.

Decision Tree Classifier
As second model I have chose the decision tree classifier setting the parameters for the grid search as:

- max_depth :[10, 100, 1000, 10000],
- max_leaf_nodes: list(range(2, 50));
- min_samples_split: [2, 3, 4, 5];
- max_features: ['sqrt', 'log2', None]

The best parameters found by the search are: max_depth: 1000, max_features: 'sqrt', max_leaf_nodes: 28, min_samples_split: 2.



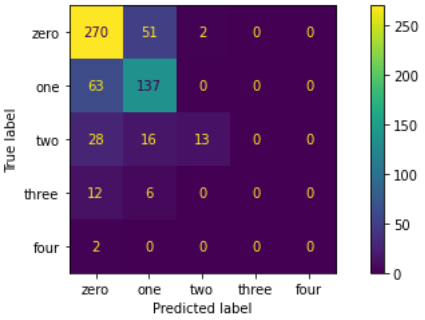
Test results on the test set:

Classification report for classifier DecisionTreeClassifier:

	precision	recall	f1-score	support
0	0.54	0.60	0.57	215
1	0.34	0.40	0.37	133
2	0.40	0.05	0.09	39
3	1.00	0.00	0.00	12
4	1.00	0.00	0.00	1
accuracy			0.46	400
macro avg	0.66	0.21	0.21	400
weighted avg	0.48	0.46	0.44	400

Figure 5: DTC performances measured on the test set.

DTC overfitting/underfitting check



Test result on the Train set:

Classification report for classifier DecisonTreeClassifier:

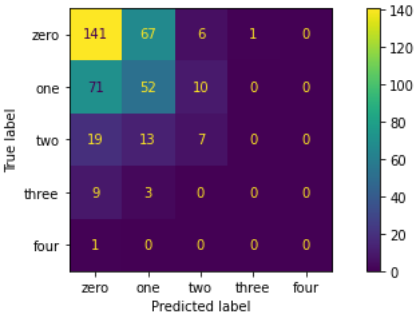
	precision	recall	f1-score	support
0	0.72	0.84	0.77	323
1	0.65	0.69	0.67	200
2	0.87	0.23	0.36	57
3	1.00	0.00	0.00	18
4	1.00	0.00	0.00	2
accuracy			0.70	600
macro avg	0.85	0.35	0.36	600
weighted avg	0.72	0.70	0.67	600

Figure 6: DTC performances measured on the train set.

Support Vector Classifier
Finally as third model I have chose SVC setting the parameters for the grid search as:

- C:[1,10,100];
- gamma:[1,0.1,0.001,0.0001];
- kernel:['rbf', 'linear']

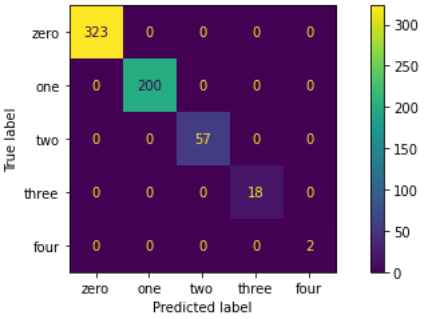
The best parameters found by the search are: C: 10, gamma: 0.1, kernel: 'rbf'.




```
Test results on the test set:

Classification report for classifier SupportVectorsClassifier:

              precision    recall  f1-score   support

           0       0.59      0.66      0.62       215
           1       0.39      0.39      0.39       133
           2       0.30      0.18      0.23        39
           3       0.00      0.00      0.00        12
           4       1.00      0.00      0.00         1

    accuracy                           0.50       400
   macro avg       0.45      0.25      0.25       400
weighted avg       0.47      0.50      0.48       400
```

Figure 7: SVC performances measured on the test set.

SVC overfitting/underfitting check




```
Test result on the Train set:


Classification report for classifier SupportVectorsClassifier:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00       323
           1       1.00      1.00      1.00       200
           2       1.00      1.00      1.00        57
           3       1.00      1.00      1.00        18
           4       1.00      1.00      1.00         2

    accuracy                           1.00       600
   macro avg       1.00      1.00      1.00       600
weighted avg       1.00      1.00      1.00       600
```

Figure 8: SVC performances measured on the train set.

Overfitting/underfitting check: For each model I have performed a prediction over the train set to make some observation about a possible overfitting or underfitting scenario. In case a model prediction over the train is completely aligned with the trai real result it is possible that the model is overfitting the train set. This is not always true but it can be a good estimation of the phenomenon. On the other hand a model that performs poorly also on the train set is almost certainly underfitting.

2.7 Results

The results of the various tests show that none of the three models that I have trained has obtained good performance. The best model among all is SVC since it is able to achieve the meat macro F1-score. The overall values of SVC though are not much better than the one of KNN, which can be considered as a kind of lower limit for performance comparison. On the other hand the Decision Tree Classifiers perform much worse than expected, the only way to overcome the underfitting phenomenon,shown by the poor results of the model even on the train set, was to apply a wide grid search allowing the algorithm to choose between multiple options to tune the parameters. The process of checking for overfitting has also shown that KNN is the most consistent on the train and test set. SVC does perform at the best on the train set and this could be a sign of overfitting but not definitive proof.

2.8 Oversampling strategy

None of the previous models has shown good results, not even after the tuning of the parameters, a possible explanation of this is the state of the data-set. To explore this option I have decided to perform an over sample of the data. Since my goal was not only to obtain new data for the minority class but to balance the whole data-set, I have decided to combine different strategies.

Oversampling
First of all I have performed a naive *RandomOverSampling* to generate duplicate samples of the minority class. I have also duplicated the samples in the other two least populated classes of the same factor, to maintain the original distribution.

Undersampling
After that I have applied a *RandomUnderSampling* to reduce the elements of the majority classes and avoid overfitting. As well as the augmentation also the reduction was done proportionally, with the same idea as above.

SMOTE
Finally I have applied the *SOMTE* function to balance all the data-set. The key point here is that *SMOTE* (Synthetic Minority Oversampling Technique) is able to generate new synthetic samples via interpolation, this should guarantee the creation of reasonable elements to extend our data-set. Since *SMOTE* uses interpolation it needs starting data to perform, that is why I have applied the *RandomOverSampling* before. Of course this technique presents some limitations, in fact the new data, for the minority classes, came from an interpolation over a set of duplicated data. This can be seen by the fact that the model does not show a significant improvement after the Oversampling process.

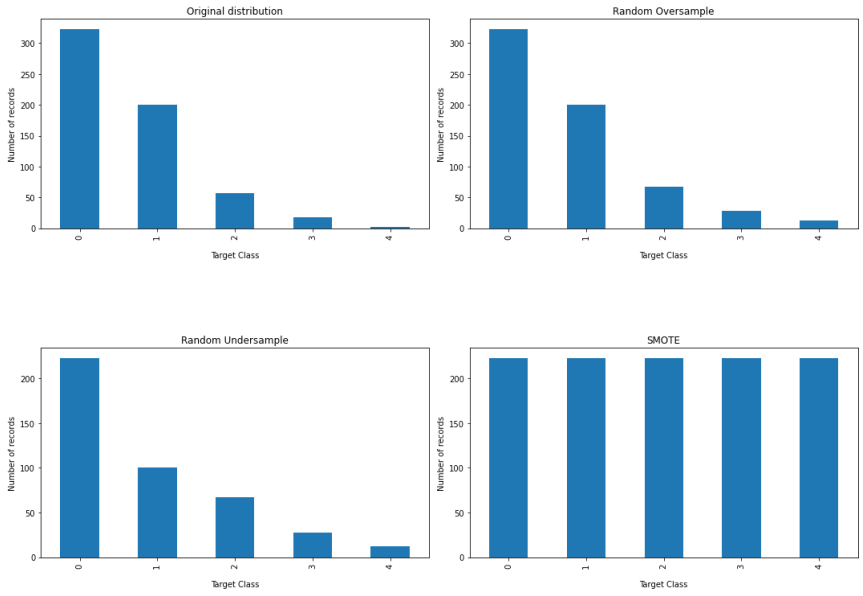
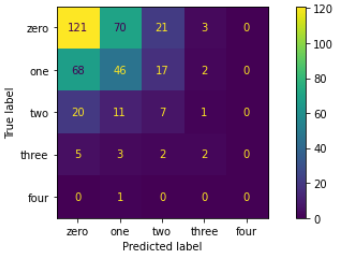


Figure 9: Evolution of the data-set during the oversampling process.

2.9 Balanced data-set results

After the balancing process I have executed another test, over the new data-set, using the best among the three models tested before, SVC. So I have performed another grid search to tune the parameter to the new data-set. SVC works better



Test results on the test set:

Classification report for classifier SupportVectorsClassifier:

	precision	recall	f1-score	support
0	0.57	0.56	0.56	215
1	0.35	0.35	0.35	133
2	0.15	0.18	0.16	39
3	0.25	0.17	0.20	12
4	1.00	0.00	0.00	1
accuracy			0.44	400
macro avg	0.46	0.25	0.26	400
weighted avg	0.45	0.44	0.44	400

Figure 10: Evolution of the data-set during the oversampling process.

on the balanced data-set, achieving for example a best F1_score on the class label 3. It does in fact achieve a better macro F1_score but it also loses some accuracy over the majority class. A trade off that could be due to the undersampling step. In general this oversampling technique does not compensate for the missing data.

3 Regression

About the regression problem, of best approximating the *min*-CPA between all the possible pairs of drones, some of the steps of the solution process are similar to the ones in the previous Section. Here again I will outline the main step of my resolution approach, focusing mainly on the difference due to the nature of the regression problem. So I will skip steps such as the data acquisition or the split of the data-set.

3.1 Strategy outline

The strategy that I have chosen to complete this regression task goes as follows:

- Step 1: normalizing and taking a first look at the data;
- Step 2: establishing the evaluation metrics;
- Step 3: testing different model on the data-set;
- Step 4: comparing different models.

Each step has a corresponded *Text* block in the Jupiter Notebook linked to the project.

3.2 Normalization and visualization

In this step I have used the same principle and technique of normalization described in Sub-section 2.3. The only difference is that for this problem I have also normalized the output column since it contains real values (expressed in meters) and not just integers (labels).
To visualize the data I have decided to focus on the output column *min_cpa*, before the splitting of the data-set. This is the 37-th column of the provided *tsv* file. After loading this column in the array y I have plotted the output distribution.

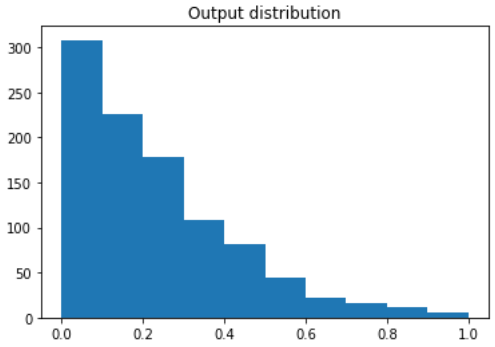


Figure 11: Distribution of the regression output generate over the whole normalized data-set

3.3 Regression metrics

Similarly to the classification problem, here I have taken in account the results given by different metrics. The ones that I have used are described below, outlining the different meaning of each metric in the context of the problem.

Mean Squared Error
The aim of MSE is to minimize the mean squared error between predictions and expected values. It is calculated as the mean or average of the squared differences between predicted and expected target values in a data-set. A larger MSE indicates that the data points are dispersed widely around its central mean, whereas a smaller MSE suggests the opposite.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i + \hat{Y}_i)^2$$

Where n = *number of data points*, Y_i = *observed values* and $\hat{Y}_i$ = *predicted values*.

R2 score
In the general case when the true y is non-constant, a constant model that always predicts the average y disregarding the input features would get a R2 score of 0. So these metrics can be interpreted as a comparison between our model an a constant model.

$$R^2 = 1 - \frac{RSS}{TSS}$$

Where R^2 = coefficient of determination, RSS = sum of squares of residuals and TSS = total sum of squares.

3.4 Models train, fit and comparison

For the regression problem I have used a strategy similar to the one used for the classification task. After choosing two models, and SVR I have made a series of tests to compare the performance of the two algorithms. In the case of SVR I have also used a grid search to tune the hyperparameters, while for the Linear Regression I have used the naive function. For the grid search in this case I have setted the evaluation metrics for the cross validation to be the R2 score. The following are the outputs of the two models on the test sets.

LinearRegression
This is the output of the Linear Regression model. It does not perform well in terms of R2 score but it do achieve a low MSE. The negative R2 score indicate that the model perform a bit worst than a constant model.

```
Model: LinearRegression()

Model in training...

Done

Test results on the test set:
Mean squared error: 0.04
Regression R2score: -0.021
Regression model score: -0.021
```

Figure 12: LinearRegression()

SVR
After Linear Regression I have tested the SVR model. In this case i have performed a grid search, choosing as parameters:

- kernel:['rbf', 'poly'];
- C:[10, 1, 0.1, 0.01];
- degree:[1, 2, 3, 4, 5];

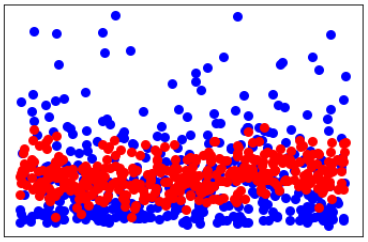
The best parameters found by the search are: C: 10, degree: 1, kernel: 'rbf'. This model does archive a positive R2 score while keeping the MSE at a low value.

```
Model: SVR()

Grid search execution...

Model in training...

Fitting 3 folds for each of 30 candidates, totalling 90 fits

SVR() best parameters: {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

Done

Test results on the test set:
Mean squared error: 0.04
Regression R2score: 0.063
Regression model score: 0.063
```

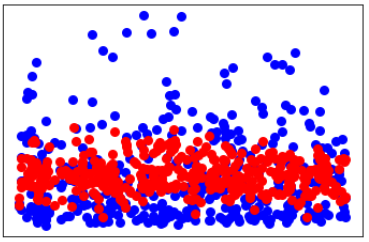
Figure 13: SVR()

3.5 Results

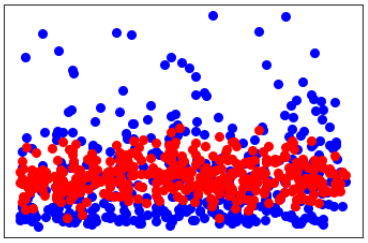
To get an idea of how the best model is approximating the real function I have tried to plot a reduced version of the data-set considering just one feature for the x-axis and the output for the y-axis. To avoid an oversimplification I have iterated this process for all the 35 features obtaining similar results. Here I'm reporting just five graphs as an example, the other 30 can be obtained by executing the code. As can be seen in 14 the model does not work very well with the outliers but since the majority of the points are concentrated in a limited area the MSE is not high. This also explains the R2 value near to zero, comparing the obtained model with a constant one that always predicts the expected value of the output distribution.



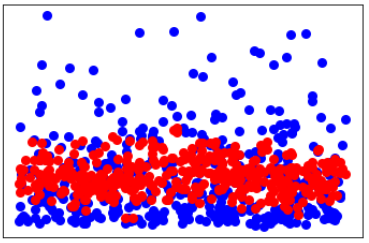
(a) UAV1



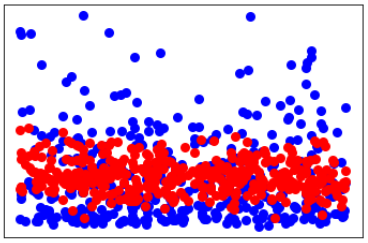
(b) UAV2



(c) UAV3



(d) UAV4



(e) UAV5

Figure 14: 5 representation of the true function (blue) and the model prediction (red). In all the plots the x-component of each point is the feature X_i_track for the i-th drone.