TOML - Projects 3

Oriol Martínez Acón

June 2022

1 Introduction

The main goal for this project is to calibrate an air pollution sensor, low cost sensor, in an air pollution monitoring sensor network. The data sample is obtained from an Internet of Things (IoT) node that accomodates an array of sensors. The data captured can be classified as:

- Topospheric ozone, O_3, which have the measurement units in kΩ.
- Nitrogen dioxide, NO_2.
- Sulfur dioxide, SO_2.
- Nitrogen monoxide, NO.
- Datetime for all the captured data.
- Ambient temperature, which have the measurement units in $^{\circ}C$.
- Relative humidity, which have the measurement units in %.
- Reference station for the O_3 measurements, which have the measurement units in $\mu gr/m^3$, high cost sensor. The reference station will be the true data for our all Machine Learning (ML) models to calibrate the low cost sensor.

In this project we are going to use the power of ML methods to calibrate the O_3 sensor. ML is really useful for prediction if we know the parameters to take into account and we do not have a lot of data; otherwise, if we do not know which are the parameters to take into account for generating the model because it is too complex maybe is better to apply Deep Learning (DL), but in that case we would need huge number of data samples.
All the code developed in this project can be found in the following GitHub repository: `https://github.com/oriolmartinezac/TOML-Labs/tree/main/project-3`

2 Observing the data

Before applying the ML methods to create our models to calibrate the sensor is really important to understand see what is the data and how is its behaviour.
In this section some plots were done to take a look to the data.
The firsts plot represents the data from Sensor O_3 (low cost sensor) and the data from the RefSt (high cost sensor) against the time (datetime).

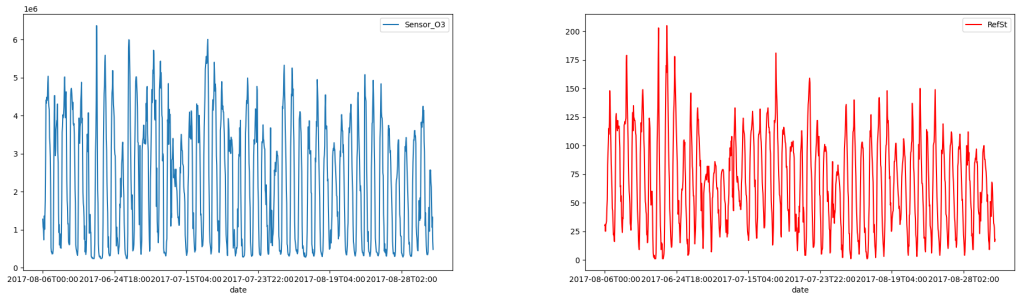


Figure 1: Plots of the O_3 sensor data (blue) and O_3 reference station data (red) against the time.

As we can see in the figure 1 both plots have a lot of similarities, it follows the same pattern but with only few differences (errors in the measures). That means that if we take the sensor O_3 data as a parameter it will have a really big impact in the model created.
The following scatter plots depicts the data captured from sensor O_3 as x-axis with the data captured from the Reference Station as y-axis.

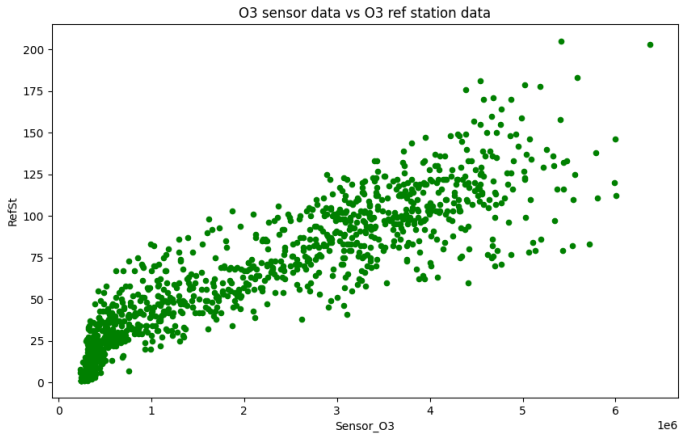


Figure 2: Scatter plot of the O_3 sensor data against the O_3 reference station data.

As we can see in the figure 2 there is a linear independence with both variables and just only some dispersion (errors) due to the precision of the low cost sensor.

Sometimes is better to normalize, all the elements lie between 0 and 1, or standardize the data before doing anything. Normalization is essential when there are some different features with big different ranges. That happens because if we create a linear regression ML model, the feature with big ranges will influence more than the other features, what it translates into a bad model because is not weighting correctly the parameters for the prediction. The way to normalize the data is done by using the mean and the standard deviation. In the following equation we can see exactly how exactly works:

$$\bar{x}_{sensor_j} = \frac{x_{sensor_j - \mu_{sensor}}}{\sigma_{sensor}}$$

In the following figure we can see how it looks the plot now when we normalize the data.

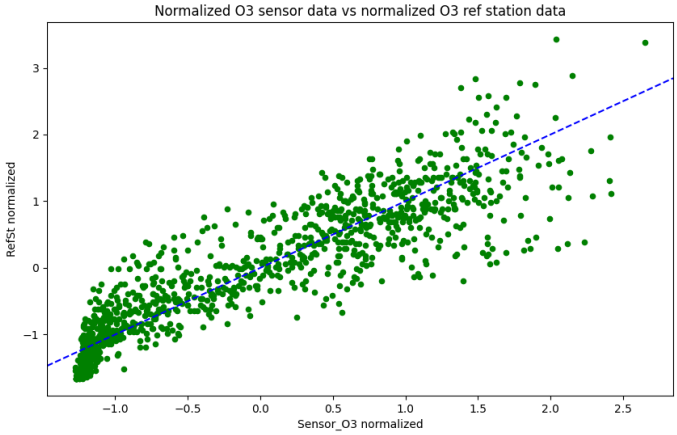


Figure 3: Scatter plot of the normalized O_3 sensor data against the normalized O_3 reference station data.

Even if the data is normalized we can see that in the figure 3 there is no difference between the 2. Although there is no difference it is better to normalize the data and have the same scale for both of the measurements, because they are working in different magnitudes. We can also see in how the data looks linear with a tangent of 45 degrees (blue line).
In the following figures we can see all the plots of the sensor O_3 against all the different metrics and the same for the Reference Station. This could be really useful to see the impact of the ozone for each of the metrics.

The following figures show the plots of O_3 from the low cost sensor and from the reference station against the temperature.

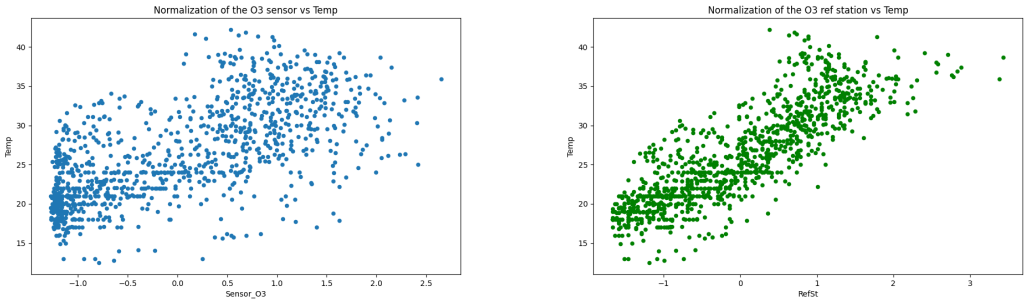


Figure 4: Plots of the O_3 sensor data (left) and O_3 reference station data (right) against the temperature.

In the figure 4 we can see a very defined linear relation between the O_3 and the temperature. That could be translatable to that the amount O_3 depends in a way by the temperature in the ambient.

The following figures depict the plots of O_3 from the low cost sensor and from the reference station against the relative humidity.

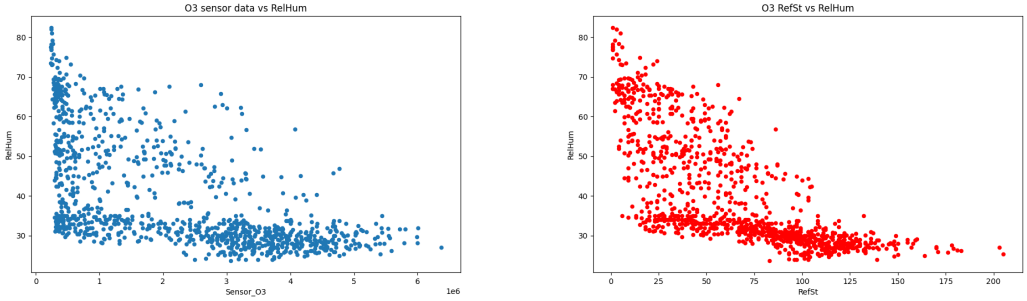


Figure 5: Plots of the O_3 sensor data (blue) and O_3 reference station data (red) against the relative humidity.

In the figure 5 we can see that there is a relation non-linear between the O_3 and the relative humidity, is not quite defined but appreciable. This means that the amount of relative humidity in the environment affects in a way, slightly, the amount of O_3.

The following figures show the plots of O_3 from the low cost sensor and from the reference station against the NO_2 sensor data.

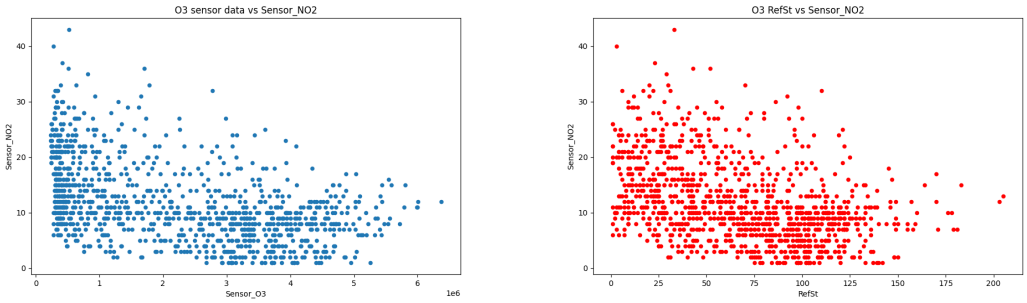


Figure 6: Plots of the O_3 sensor data (blue) and O_3 reference station data (red) against the NO2 sensor.

In the figure 6 is not a visibly defined linear relation between the O_3 and NO_2. But it seems there is a correlation.

The following figures depict the plots of O_3 from the low cost sensor and from the reference station against the NO sensor data.

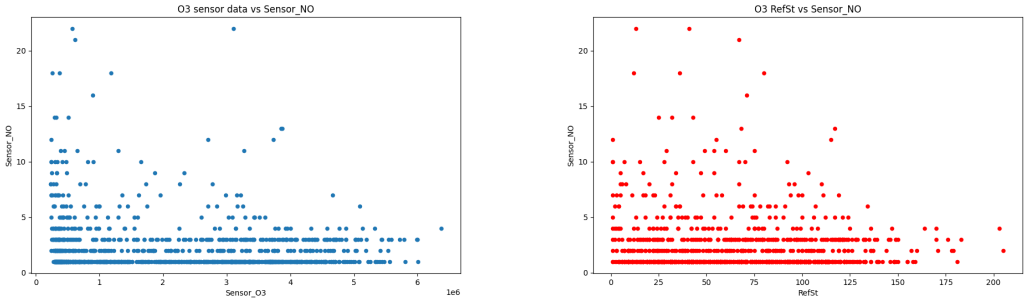


Figure 7: Plots of the O_3 sensor data (blue) and O_3 reference station data (red) against the NO sensor.

As we can see in the figure **??** there is not visibly correlation between the data.

Finally, the following figures show the plots of O_3 from the low cost sensor and from the reference station against the O_2 sensor data

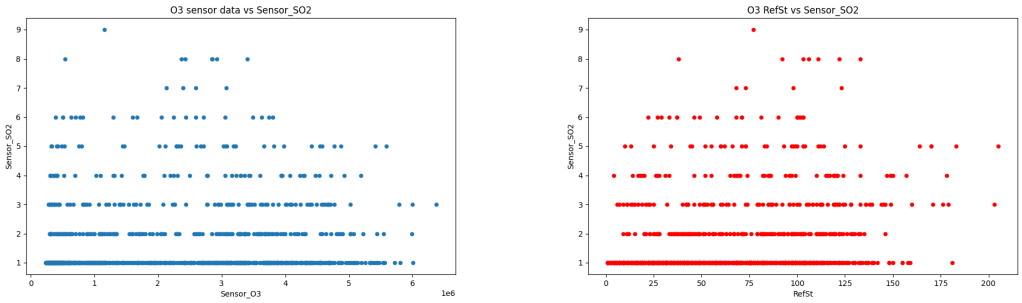


Figure 8: Plots of the O_3 sensor data (blue) and O_3 reference station data (red) against the SO2 sensor.

As in the previous figure, the figure 8 seems that does not have a correlation with the data.

Before doing the ML models, computing the **mean** for each of the **metrics** could be handy. Below we can see the resulting table of computed means per parameter.

Metric	**Mean**
Sensor_O3	2227378
RefSt	67.59229
Temp	26.21402
RelHum	39.65689
Sensor_NO2	11.79798
Sensor_NO	2.317723
Sensor_SO2	1.755739

Table 1: Table with the mean for each of the metrics.

A good way to see with numbers everything said before is by making the **Pearson correlation** between the data. Below we can see the plot of **correlation heatmap** with all the data, this heatmap is the same as the covariance matrix (Σ). Is important to highlight that in the heatmap of correlations all the number oscillates between 1 and -1 (included).

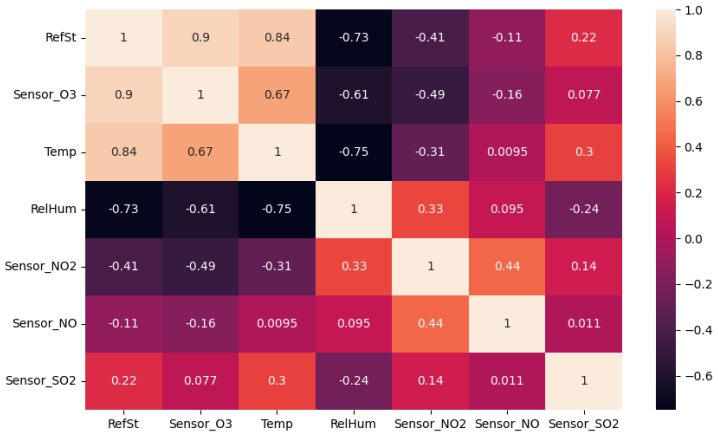


Figure 9: Plot of the heatmap correlation matrix with all the metrics from the sensors.

As we can see in the figure 9 it corroborates what we have said, the metrics that has bigger impact on RefSt are: sensor O_3 (0.9), ambient temperature (0.84) and the relative humidity (-0.73). The positive numbers mean that how much the correlation is directly proportional while negative numbers mean that how much the correlation is inversely proportional, then a 0 value means there is not correlation.

3 Calibrating the sensor

Now that we have analyzed the data that we are going to use, we will use different ML methods to calibrate the sensor O_3. At the end, all these different methods do the same but with different approaches.
In the Machine Learning algorithms we can see various classifications but we are going to take a look to these two:

- **Supervised learning**: It uses labeled datasets, information know previously, to train algorithms to classify or predict outcomes accurately. The model is adjusting the weights by the data given until it fits appropriately, as part of the cross validation process. **In this lab project we are going to use supervised learning algorithms**.
- **Unsupervised learning**: It learns the pattern by the not tagged data given. Instead the model itself find the hidden patters and insights from the given data. Some algorithms: K-means clustering, Hierarchical clustering.

It is also important to take a look to the way of how to calculate the distances. The computation of distances is an important thing in Machine Learning for the model creation, normally minimizing the distances. Here are some ways to calculate the distances between two points:

- **Hamming distance**: Calculates the distance between two binary vectors.

- **Euclidean distance**: Calculates the distance between two real-valued vectors. It represents a right triangle between the two points, where the minimum distance is its hypotenuse. It can be represented like the following equation:

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

(in Cartesian coordinates). **This one plays an important role in the algorithms that we are going to use in ML**.

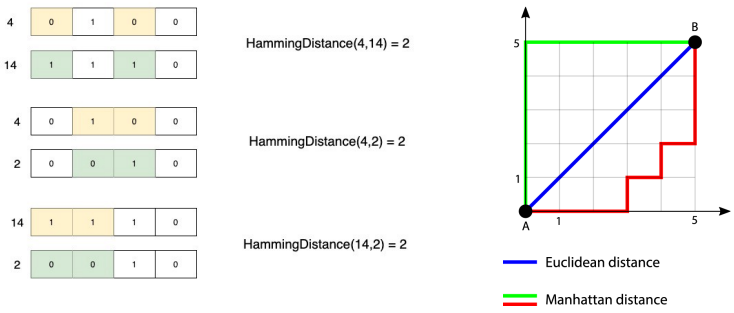
- **Manhattan distance**: Calculates the distance between two real-valued vectors. It represents a right triangle between the two points, where the distances to compute is the sum of the legs in form of blocks of the right angle. It can be represented like the following equation:

$$D_m(p,q) = \sum_{i=1}^{n} |p_i q_i|$$

(which is its L1 norm). **This one plays an important role in the algorithms that we are going to use in ML**.

- **Minkowski distance**: Calculates the distance between two real-valued vectors. It is a generalization of the Euclidean and Manhattan distance measures with a parameter called *order* or p.

In the figure 10 we can see the graphic representation of all the distances commented previously.



(a) Hamming distance

(b) Euclidean and Manhattan distances.

Figure 10: Different distances.

In our problem statement, we call t, which are the values we want to predict or fit, the **dependent variable**. In our case, we want to approximate t to the true values, which is the Reference Station data. The **independent variables** or **regressors** are

- x_1, the values of the O_3 sensor ($\mu gr/m^3$).
- x_2, the values of the temperature sensor ($^{\text{o}}$C).
- x_3, the values of the relative humidity sensor (%).
- x_4, the values of the NO_2 sensor.

- x_5, the values of the NO sensor.
- x_6, the values of the SO_2 sensor.

We assume a linear independence $f(x, \beta)$ such that

$$t \sim f(x, \beta) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \varepsilon$$

The idea of the model is to find the $\beta_0, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5$ and β_6, weights/coefficients of the features, given that we know t, true data, x_1, x_2, x_3, x_4, x_5 and x_6 that minimizes the loss function, and so, that will mean the model is fitting correctly and then all the predictions are going to be very accurately.

3.1 Structure and importance of Data

The data plays an important role to create the perfect model. More data has always a positive feedback because it gives more information to the model which it can use in order to learn, to test or to validate. Not having enough data or valid data can lead into some problems that we will discuss after in the following sections. Before starting to create the model is really important to split the data in different sets. These sets are:

- **Training set**: It contains the data necessary to train the model. Usually it is in range of 70% to 80% of the total data given to give good results.
- **Test set**: It contains the data to test the model trained. Usually it is a 10% of the total data given to give good results.
- **Validation set**: It contains the data necessary to tune the hyper-parameters of each model. Usually it is a 10% of the total data given to give good results.

The data used in this lab project has the size of $N = 1088$ and the number of features are $P = 7$. All the data used for the models have been normalized.

3.2 Cross-validation

Cross-validation is a re-sampling procedure used in the evaluation of a machine learning model on a limited data sample. It basically used in applied machine learning to estimate the skill of a model on unseen data.
This method has become popular because it is simple to understand it gives, generally, results in a less biased or less optimistic estimate of the model than other methods, like with train/test split.
K-Fold cross-validation consists into generate k number of groups by given data sample.
The steps that follows the method generally are:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups.
3. For each unique group:
   (a) Take the gorup as a hold out or test data set.
   (b) Take the remaining groups as a training set.
   (c) Fit a model on the training set and evaluate it on the test set.

(d) Retain the evaluation score and discard the model.

4. Summarize the skill of the model using the sample of model evaluation scores.

In the following figure we can also see how the method works.

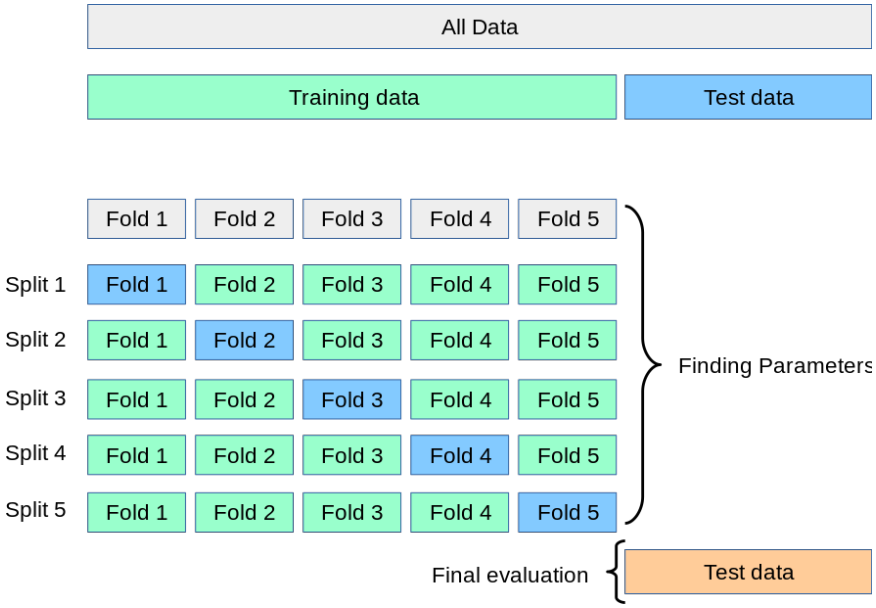


Figure 11: Example of 5-fold cross validation.

3.3 Loss functions and Empirical risk

The **loss functions** are really useful to verify how much good is our model approximating t. These can be classified in classification loss functions or regression loss functions. Some examples of regression loss functions are:

- **L-1 norm (Mean Absolute loss function)**.

  $f(x,\beta)$ is a scalar $\rightarrow l(t, f(x,\beta)) = |f(x,\beta) - t| f(x,\beta)$ is a vector $\rightarrow l(t, f(x,\beta)) = ||f(x,\beta)||_1$

- **L-2 norm (Quadratic loss function)**.

  $$f(x,\beta) \text{ is a scalar} \rightarrow l(t, f(x,\beta)) = (f(x,\beta) - t)^2$$

  $$f(x,\beta) \text{ is a vector} \rightarrow l(t, f(x,\beta)) = ||f(x,\beta)||_2^2$$

- **Huber loss**.

  $$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

The idea is that by obtaining an estimation ($\hat{f}$) of the function f as the function that minimizes the empirical risk of the training set.

Empirical risk minimization is a principle in statistical learning theory which defines a family of learning algorithms and is used to give theoretical bounds on their performance. The core idea is that we can not know exactly how an algorithm will perform in terms of goodness in practice (true risk) because we do not know the true distribution of data given to the algorithm; instead of that, we can measure the performance on a known set of training data (empirical risk). In other words, is the average loss over the data points. The following equation corresponds to the empiriccal risk function:

$$J(\beta) = R_{\text{emp}}(w) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{f}(x_i, w))$$

In the case of the quadratic function instead of using $\frac{1}{N}$ we will use $\frac{1}{2N}$.

3.4 Bias-Variance trade-off

Before getting into the ML methods is really important to define the concepts of **Bias** and **Variance**, and how can they lead into an **underfitting** or **overfitting** scenario

- **Bias**: The error between the average model prediction and the ground truth. This is concept is related directly on the training data set of our model.
- **Variance**: Is the variability in the model prediction-how much the ML function can adjust depending on the given data set.

So, having a low bias means that our model understands perfectly the relationship of our data set (training set). And having a low variance means that our model can performs good in terms of predicting different data, not training set. The important conclusion we can take are: if our model has low bias and high variance means that is **overfitting** and if it is just the opposite, high bias low variance, means that our model is **underfitting**. The idea behind Machine Learning is trying to fit perfectly the data relationship, so there is a trade-off between Bias and Variance, where both are low.
In the following figure 12 we can see a graphic representation of what is have been said.

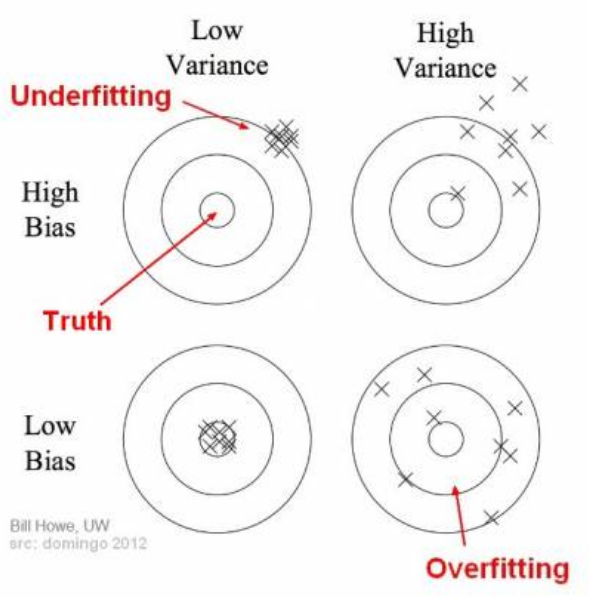


Figure 12: Bias and variance using bulls-eye diagram.

But what is **overfitting** and **underfitting**? In the following section they will be explained.

3.5 Underfitting, Overfitting and Good fitting

When a model is **underfitting** means that it can not understand the true relationship with the data given and then the predictions done are not quite good in accuracy terms. That could happen because the model needs more complexity, a function with more coefficients (weights), to understand the behaviour of the data given (training set), or the model does not have enough data to create a good model to predict.
Overfitting on the other hand, means that the model is understanding perfectly the behaviour of the data given (training data) but, if that model is using different data from the training one it will give bad results as it is not exactly the same data used to train the model (high variance). That could happen because the model is too complex for the amount of data given.
Is important to remark that in a case which you only have one point in a space, all your models would overfit because there is not enough data to understand the behaviour of the data, even though the **Ridge regression** can be very useful in this situation to low the variance of the model.
In the following figure we can see a graphic representation of when a model is underfitting, overfitting or making good fitting.

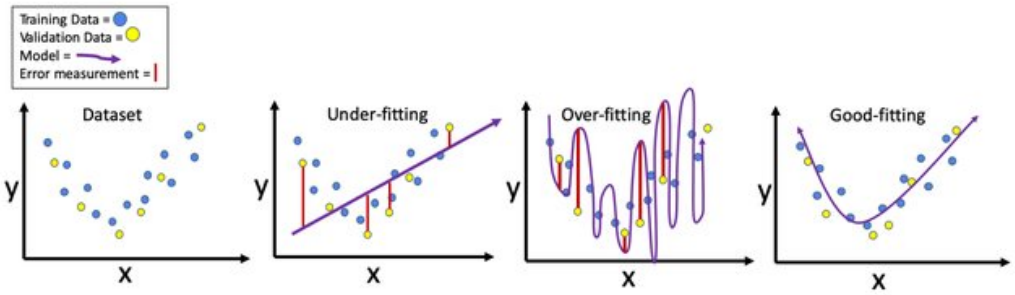


Figure 13: Example of underfitting, overfitting and good fitting models.

3.6 Metrics

A way to measure the error of a model in predicting and then evaluating the accuracy of the model is by some different metrics. These are the metrics used in this lab project:

- **Mean Absolute Error** (MAE): Is a metric that measures the average of the absolute difference the actual and predicted values in the dataset.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

- **Root Mean Squared Error** (RMSE): Is a metric that measures the average of the squared differences between the original and predicted values in the data set.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

- **R-squared** (R^2): Is a metric that represents the proportion of the variance in the dependent variable which is explained by the linear regression model. The possible values are in range of 0 and 1.

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

Having a lower values for MAE and RMSE means that the regression model created has good accuracy in predicting the data. And having a high value for R-squared means that the variables chosen for the model has keeps a good proportion between them and it explains very well the behaviour. The idea is to have low MAE/RMSE and high value of R-squared.

3.7 Multiple Linear Regression

Multiple Linear Regression (MLR) is a method to find a model to predict the data. Multiple Linear Regression act as traditional linear regression but taking into account more dimension/features. Is important to remind that the model created is linear and it can be underfitting if the behaviour of the data is non-linear. The idea of the method is to find the β coefficients that minimizes the distance between the points used to create the model (training points). The coefficients represent the weights of each of the parameters/features in the space.
Generally, the MLR uses the least-square error to approximate the problem. Then if we take a look the optimization problem to solve is the following one:

$$\begin{aligned} &\text{minimize} \quad J(\beta) = \frac{1}{2N}||X\beta - t||_2^2 \\ &\text{var} \qquad\quad \beta \in \mathbb{R} \end{aligned}$$

3.7.1 Forward subset selection

The **Forward Subset Selection** is a method that creates different subsets and select the features that has more impact and relevance among others. Forward subset selection is not the only method to apply the subset selection; however, in this project we are going to use the forward subset selection to see which are the features that has more impact on the models that we want to create. This is really useful if we want to create a model that does not overfits due the low complexity (less unnecessary features). Although, we should be careful in the number of features that we are taking, in other case, we may be underfitting.
Before implementing the forward subset selection, we saw that the best features to take into account to create the model should be: Sensor O_3, Temp and RelHum. Then the results of the method should be the same if we ask for the best 3 possible features.
The steps that follows the method are the following ones:

1. Train n model using each feature (n) individually and check the performance.
2. Choose the variable which gives the best performance.
3. Repeat the process and add one variable at a time.
4. Variable producing the highest improvement is retained.
5. Repeat the entire process until there is no significant improvement in the model's performance.

Since we do not have to tune any hyper-parameter, the data was split into 75% and 25% to training set and test set respectively.
To implement the Forward subset selection we have used the python package called `SequentialFeatureSelection` from the `mlextend` package. The code will give us the best k features to create the models, k is a parameter of the function that we specify. In our case we have chosen 3 as we want to see the same best features talked in the previous sections. In the figure below we can see the best 3 features and the metrics got from them.

```
Best features ['Sensor_O3', 'Temp', 'RelHum']
BEST SUBSET PREDICTION
R^2:  0.9147709763179087
RMSE:  0.29280365799454694
MAE:  0.2258928051914868
```

Figure 14: Results got from applying the forward subset selection method.

In the figure 14 we can see that as we thought, the best 3 features to create the different models are the Sensor O_3, Temp and RelHum.
If now we create a new model by only using the best features got from the forwarded subset selection, we will that the results obtained seems pretty good. The image below depicts the MLR model (blue) against the RefSt (red, true value).

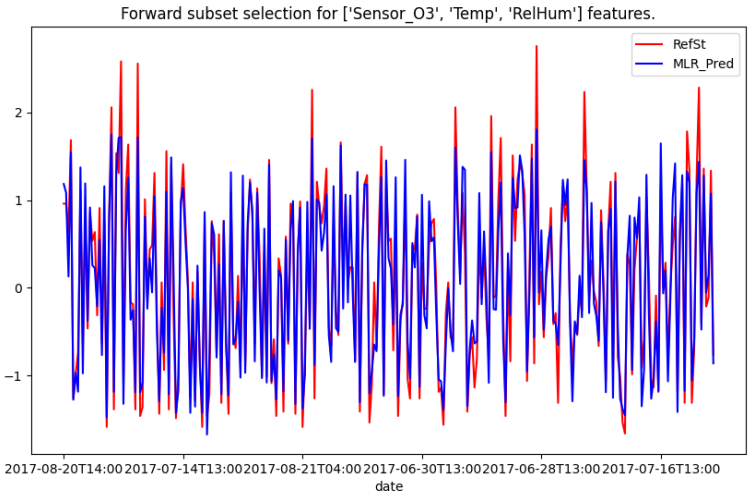


Figure 15: Prediction with best 3 features from forward subset selection.

In the figure 15 we can see that the MLR model generated predicts really good the values against the RefSt.
If we use the `seaborn` python package to plot the results from the MLR predictions (y-axis) against the RefSt (x-axis) we get the following figure.

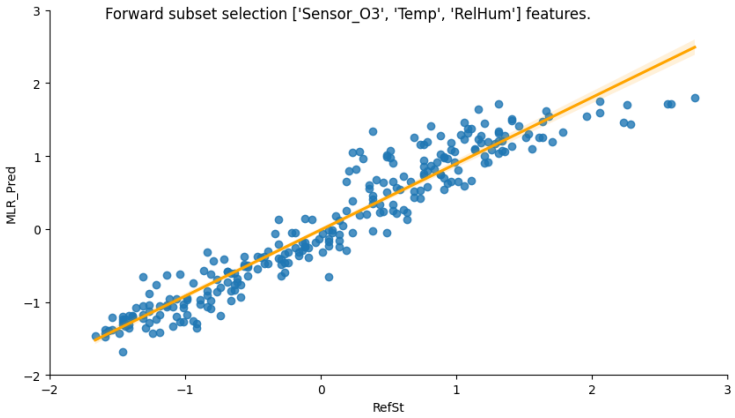


Figure 16: Seaborn plot of the MLR predictions against RefSt.

As we can see in the figure 16 the results seem consistent with the linear model.

3.7.2 MLR with regularization

When we are talking about the ridge regression and lasso regression we are applying a regularization in our model.
Regularization is a method to subtract the complexity and then avoid the possible overfitting of the model created and then, lowering the variance of the model. The way to do the regularization is by adding a hyper-parameter called λ that adds penalization in the slope of the prediction function of the model. In other words, the penalization is paying a small biased to get less variance. Following we can see in form of equation the regularization:

$$J(\beta) = \frac{1}{2N}||f(x,\beta) - t||_2^2 + \frac{\lambda}{2}||\beta||_q^2$$

Where:

- when $q = 1$, it is called Lasso regression (L1-norm).
- when $q = 2$, it is called Ridge regression (L2-norm).

The value of q determines also the shape in the space. For values:

- $q < 1$, not convex shape.
- $q = 1$, diamond shape.
- $q = 2$, circle shape.
- $q = \infty$, square shape.

In the figure 17 we can see the shape of L by changing the values of q.

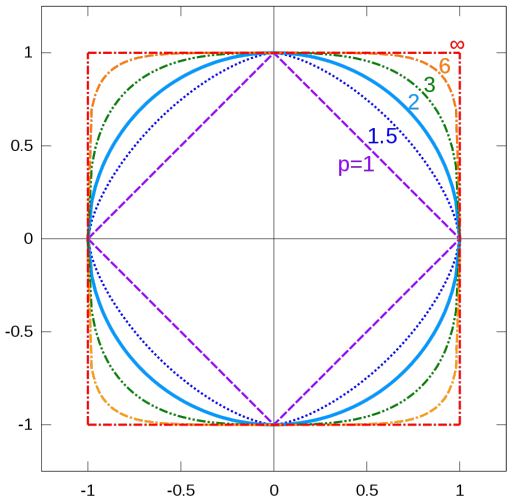
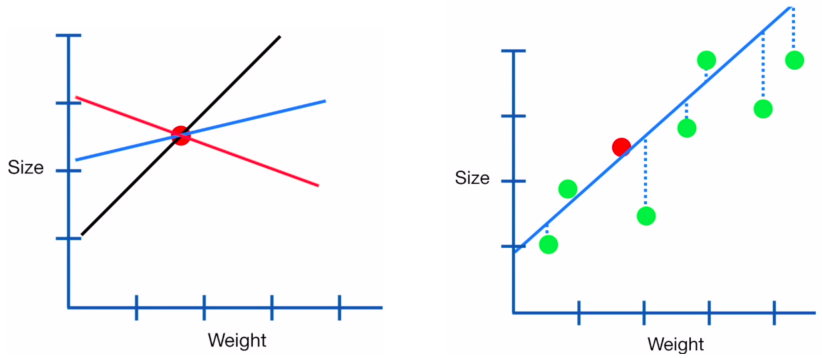


Figure 17: L with different q values.

As said before, this method could be not only useful for avoiding the overfitting, it can be also really useful to give results even if we do not have enough data for all the coefficients (features) that we want to use for our model. E.g. current problem in genetic problems, where the gens have a lot of values to take into account (coefficients to find) and maybe there is not enough data.
Below, there is an example to show in a good way the utility of applying regularization if we do not have enough data.



(a) Possible models without using ridge regularization.

(b) Result of applying ridge regularization.

Figure 18: Generic example of possible linear models of one training data point (red dot) and different test points (green dots).

Ridge regression
Ridge regression computes the sum of squared residuals, as in linear regression, but adds the hyper-parameter λ multiplied by the slope squared. The bigger is λ the less importance give to the distances

and the more importance it give to the slope of the result function. If the value of λ is equal to 0, we are talking of traditional linear regression, on the other hand if the value of λ is equal to ∞ the image of the straight line will be constant and then we will see a horizontal line. At the end, all the coefficients of the function will be close to zero or zero if we increase a lot the number of λ, this is the way to subtract all the features that has not importance, they get faster close to 0.
In the figure 19 we can see how the ridge regression works.

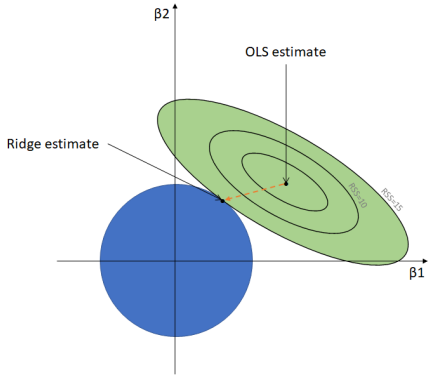


Figure 19: Contours of the error and constraint functions for ridge regression.

The blue area is the constraint region ($\beta_1^2 + \beta_2^2 \leq s$), which has the shape of a ball ($\beta_1, \beta_2 \neq 0$) where you have some large betas and some small, but all participate in the prediction. The green ellipses is the shape of the Residual Sum of Squares (RSS) function. Below there is the equation that represents the RSS:

$$RSS = ||r||_2^2 = \sum_{i=1}^{N} (\hat{y}_i - t_i)^2$$

In order to apply the ridge regression the data has been splitted in 60% for the training, 20% for the testing and 20% for validating the hyper-parameter λ.

To see the impact of the hyper-parameter λ in the ridge regression models, we have executed the program with different values for λ, [0, 1, 5, 10, 50, 100, 250, 500], and then we created a table 2 with all the metrics got by the different λ. In the python ridge regression calls the λs are specified as alphas.

Lambda values (alphas)	R^2	**RMSE**	**MAE**
0	0.923387	0.278235	0.217195
1	**0.92337**	**0.278265**	**0.217231**
5	0.923004	0.27893	0.21781
10	0.92202	0.280706	0.219314
50	0.905356	0.309249	0.240691
100	0.87866	0.350158	0.274033
250	0.795889	0.454145	0.361608
500	0.674812	0.573229	0.573229

Table 2: Table with the R^2, RMSE and MAE for different alphas.

In the table 2 we can see that with $\lambda = 0$ (linear regression) we have less error with RMSE, MAE, this is expected because by increasing λ values we are giving more importance to slope than the real distance between the points. We can also see that the λ that give best results of RMSE, MAE and R^2 is when is equal to 1.
A good way to see the impact of λ in ridge regression is by plotting the metrics by changing it values. In the figure below we can see the results of plotting the metrics by λ values.

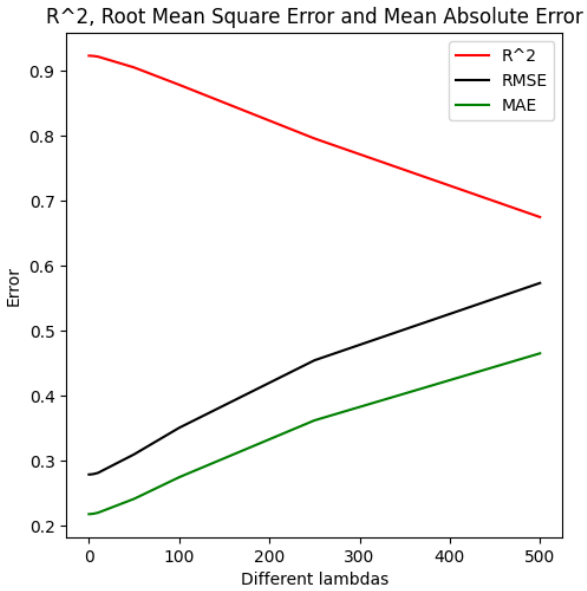


Figure 20: Plot of R^2, RMSE and MAE against different λs.

The following plot shows the coefficients of our ridge regression by changing the λ values.

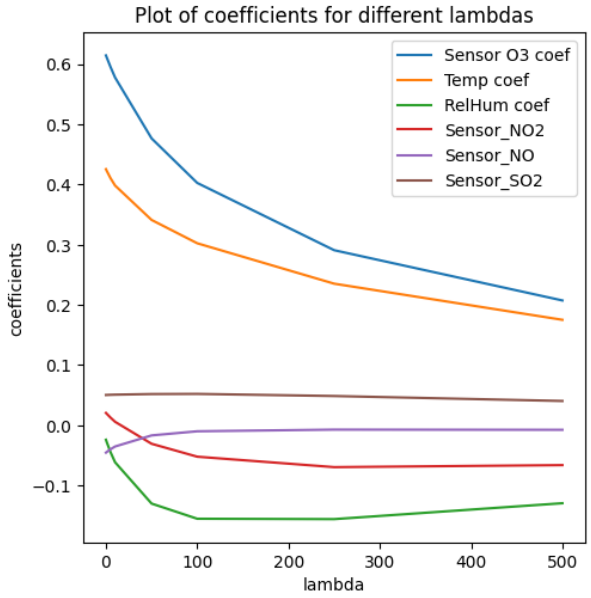


Figure 21: Plot of the βs for every λ.

In the figure 21 we can see what we said before, the more we increase the value of λ the more close we make all the coefficients to 0. Also we can see that the coefficients that have more weights are Sensor O_3, Temp and RelHum, something that we have already confirmed by looking the plots of data and the forward subset selection.
Once we know the best value for λ we can take the all the coefficients of its model and, check the metrics and how good it is by plotting against the RefSt.
The values of the coefficients for $\lambda = 1$ are $\beta_0 = 0.00418$, $\beta_1 = 0.61023$, $\beta_2 = 0.42208$, $\beta_3 = -0.02839$, $\beta_4 = 0.01901$, $\beta_5 = -0.04409$ and $\beta_6 = 0.05048$.
The metrics result got by testing data with the ridge regression model with $\lambda = 1$ are:

- R^2: 0.92106
- **RMSE**: 0.28242
- **MAE**: 0.22115

In the figure 22 we can see the Ridge regression model generated with $\lambda = 1$ (blue) against the RefSt (red).

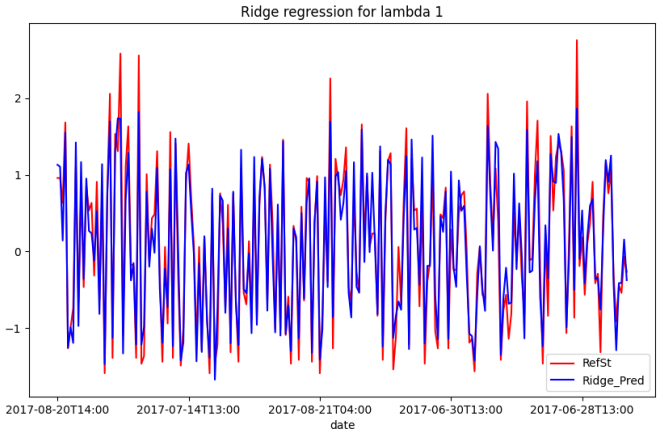


Figure 22: Prediction with ridge regression for $\lambda = 1$.

We also have plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

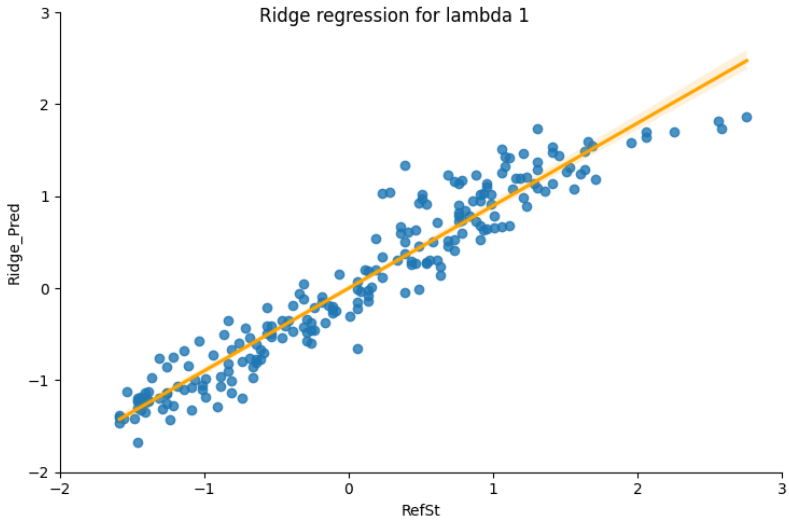


Figure 23: Linear dependence between the ridge regression prediction and Reference Station values.

Lasso regression

In the figure 24 we can see how Lasso regression works. The diamond area is the constraint region ($|\beta_1| + |\beta_2| \leq s$), where some betas are eliminated and not participate in the prediction. The ellipses are the contours of the RSS.

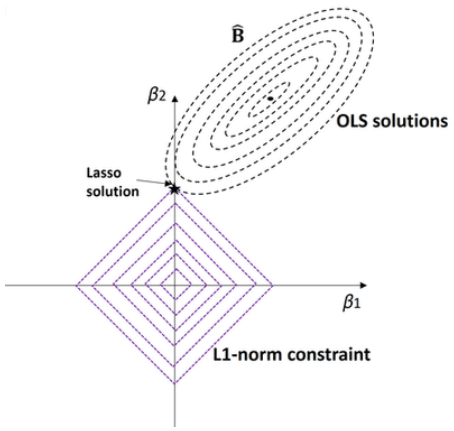


Figure 24: Contours of the error and constraint functions for LASSO regression.

In order to apply the ridge regression the data has been splitted in 60% for the training, 20% for the testing and 20% for validating the hyper-parameter λ.

To see the impact of the hyper-parameter λ in the ridge regression models, we have executed the program with different values for λ, [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9], and then we created a table 3 with all the metrics got by the different λ. In the python lasso regression calls the λs are specified as alphas.

Lambda values	R^2	**RMSE**	**MAE**
0.1	**0.906757**	**0.306951**	**0.238852**
0.2	0.870084	0.36232	0.284852
0.3	0.808958	0.439365	0.351345
0.4	0.723381	0.528691	0.430723
0.5	0.613355	0.625054	0.51875
0.6	0.478878	0.725656	0.608961
0.7	0.341566	0.815675	0.688679
0.8	0.193998	0.902462	0.764586
0.9	0.0267552	0.991681	0.845379

Table 3: Table with the R^2, RMSE and MAE for different lambdas.

In the table 3 we can see that the λ that give best results of RMSE, MAE and R^2 is when is equal to 0.1. The coefficient values with that λ are $\beta_0 = 0.01811$, $\beta_1 = 0.55316$, $\beta_2 = 0.38535$ and $\beta_3 = \beta_4 = \beta_5 = \beta_6 = 0$.
A good way to see the impact of λ in ridge regression is by plotting the metrics by changing it values. In the figure below we can see the results of plotting the metrics by λ values.

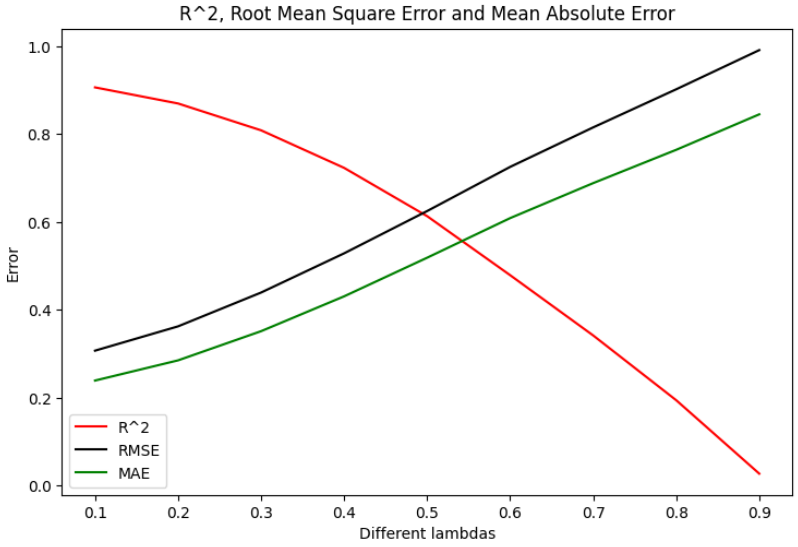


Figure 25: Plot of R^2, RMSE and MAE against different λ.

In the figure 25 we can see that unlike ridge regression, lasso regression's hyper-parameter penalize more in the model.
The following plot shows the coefficients of our lasso regression by changing the λ values.

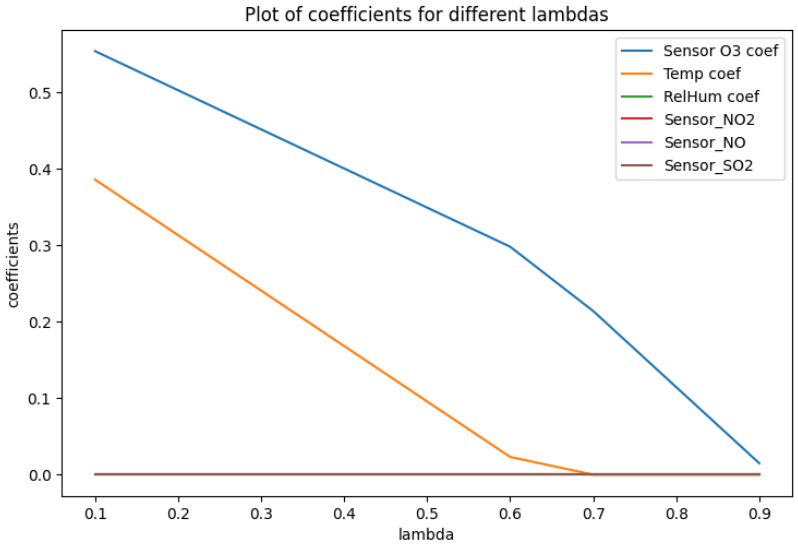


Figure 26: Plot of the βs for every λ.

In the figure 26 we can see that lasso regression penalize more all the coefficients of our model. Only two features, Sensor O_3 and Temp that has a direct impact in the model resist more before converging to 0. That remarks that these are in fact the most important features of them.

In the figure 27 we can see the Lasso regression model generated with $\lambda = 1$ (blue) against the RefSt (red).

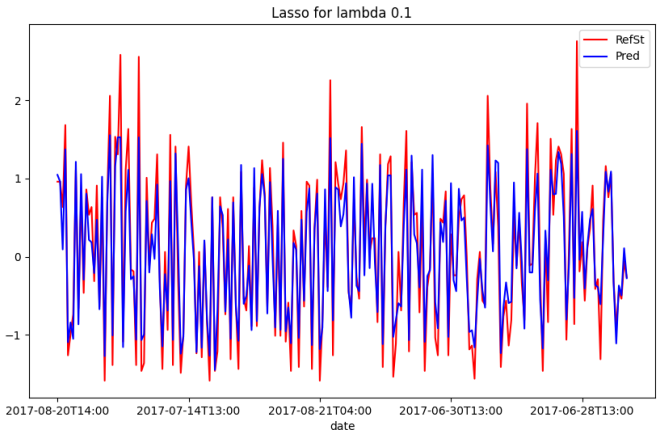


Figure 27: Prediction for Lasso regression for $\lambda = 0.1$ against RefSt.

The model has the following errors metrics if we compare to the test data:

- R^2: 0.90158
- **RMSE**: 0.31536
- **MAE**: 0.24577

We also have plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

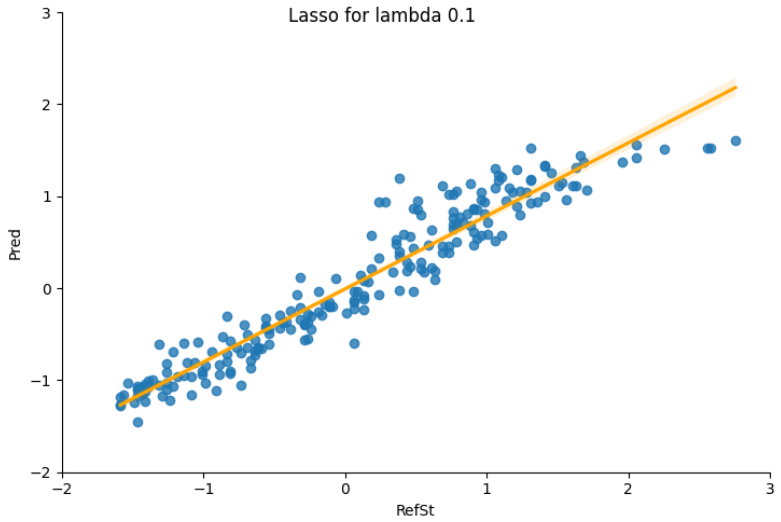


Figure 28: Linear dependence between the Lasso regression prediction and RefSt.

3.8 K-nearest neighbour

K-nearest neighbors (KNN) is a data classification method for estimating the likelihood that a data point will become a member of one group or another based on the data points that are closer to it. In other words, the way to predict a point is done by calculating the average of the k-nearest neighbours of a given point.
The following ordered list shows the pseudo-code of applying the method:

1. Load the data.
2. Choose the K value.
3. For each data point in the data:
    - Find the Euclidean distance to all training data samples.
    - Store the distance on an ordered list and sort it.
    - Choose the top K entries from the sorted list.
    - Label the test point based on the majority of classes present in the selected points

So, as we can see, the concept is the same as the previous models but with different letter, our hyper-parameter now is k. Then before doing creating any model we should take into account the following:

- For smaller values for k, e.g. $k = 1$. The model will be taking only into account only the closer first point to the current point, without taking into account the average. That translates into a **low bias** and a **big variance**, i.e. **overfitting**.
- On the other hand, for large values of k, the model is taking to account the average of all the k neighbours. Although, is important to know that if we do not stop increasing the value of k we will increase the error of the model. That case translates into a **big bias** and **low variance**, i.e. **underfitting**.

Then how can we know which is the best value for k? Running the model with k-fold cross validation. To find the right value of k to have a good-fitting we have run the model for different values, [2, 5, 7, 10, 12, 15, 18, 22, 25], using the cross-validation and we checked for the metrics and the following respective plots. Is important to remind that we are going to use the **Euclidean distance** in order to search for the minimum distance between the points, parameter p.

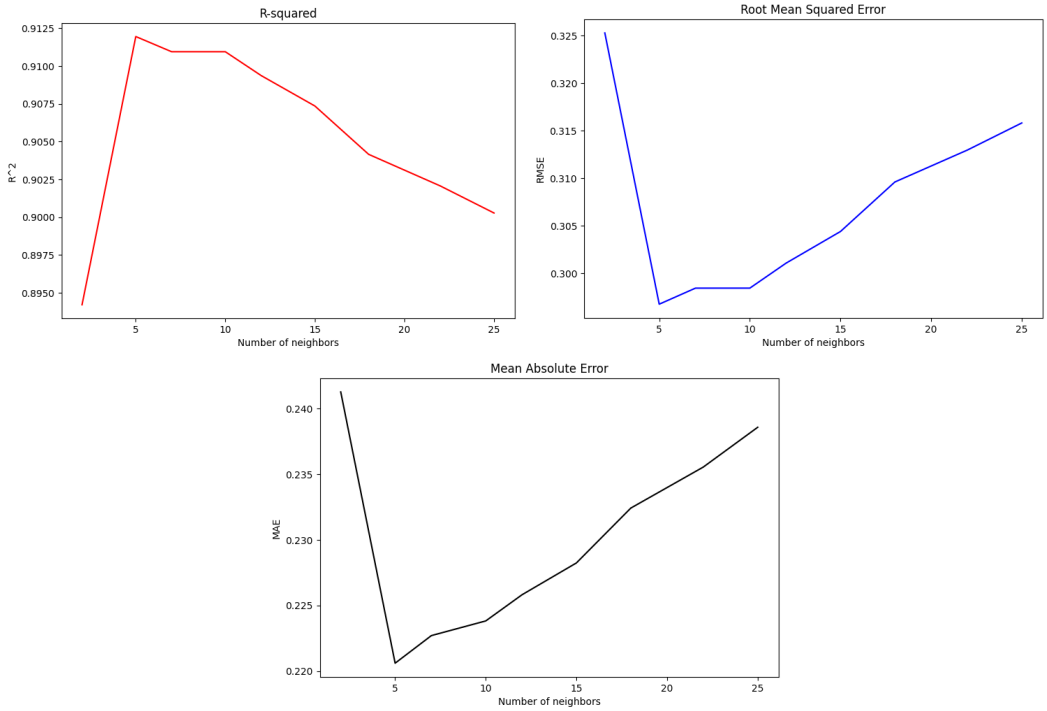


Figure 29: Plot of the R-squared, RMSE and MAE against different k values.

As we can see in the figure 29 we can corroborate that if we increase a lot the number of neighbors the system is increasing the RMSE and MAE errors and decreasing the R-squared, what it means that the system is underfitting. We can also see the other case, where it overfits, this case is probably when the number of neighbors is 5.
The values of the plot are also shown in the following table:

Neighbors	R^2	**RMSE**	**MAE**
2	0.894198	0.325272	0.241256
5	**0.91195**	**0.296732**	**0.220597**
7	0.910945	0.298421	0.222699
10	0.910944	0.298423	0.223816
12	0.909367	0.301054	0.225809
15	0.907352	0.304382	0.228232
18	0.904154	0.309591	0.23241
22	0.902062	0.31295	0.235536
25	0.90027	0.3158	0.238564

Table 4: Table with the R^2, RMSE and MAE for different number of neighbors.

In the table 4 we can see that the best results for R-squared, RMSE and MAE are with $k = 5$.
In the figure 30 we can see the K-nearest neighbors model generated with $k = 5$ (blue) against the RefSt (red) in terms of prediction.

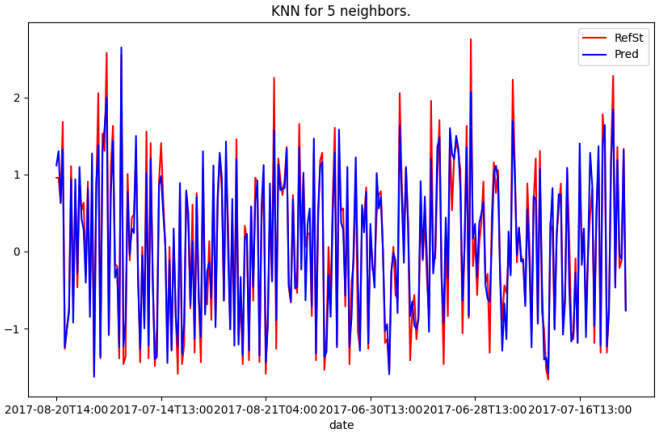


Figure 30: Prediction with KNN regression with $k = 5$.

In the figure 30 we can appreciate how the model is fitting good with the predictions in comparison to the RefSt.
The model has the following errors metrics if we compare to the test data:

- R^2: 0.93134
- **RMSE**: 0.26280
- **MAE**: 0.20064

We have also plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

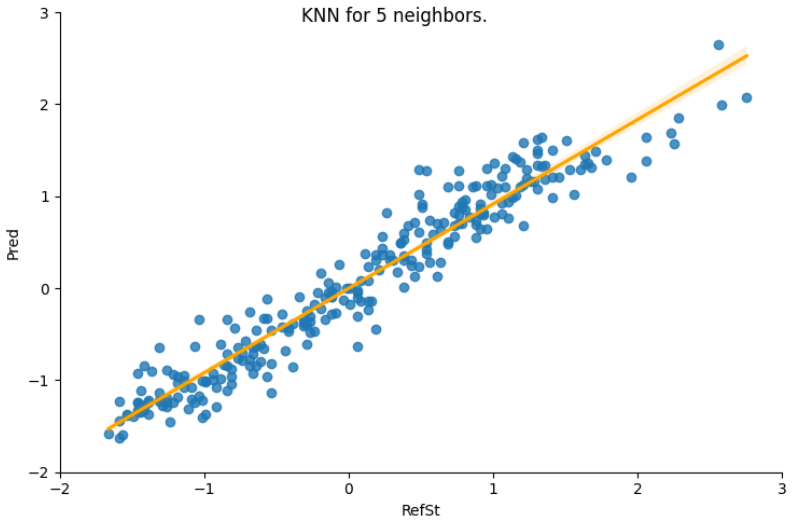


Figure 31: Linear dependence between KNN prediction against RefSt.

3.9 Kernel ridge regression with RBF kernel

In linear regression the model are created by using a linear function that express the relation between the data given, but sometimes the true-relationship of the data is non-linear and then the traditional linear models are not enough. This is why Kernel ridge regression is really important. The idea behind kernel method is to transform the data χ of our non-linear problem to in a feature space ($\mathbb{H}$) by feature mapping (Φ), this space can have infinite dimensions $\mathbb{R}^d$ where $d \to \infty$, where we can make the linear regression using an hyper-plane, this could be done by adding more dimensions to the data given in the current space.

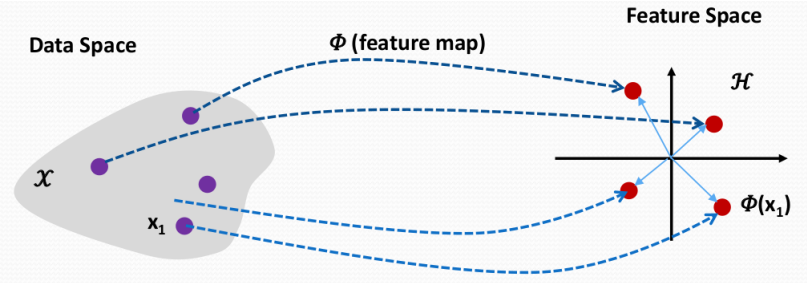


Figure 32: Representation of the feature map Φ.

Sometimes to create our model we have to operate doing the scalar product into this new feature space.

$$\tilde{y} \backsim <\Phi(x_i), \Phi(x_j)>$$

But this can be out of mind in terms of computation, the **kernel trick** let us to operate over the feature space without computing the embedding $\Phi(x_i)$ directly by using the kernel functions, which

simplify the problem.

$$k : \chi \times \chi \to \mathbb{R}, k(x_i, x_j) = <\Phi(x_i), \Phi(x_j)>$$

Another way to explain this method, all we are doing is using the **Gram Matrix K** of the data, in the sense that once we know K, we can throw away our original data. The **design matrix** is the matrix containing all the data X. Then, once we mapped to the feature space the Gram Matrix will contain the inner products of the feature space (Φ). Then without knowing the Φ we need to know how to compute the kernel, *K(x,y)*. In order to achieve that, the kernel, must fulfill the Mercer's theorem in order to be valid for the kernel trick. **Mercer's theorem**:

- A symmetric function *K(x,y)* can be expressed as an inner product:

$$K(x, y) = <\Phi(x), \Phi(y)>$$

- For some Φ if and only if *K(x,y)* is positive semi-definite:

$$\int K(x, y)g(x)g(y)dxdy \geq 0 \; \forall g$$

In our case the we are going to use the Radial Basis Functions (RBF), also called RBF kernel to create our Machine Learning model. RBF are real value functions, $\varphi(x)$, whose value depends on the distance between the input value and some fixed point. In the kernel Gaussian functions each Gaussian function has the average fixed on the value of x. The function can be represented as:

$$\varphi(x) = exp(-\frac{1}{2\sigma^2}(||x - \mu||^2))$$

This can kernel function can be very useful because it gives weights (importance) in a non-linear way to the different data.

In addition, if we take a look to the **Moore-Aronszajn theorem**: k($\cdot$) is a p.d. (positive definite) kernel on the set χ iff (if and only if) there exists a Hilbert space H and a mapping.

$$\Phi : \chi \to \mathbb{H}$$

such that, for any x, y $\in \mathbb{H}$. Then we can say that each data point of of our training data is mapped to a Gaussian function living in a Hilbert space. Is important to remark that Hilbert space belongs to the inner product space H. Then, in this Hilbert space we can create a convex set by making the linear combinations between the all points (functions).

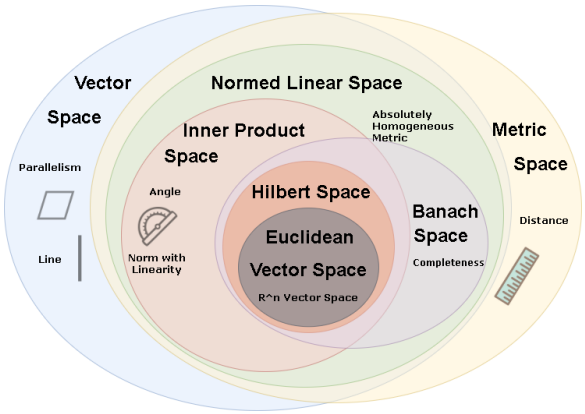


Figure 33: Sets of the different spaces.

Just to remind Kernel ridge regression combines ridge regression (L2-norm regularization) with the kernel trick so it will have also the hyper-parameter that will penalize in order to regularize.

As we did in the previous model we are going to tune the hyper-parameter λ by using the k-fold cross validation (10-fold cross validation). The values that we will take in order to take the metrics and plot them are [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1].
In the following figures we can see the results of applying the different values for lambdas.

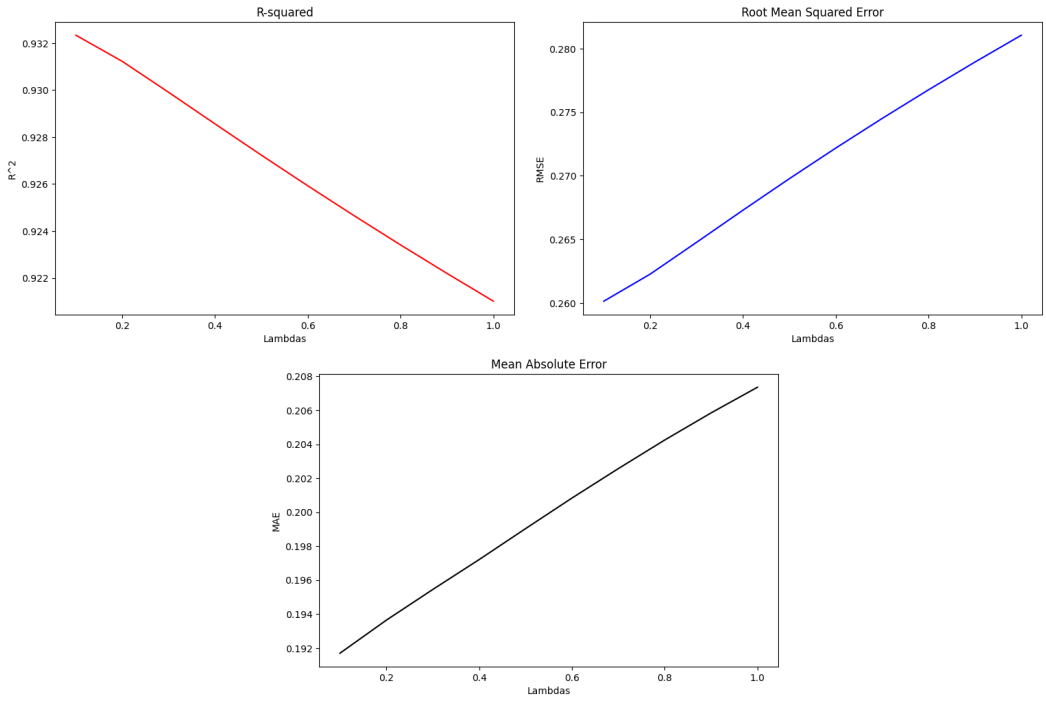


Figure 34: Plot of the R-squared, RMSE and MAE against different λs.

As we can see in the figure 34 the error behaviour of the metrics follow, more or less, a linear function. When $\lambda = 1$ the errors of the RMSE and MAE are too much and it is not a good model to take into account, and probably is going to underfit. On the other hand with a value of $\lambda < 0.2$ the model seem that is predicting pretty good, but we have to be careful to not overfit.
The values of the plot are also shown in the following table:

Lambda values	R^2	**RMSE**	**MAE**
0.1	**0.932331**	**0.260133**	**0.191691**
0.2	0.931218	0.262264	0.193638
0.3	0.929901	0.264763	0.195443
0.4	0.928557	0.267289	0.197205
0.5	0.927227	0.269766	0.199027
0.6	0.925923	0.272171	0.200828
0.7	0.92465	0.2745	0.202562
0.8	0.923408	0.276753	0.204246
0.9	0.922197	0.278933	0.205844
1	0.921014	0.281044	0.207357

Table 5: Table with the R^2, RMSE and MAE for different number of neighbours.

As it shows the table 5 the best value that we can take for the hyper-parameter λ is when is 0.1.
In n the figure 35 we can see the Kernel ridge regression with RBF model generated with $\lambda = 0.1$ (blue) against the RefSt (red) in terms of prediction.

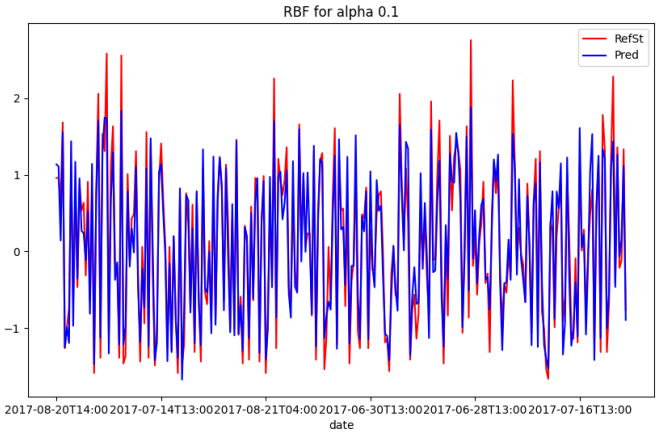


Figure 35: Prediction with RBF kernel regression for $\lambda = 0.1$.

In the figure 35 we can appreciate how the model is fitting good with the predictions in comparison to the RefSt.
The model has the following errors metrics if we compare to the test data:

- R^2: 0.91747
- **RMSE**: 0.28813

- **MAE**: 0.22463

We have also plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

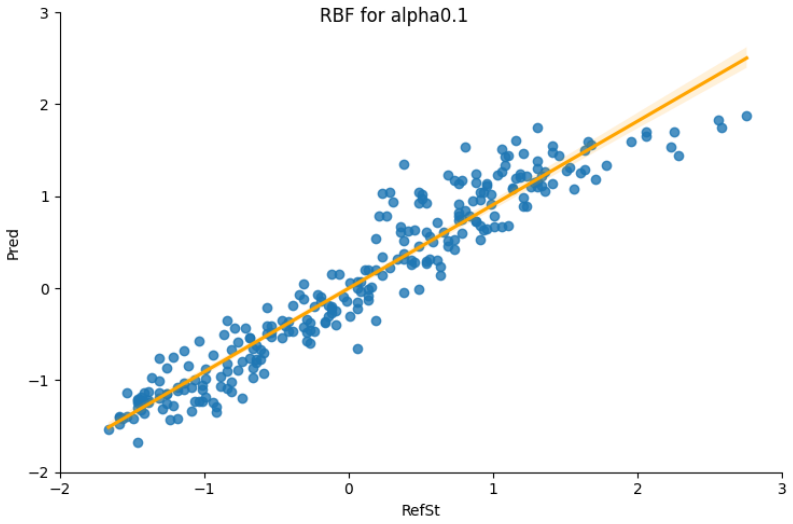


Figure 36: Linear dependence between RBF kernel prediction against RefSt.

3.10 Random forest

Random Forest (RF) is an algorithm that can be used for Classification and Regression problems in ML. It is based on the concept of **ensemble method**. The ensemble learning is a technique which combines multiple models, trained over the same data, to make accurate predictions than a single model. The idea behind is that the errors, in random forest decision trees, of each model are just independent and different from tree to tree. There are a lot of ensemble methods, one example is Bootstrapping. **Bootstrapping** samples randomly subsets from the data given over a given number of iterations and a given number of variables. Once is done, the results obtained are averaged to obtain a better result.

Random Forest create the decision trees and averages all of them in order to create the model. The steps that the algorithm follows are the following ones:

1. Taking n number of random records from the data set having k number of records.
2. Individual decision trees are constructed for each of the samples.
3. Each decision tree will generate an output (result).
4. The final result is based on the Majority Voting or Average.

In the figure 37 we can see an overview of the functionality of the Random Forest algorithm.

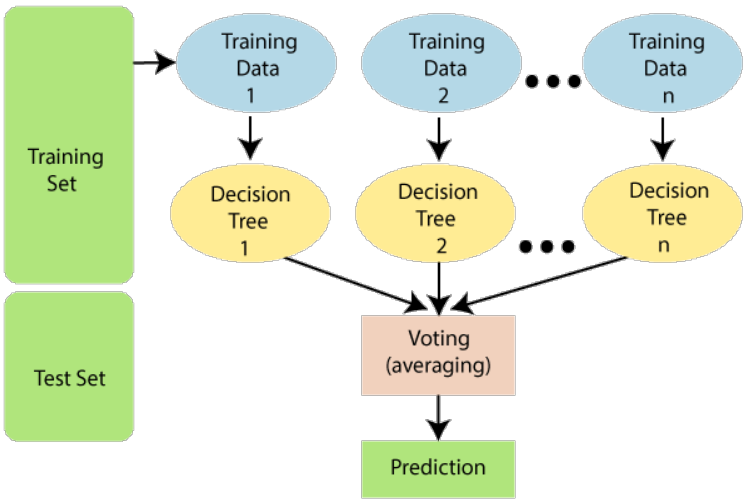


Figure 37: Diagram of the Random Forest's functionality.

As we did in the previous model we are going to tune the hyper-parameter, in this case M, by using the k-fold cross validation (10-fold cross validation). The values that we will take in order to take the metrics and plot them are [1, 2, 5, 7, 10, 13, 15, 18, 20, 25].

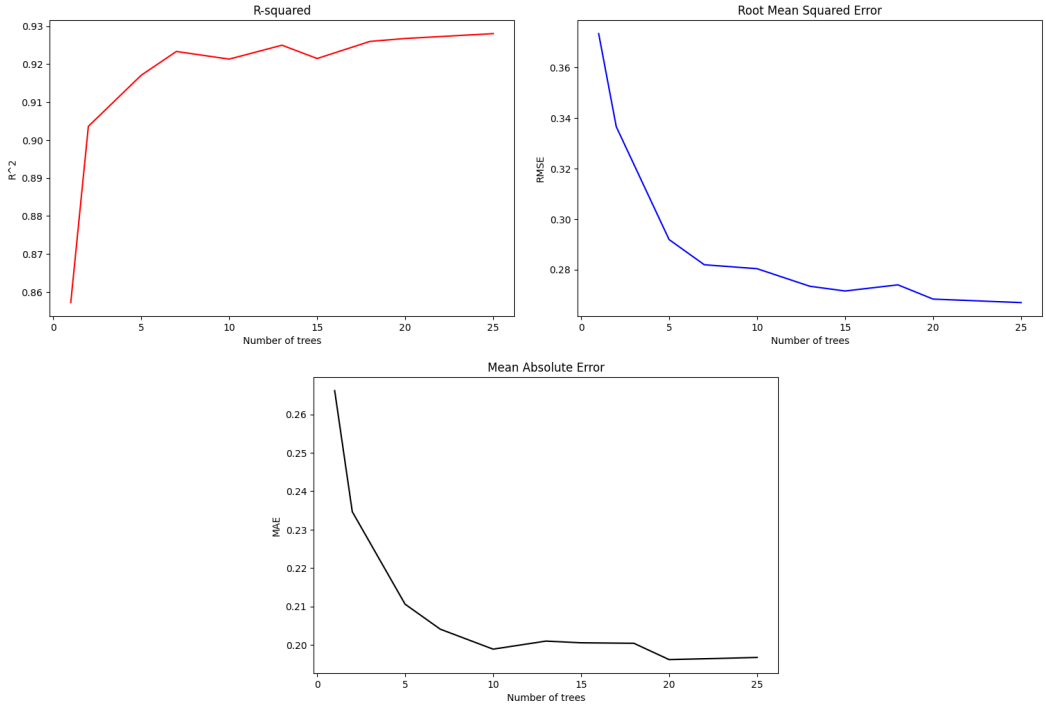


Figure 38: Plot of the R-squared, RMSE and MAE against different M values.

As we can see in the figure 38 the more we increased the number of M, decision trees, the more complex is our model, it give more weight to all the features. If we increased a lot the number of trees our model is going to overfit probably. On the other hand, if the have just a few number of trees, the model is not taking into account all the features with a correct weight, that leads to an underfitting scenario.
The values of the plot are also shown in the following table:

Number of trees	R^2	**RMSE**	**MAE**
1	0.85717	0.373464	0.266167
2	0.903606	0.336582	0.234657
5	0.917052	0.291876	0.21059
7	0.923331	0.281896	0.204082
10	0.921316	0.280323	0.198884
13	0.924982	0.273366	0.200978
15	0.921474	0.271496	0.200544
18	0.925984	0.273932	0.200409
20	0.926749	0.268288	**0.196164**
25	**0.928028**	**0.26689**	0.196719

Table 6: Table with the R^2, RMSE and MAE for different number of trees.

As it shows the table 6 the best value for MAE is when the number of trees are 20, and for the R-squared and RMSE when is 25. As we are giving more importance to RMSE, we are going to take as the best result for the hyper-parameter M the value 20.

In the figure 39 we can see the Random Forest model generated with $M = 25$ (blue) against the RefSt (red) in terms of prediction.

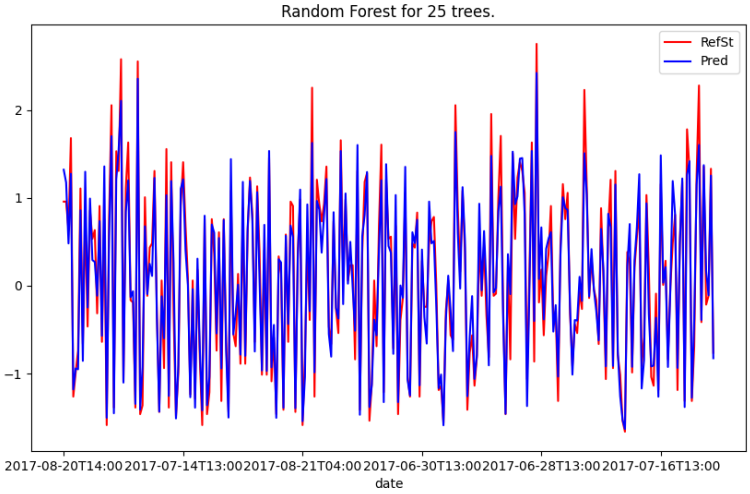


Figure 39: Prediction with Random Forest regression for $M = 25$.

The model has the following errors metrics if we compare to the test data:

- R^2: 0.942480
- **RMSE**: 0.240543
- **MAE**: 0.184892

We have also plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

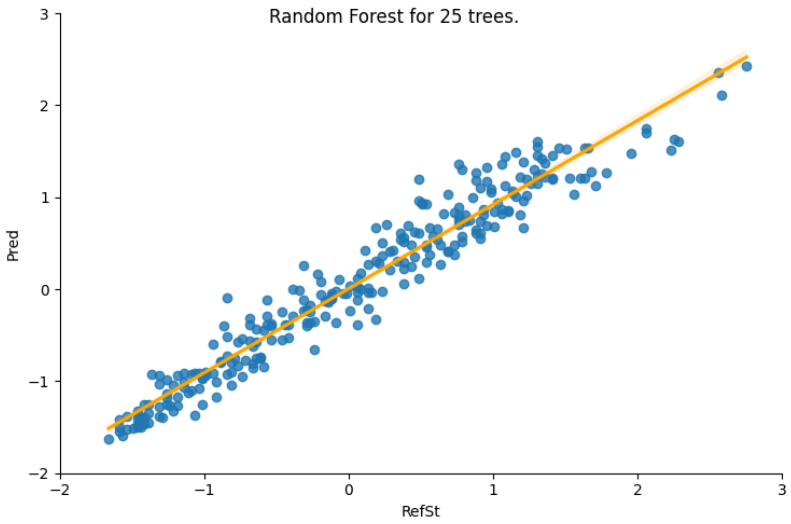


Figure 40: Linear dependence between Random Forest prediction against the RefSt.

3.11 Support Vector Regression

Support Vector Machines (SVM) are supervised learning models that analyze data used for classification and regression analysis. The objective of SVM algorithm is to find a hyper-plane, among all, in a n-dimensional space that classifies the data points given (training dataset). In SVM there are two distinguishable margins close to the hyper-plane, where there is not data point inside them, and then is going to select as a good hyper-plane the one that maximizes the margin. All the data points that are closest to the hyper-plane are called the Support Vectors. In our case we are going to implement **Support Vector Regression** (SVR).

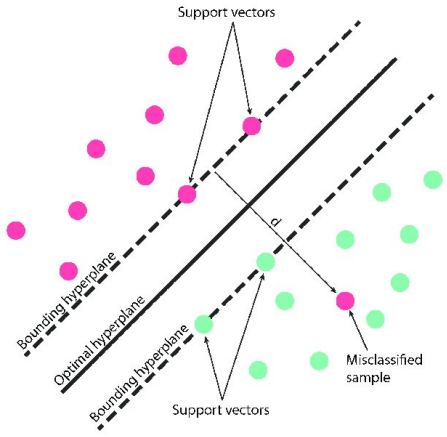


Figure 41: Example of a Support Vector Machine.

In the figure 41 we can see how the bounding hyperplanes are parallel to the optimal hyperplane and how the closest data point (support vectors) are part of them.
The idea behind the algorithm is to find the minimum perpendicular distance to the closest point (euclidean distance over all data points), and find the maximum distance to the for the weights w to closest points. The perpendicular distance of a data point from the margin is given by equation:

$$d_i = \frac{|w^T x_i + w_0|}{||w||}$$

And the maximizing problem as:

$$\min \frac{1}{2}||w||^2 \text{ subject to } t_i(w^T x_i + w_0) \geq +1, \forall i$$

Is important to remind that this last part is done by using the Lagrange multipliers.
In the following figure we can see a graphic representation of maximizing the margin in SVM.

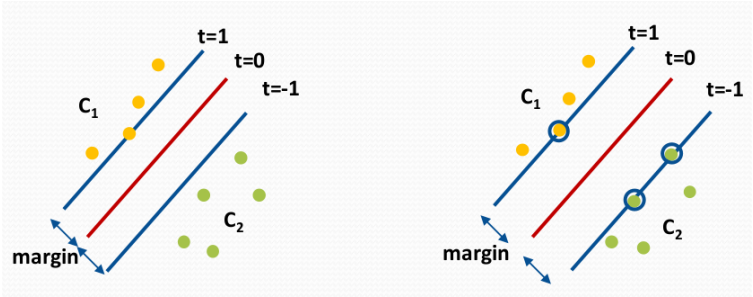


Figure 42: Maximizing the margin in SVM.

In the case of the regression, the SVR goal is to find a function $f(x)$ with at ε-most (ε-tube) deviation from target t. It solves a modified version of the SVM considering some errors, ε.

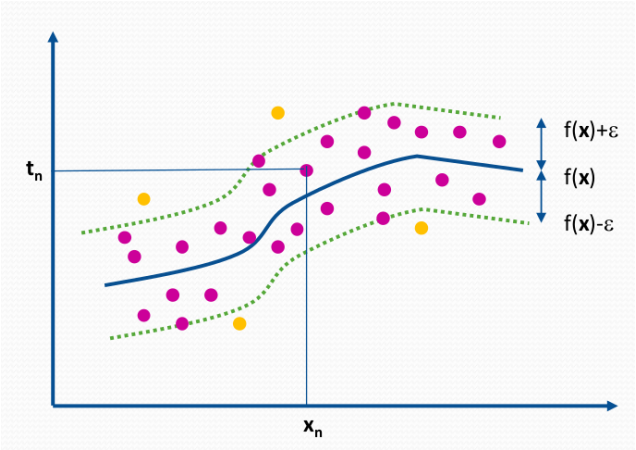


Figure 43: Representation of SVR.

In this model we also have the hyper-parameters that we have to tune in order to have a good fitting. In SVR the hyperaparameters that we need are:
The hyper-parameters we need to find are

- **C** (regularization): Adds a penalty for each misclassified data point. If the c is small, the penalty for the missclassified points is so low so the decision boundary with large margin is chosen, in other words, **underfitting**. While, large c values tries to minimize the number of missclassified examples due to the high penalty which results in a decision boundary with less margin, in other words, **overfitting**. It must be strictly positive and the penalty is a squared L-2 penalty.

- **Gamma** (γ): Defines how far the influence of a single training example reaches. With low γ, points far away from plausible separation lines are considered in the calculation for the separation line. While, high γ means the points close to a plausible line are considered in the calculation.

- **Epsilon**: In SVR, there is an ε-tube function that defines the loss function, and that is called ε-intensive loss function, and it affects the smoothness of the SVR response. The errors are zero for those points that are **inside** the ε-intensive loss function.

If we are using a linear kernel in our example we just need to optimize the c hyper-parameter. However, if we want to use an RBF kernel we need to use γ and c hyper-parameters.
In the following figures we can see a graphic representation of the hyper-parameters. The first figure (44) represents the C, the second one the γ (45) and the last one the Epsilon (46) hyper-parameter.

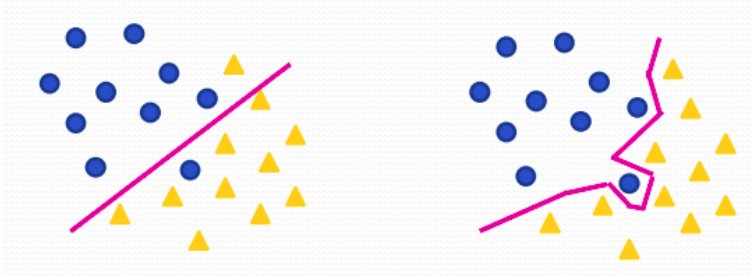


Figure 44: Hyper-parameter C in SVR with small value (left) and large value (right).

In the figure 44 we can see that if C is small tends to **underfitting**, while, big C tends to **overfitting**.

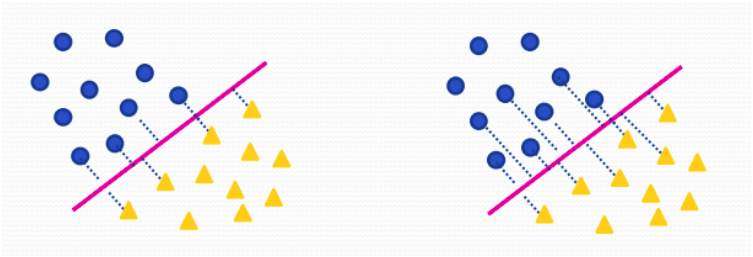


Figure 45: Hyper-parameter γ in SVR with high value (left) and low value (right).

In the figure 45 we can see that if γ is high only nearby points are considered, while, if γ is low far away points are also considered

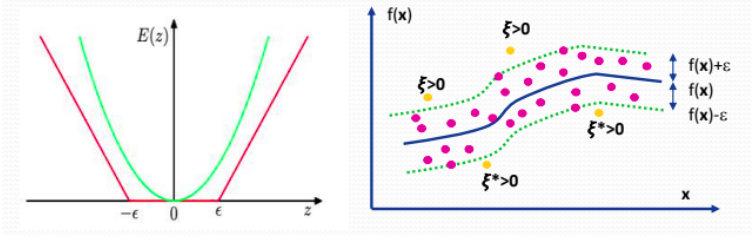


Figure 46: Hyper-parameter epsilon in SVR.

In order to implement the k-fold cross validation (10-fold cross validation) on the hyper-parameters we have used the `GridSearchCV` method from the sklearn python package.
For the values of the hyper-parameter we are going to use $C = [0.001, 0.01, 0.1, 1, 10, 100]$, $\gamma = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ and $epsilon = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$. The kernel implementation of model will used the RBF of kernel regression method implemented previously.
In the following table we can see the results of the respective hyper-parameters mentioned previously and all their metrics.

kernel	C	gamma	epsilon	MAE	MSE	R^2
rbf	1	1	0.100	-0.232	-0.109	0.889
rbf	1	1	0	-0.232	-0.108	0.890
rbf	10	1	0.100	-0.241	-0.107	0.890
rbf	1	1	0.200	-0.244	-0.118	0.880
rbf	10	1	0.200	-0.248	-0.114	0.883

Table 7: Table with the results of the grid search for the SVR hyper-parameters.

In the table 7 we can see that the best result results in $C = 1$, $\gamma = 1$ and $epsilon = 0.1$.

In the figure 47 we can see the SVR model generated with the best hyper-parameters found (blue) against the RefSt (red) in terms of prediction.

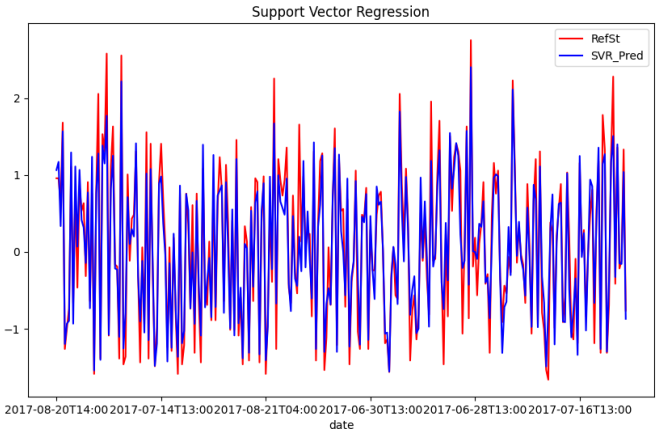


Figure 47: Prediction with SVR regression with best hyper-parameters values found.

The model has the following errors metrics if we compare to the test data:

- R^2: 0.889
- **RMSE**: 0.33015
- **MAE**: 0.232

We have also plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

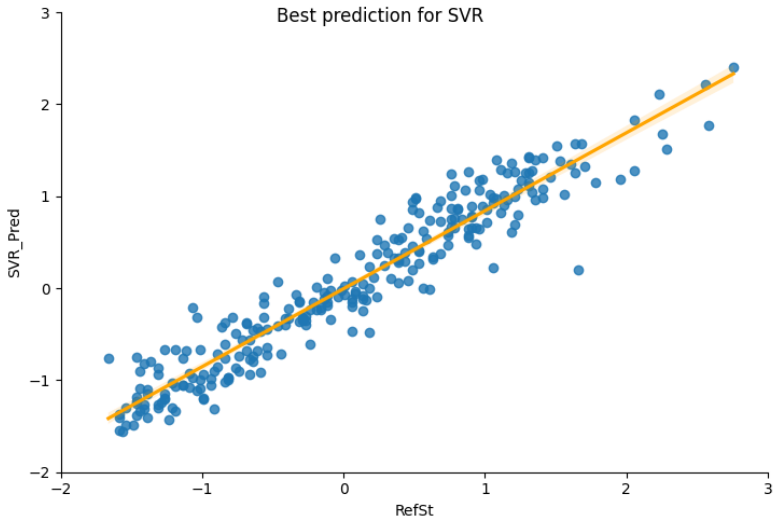


Figure 48: Linear dependence between SVR prediction against the RefSt.

The `SVR` method of the `sklearn.svm` Python package that we are using also accepts some predefined values for the `gamma` parameter.

- `Auto`. Uses $1/(n\_features * X.var())$.
- `Scale`. Uses $1/n\_features$.

The following table is obtained by using $gamma = ['auto', 'scale']$.

kernel	**C**	**gamma**	**epsilon**	**MAE**	**MSE**	R^2
rbf	10	auto	0.100	-0.189	-0.067	0.932
rbf	10	scale	0.100	-0.189	-0.067	0.932
rbf	1	auto	0	-0.194	-0.071	0.927
rbf	1	scale	0	-0.194	-0.071	0.927
rbf	1	scale	0.100	-0.195	-0.070	0.928

Table 8: Table with the results of the grid search for the SVR hyper-parameters.

In the table 8 we can see that the best values for the hyper-parameters are: $C = 10$, $\gamma = auto$ and $epsilon = 0.1$.
In the figure 49 we can see the SVR model generated with the new best hyper-parameters found (blue) against the RefSt (red) in terms of prediction.

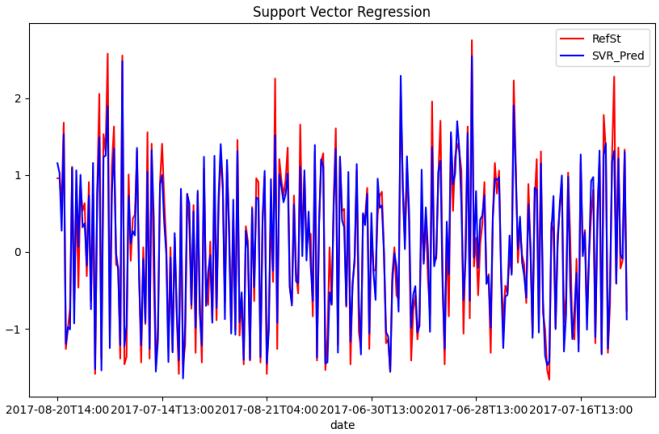


Figure 49: Prediction with SVR regression for the new best hyper-parameters found.

The model has the following errors metrics if we compare to the test data:

- R^2: 0.932
- **RMSE**: 0.25884
- **MAE**: 0.189

We have also plot the predicted values against the RefSt to see their linear dependence by using the `seaborn` python package.

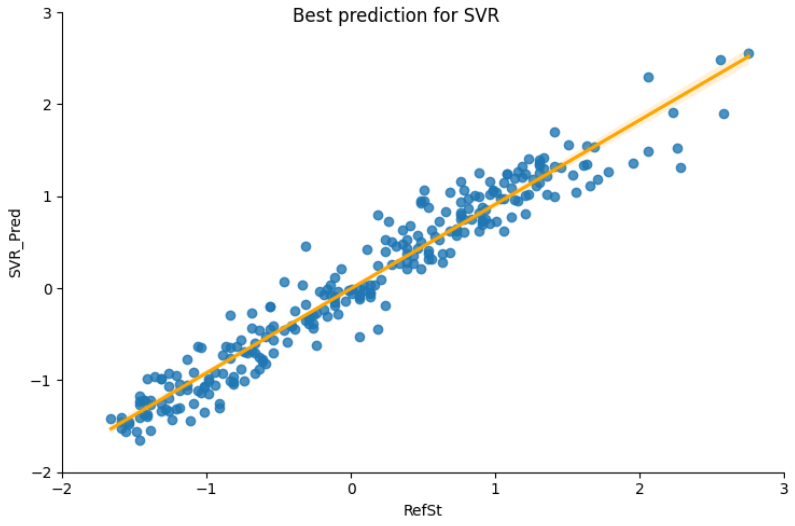


Figure 50: Linear dependence between SVR prediction against RefSt.

4 Evaluation and comparison

In this section we are going to compare all the models between them. In ML there is not the perfect model for predicting the data, it depends on the problem statement. At the end all the methods are doing the same but with different approaches.

Machine Learning Method	R^2	**RMSE**	**MAE**
Best forward subset	0.91477	0.29280	0.22589
Ridge regression $\lambda = 1$	0.92106	0.28242	0.22115
Lasso $\lambda = 0.1$	0.90158	0.31536	0.24577
KNN $k = 5$	0.93134	0.26280	0.20064
Kernel RBF $\lambda = 0.1$	0.91747	**0.228813**	0.22463
Random Forest $trees = 25$	**0.942480**	0.240543	**0.184892**
SVR	0.889	0.33015	0.232
SVR $gamma = auto$	0.932	0.25884	0.189

Table 9: Table with the all the results of ML models using the best values for the hyper-paramters.

We can see in the table 9 that the best values for the metrics R-squared and MAE is by using the Random Forest with $trees = 25$ and, the best value for the metrics RMSE is for the Kernel RBF model using the $\lambda = 0.1$. As we are giving more weight to the RMSE for creating our Machine Learning model, we can say that the model that it seems to **fit better** with our data is the **Kernel RBF** ($\lambda = 0.1$). However, we should be very careful and watch closely if we are close to be overfitting.