Supervised Learning

Miguel Ángel Matas Rubio (User Driven-Data: Miguel-Matas)

Jesús Santiyán Reviriego Miguel Ángel Roldán Mora

January 23, 2023

Abstract

Based on aspects of building location and construction, the goal is to predict the level of damage to buildings caused by the 2015 Gorkha earthquake in Nepal. The data was collected through surveys by Kathmandu Living Labs and the Central Bureau of Statistics, which works under the National Planning Commission Secretariat of Nepal. We're trying to predict the ordinal variable damagegrade, which represents a level of damage to the building that was hit by the earthquake.

Contents

1 Introduction

Nowadays, the massive amount of data generated by all devices requires the use of mechanisms to process the data and obtain more relevant information from it. To do this, machine learning algorithms are used which treat the data in different ways depending on the algorithm chosen. This work consists of applying these algorithms to data collected on the buildings damaged in an earthquake in Nepal in order to predict the damage that these buildings will suffer.The data used can be obtained from the following link[2]. The algorithms used in this work are: NaiveBayes, Knn, Decision Tree, Random forest and Hyperparameters Optimization and Boosting.

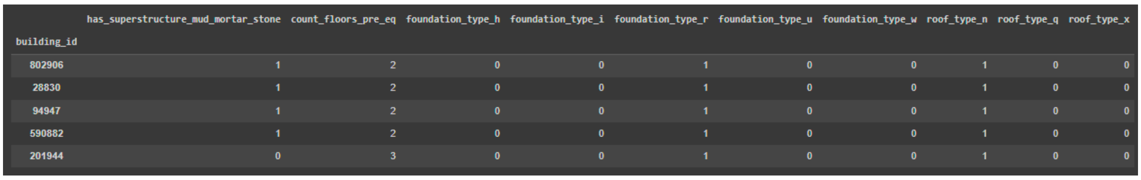
2 Data

For the realization of the work we have obtained data from the "DrivenData" contest, which provides a database[1] to work on, where you can see all the variables that affect the destruction of buildings in a city in Nepal produced by the effect of an earthquake.

The choice of attributes is very important when making the prediction algorithms, as the predictions will depend on which ones are chosen to be more or less accurate. For this study we have chosen the features that are the most representative when making predictions, as the rest of the features did not have any relevant impact. The most representative ones are:

- has_superstructure_mud_mortar_stone
- count_floors_pre_eq
- foundation_type
- roof_type

Below is the dataset of the chosen features with which the prediction algorithms will subsequently work.



	has_superstructure_mud_mortar_stone	count_floors_pre_eq	foundation_type_h	foundation_type_i	foundation_type_r	foundation_type_u	foundation_type_w	roof_type_n	roof_type_q	roof_type_x
building_id										
802906	1	2	0	0	1	0	0	1	0	0
28830	1	2	0	0	1	0	0	1	0	0
94947	1	2	0	0	1	0	0	1	0	0
590882	1	2	0	0	1	0	0	1	0	0
201944	0	3	0	0	1	0	0	1	0	0

Figure 1: Selected features table

The heat map below shows the correlations of the chosen attributes.

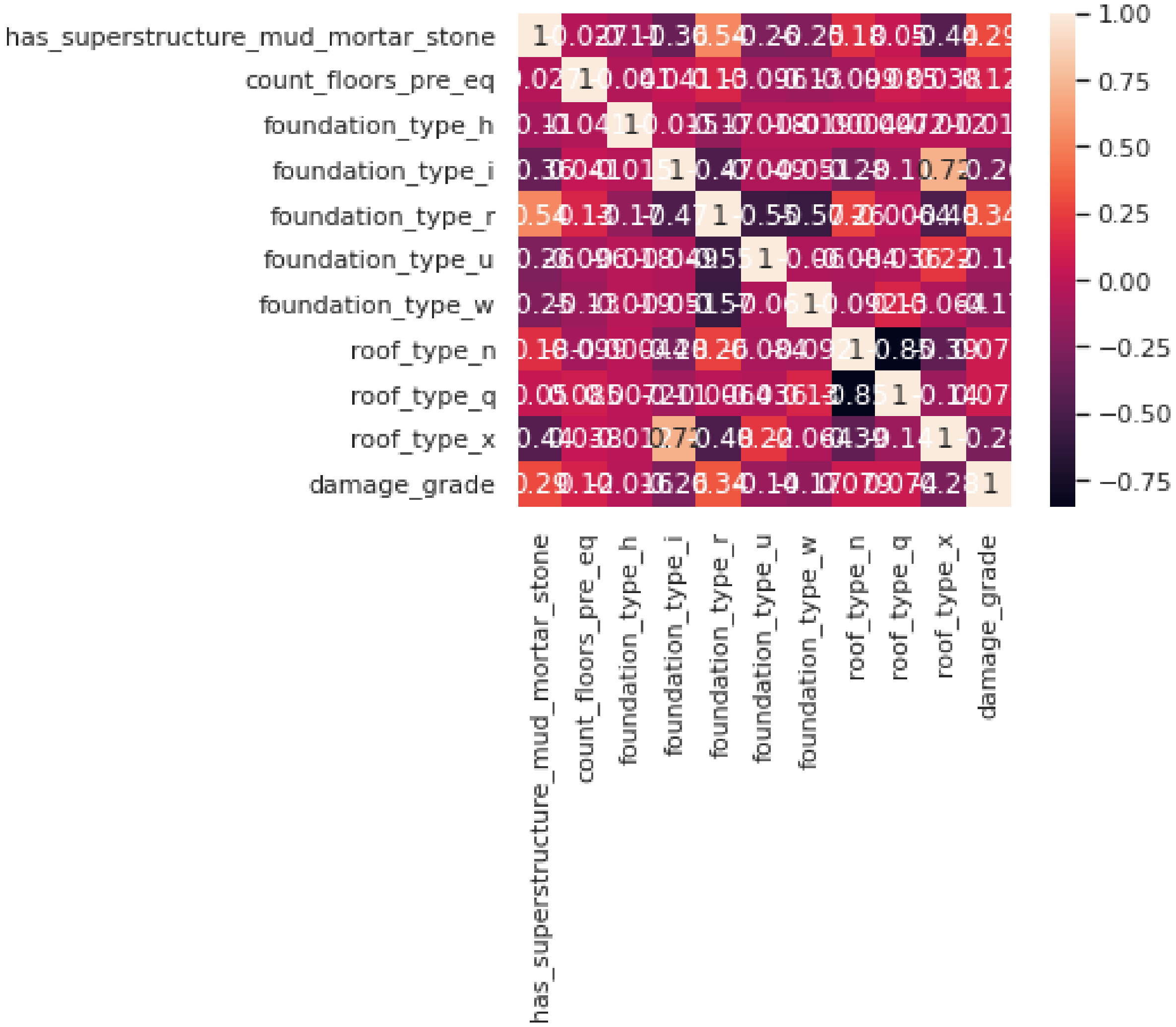


Figure 2: Heatmap

3 Algorithms

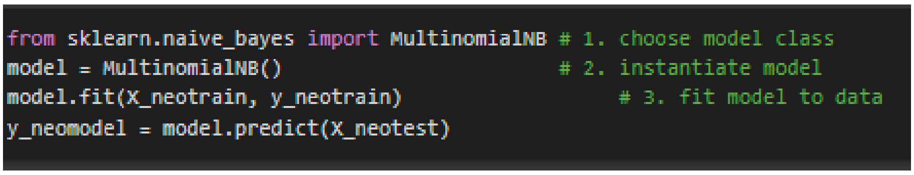
3.1 NaiveBayes

Naive Bayes is a simple, yet effective, machine learning algorithm for classification tasks. It is based on the idea of using Bayes' Theorem to make predictions about the likelihood of an event occurring, given certain features or evidence.

The algorithm works by first building a model of the data using statistical techniques to estimate the probability of certain outcomes based on the available features. Then, when given a new set of data, the model can predict the class or category that the new data belongs to, based on the probabilities calculated in the training phase.

One of the key assumptions of the Naive Bayes algorithm is that all the features are independent of one another. This means that the presence or absence of one feature does not affect the presence or absence of any other feature. While this assumption is often unrealistic, the algorithm still tends to perform well in practice, especially for large datasets with many features.

There are several variations of the Naive Bayes algorithm, including the Gaussian Naive Bayes, which is used for continuous data, and the Multinomial Naive Bayes, which is used for count data. Overall, Naive Bayes is a fast and easy-to-implement algorithm that can be a good choice for classification tasks, especially when the dataset is large and the relationships between features are not well understood.[2]



```
from sklearn.naive_bayes import MultinomialNB # 1. choose model class
model = MultinomialNB()                       # 2. instantiate model
model.fit(X_neotrain, y_neotrain)                  # 3. fit model to data
y_neomodel = model.predict(X_neotest)
```

Figure 3: NaiveBayes

Figure 3 shows the NaiveBayes code where the multinomial model was chosen, as it provided a better prediction than the others.

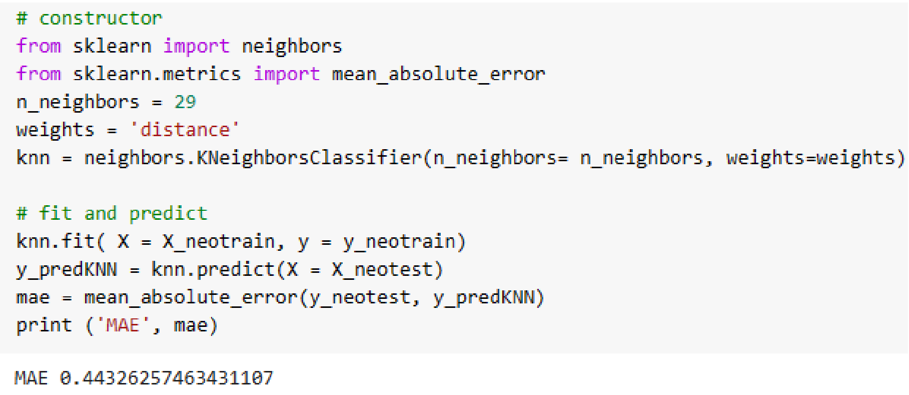
3.2 KNN

K-nearest neighbors (KNN) is an algorithm used for classification and regression. It works by finding the K-nearest data points in the training set for a given data point in the test set and using the labels (for classification) or the target values (for regression) of those data points to make a prediction.

Here's how the KNN algorithm works:

- Choose the number of neighbors K.
- Find the distance between the test point and all training points.
- Sort the distances in ascending order and pick the K-nearest data points.
- If the algorithm is used for classification, assign the label that appears most frequently among the K-nearest data points. If the algorithm is used for regression, assign the mean or median target value of the K-nearest data points.

In our work, by implementing KNN on the selected attributes, we obtain that when performing cross-validation the optimal number of neighbours is 29.



```
# constructor
from sklearn import neighbors
from sklearn.metrics import mean_absolute_error
n_neighbors = 29
weights = 'distance'
knn = neighbors.KNeighborsClassifier(n_neighbors= n_neighbors, weights=weights)

# fit and predict
knn.fit( X = X_neotrain, y = y_neotrain)
y_predKNN = knn.predict(X = X_neotest)
mae = mean_absolute_error(y_neotest, y_predKNN)
print ('MAE', mae)
```

```
MAE 0.44326257463431107
```

Figure 4: KNN

When running the algorithm we obtain the following graph:

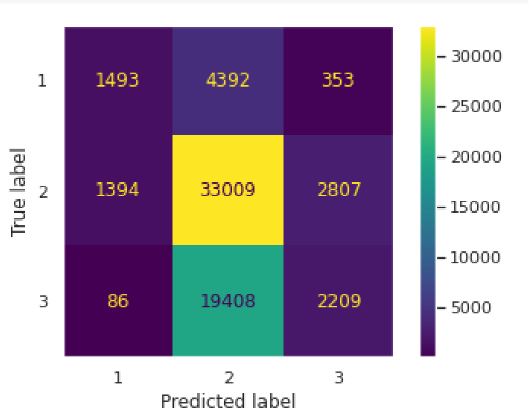


Figure 5: Matrix KNN

Figure 5 shows the results of the KNN algorithm, where of the three values to be predicted, the one that is most correct is value 2, giving much better results than for the results of 1, since it is very wrong with the values that are also 2 in proportion to the rest of the values, and it can be deduced that the model is more likely to predict values 2.

3.3 Decision Tree

A decision tree algorithm that can be used for both, classification and regression tasks. It works by creating a tree-like model of decisions based on the features of the data.

The tree is made up of decision nodes and leaf nodes. The decision nodes ask a question about the data, and the leaf nodes provide a prediction or classification. The tree is constructed by repeatedly asking questions that partition the data until each group is pure, meaning that all the data in the group belongs to the same class.[2]

The result is a tree of decisions that can be used to predict the class of something based on its features. The tree above is the tree obtained by our algorithm using the previously named parameters.

In the work, once the cross-validation for the decision tree has been carried out, we obtain that the best depth for the tree is 6, therefore, when carrying out the decision tree definition, we indicate as a parameter that the maximum depth should be 6.

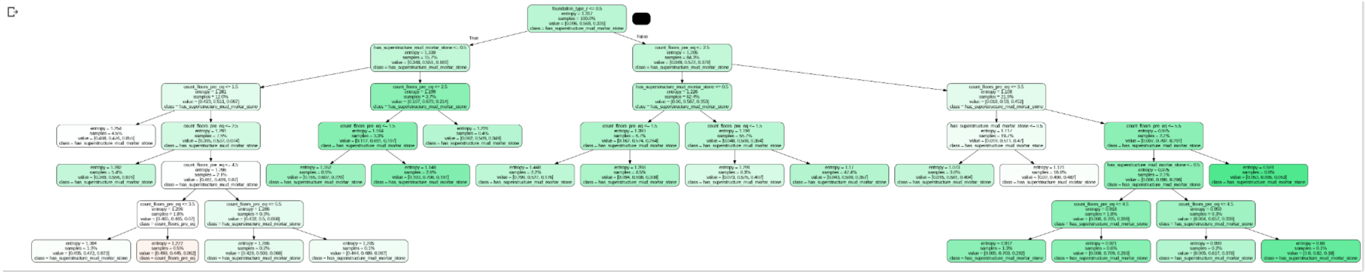
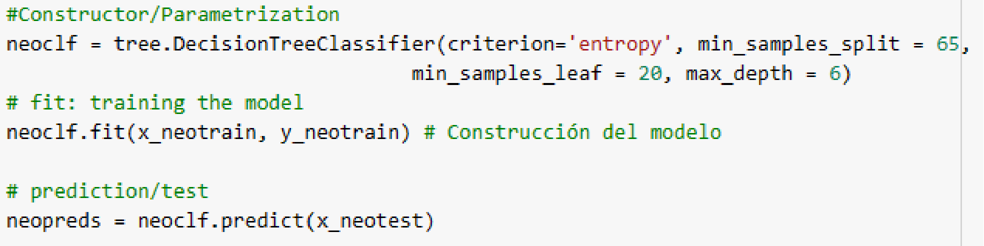

foundation_type_r <= 0.5
entropy = 1.317
samples = 100.0%
value = [0.096, 0.568, 0.335]
class = has_superstructure_mud_mortar_stone
True
False
has_superstructure_mud_mortar_stone <= 0.5
entropy = 1.339
samples = 15.7%
value = [0.348, 0.551, 0.101]
class = has_superstructure_mud_mortar_stone
count_floors_pre_eq <= 2.5
entropy = 1.205
samples = 84.3%
value = [0.049, 0.572, 0.379]
class = has_superstructure_mud_mortar_stone
count_floors_pre_eq <= 2.5
entropy = 1.199
samples = 3.7%
value = [0.107, 0.679, 0.214]
class = has_superstructure_mud_mortar_stone
has_superstructure_mud_mortar_stone <= 0.5
entropy = 1.226
samples = 62.4%
value = [0.06, 0.587, 0.353]
class = has_superstructure_mud_mortar_stone

Figure 6: Tree

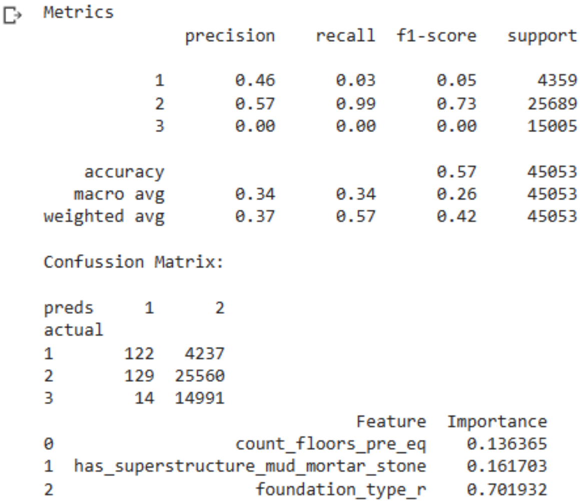


```
#Constructor/Parametrization
neoclf = tree.DecisionTreeClassifier(criterion='entropy', min_samples_split = 65,
                                     min_samples_leaf = 20, max_depth = 6)
# fit: training the model
neoclf.fit(x_neotrain, y_neotrain) # Construcción del modelo

# prediction/test
neopreds = neoclf.predict(x_neotest)
```

Figure 7: Tree Code

The matrix shown in figure 8, is the result of the predictions of the decision tree algorithm, it can be seen that the tree model is not able to predict any result out of 3, therefore and although it achieves an accuracy of 0.57 (very similar to that of the KNN) it can be seen that it is clearly worse.



```
Metrics
              precision    recall  f1-score   support

           1       0.46      0.03      0.05      4359
           2       0.57      0.99      0.73     25689
           3       0.00      0.00      0.00     15005

    accuracy                           0.57     45053
   macro avg       0.34      0.34      0.26     45053
weighted avg       0.37      0.57      0.42     45053

Confussion Matrix:

preds     1      2
actual
1       122   4237
2       129  25560
3        14  14991
                              Feature  Importance
0                  count_floors_pre_eq    0.136365
1  has_superstructure_mud_mortar_stone    0.161703
2                    foundation_type_r    0.701932
```

Figure 8: Matrix Tree

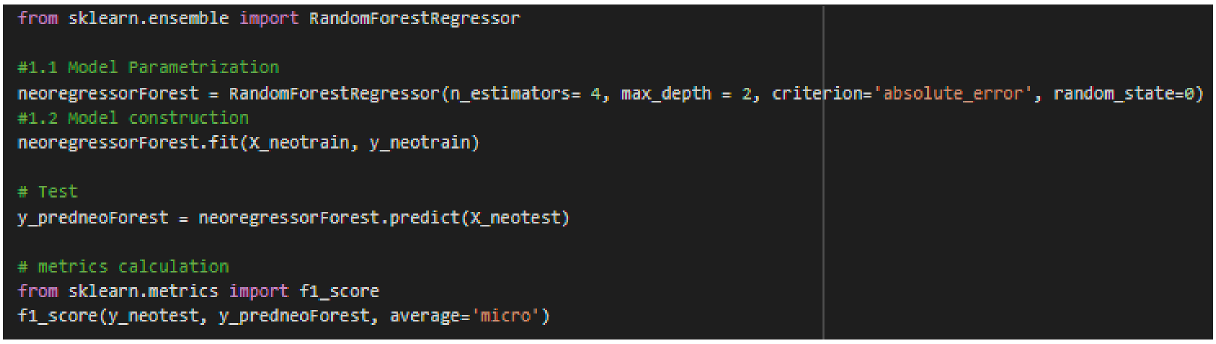
3.4 Random forest

Random forest is an algorithm that can be used for classification or regression tasks. It works by constructing a ensemble of decision trees and making a prediction based on the majority vote of the individual trees.

Here's how the algorithm works:

- Select a random sample of data points from the training set.
- Build a decision tree using the sample.
- Repeat steps 1 and 2 a specified number of times, creating a separate decision tree for each iteration.
- For a new data point, make a prediction using each decision tree and assign the new data point the label that is predicted most frequently.

In figure 9 we can see the code used to perform this algorithm, in which we set the number of estimates to 4 and the maximum depth to 2.
Later, once the algorithm is done, we calculate the F1-score to know the quality of our algorithm, which is 0.5779 (this is the best result we have been able to obtain).



```
from sklearn.ensemble import RandomForestRegressor

#1.1 Model Parametrization
neoregressorForest = RandomForestRegressor(n_estimators= 4, max_depth = 2, criterion='absolute_error', random_state=0)
#1.2 Model construction
neoregressorForest.fit(X_neotrain, y_neotrain)

# Test
y_predneoForest = neoregressorForest.predict(X_neotest)

# metrics calculation
from sklearn.metrics import f1_score
f1_score(y_neotest, y_predneoForest, average='micro')
```

Figure 9: Random forest Code

Random forest shows us different results to those obtained by the Decision Tree, the data provided by this algorithm are somewhat better than the decision tree as it correctly predicts both the values of 1.00 and 2.00, although it does not predict any value of 3.00, this is the algorithm that has given us better results as most of the values in the data are 2.00 and as has been said it also predicts those of 1.00.

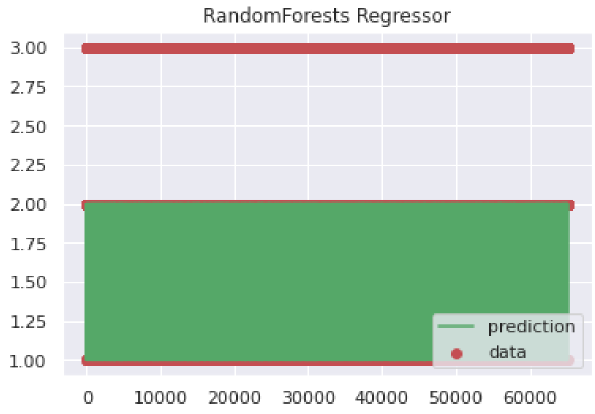


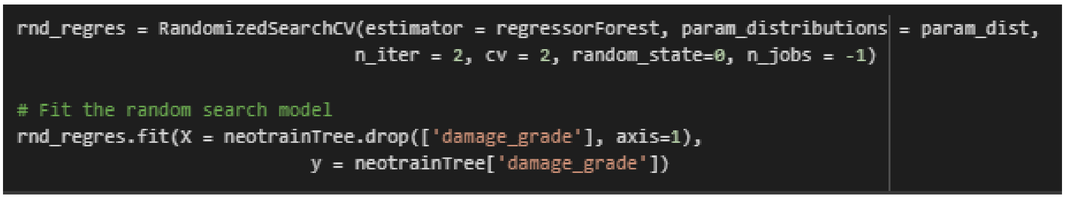
Figure 10: Graph Random forest

3.5 Hyperparameters Optimization

Hyperparameter optimization is the process of selecting the best values for the hyperparameters of a machine learning model. This is important because the performance of a model can depend significantly on the choice of hyperparameters. If the hyperparameters are set poorly, the model may underperform or may not even converge. On the other hand, if the hyperparameters are set well, the model can achieve improved performance.

There are several approaches to hyperparameter optimization, including manual tuning, grid search, and random search. In manual tuning, the practitioner selects the hyperparameter values based on their domain knowledge and experience. Grid search involves specifying a list of values for each hyperparameter, and the algorithm searches through the combinations of these values to find the best combination. Random search involves sampling random combinations of hyperparameter values and selecting the best combination based on performance.

When making the hyperparameter in our project we have had to use slightly lower values than normal because with the amount of data we have, if both the number of iterations and the number of cross-validation is higher, the time it takes to run the algorithm is too excessive, reaching more than five hours without results.

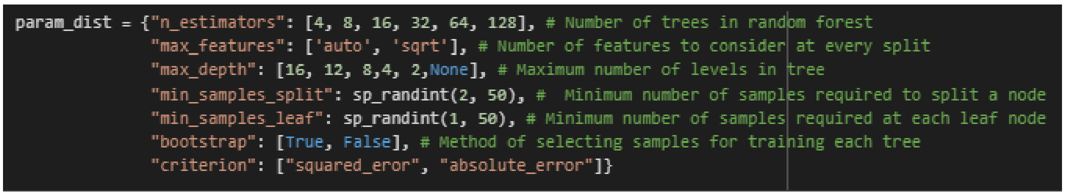


```
rnd_regres = RandomizedSearchCV(estimator = regressorForest, param_distributions = param_dist,
                               n_iter = 2, cv = 2, random_state=0, n_jobs = -1)

# Fit the random search model
rnd_regres.fit(X = neotrainTree.drop(['damage_grade'], axis=1),
                            y = neotrainTree['damage_grade'])
```

Figure 11: Hyperparameter Code

Below, you can see the hyperparameter we have used to perform the RandomizedSearchCV algorithm:



```
param_dist = {"n_estimators": [4, 8, 16, 32, 64, 128], # Number of trees in random forest
              "max_features": ['auto', 'sqrt'], # Number of features to consider at every split
              "max_depth": [16, 12, 8,4, 2,None], # Maximum number of levels in tree
              "min_samples_split": sp_randint(2, 50), #  Minimum number of samples required to split a node
              "min_samples_leaf": sp_randint(1, 50), # Minimum number of samples required at each leaf node
              "bootstrap": [True, False], # Method of selecting samples for training each tree
              "criterion": ["squared_eror", "absolute_error"]}
```

Figure 12: Hyperparameter features

As we can see in the graphical representation of the Hyperparameter it is even worse than in the case of KNN and decision tree, since in this representation there are very few predicted data out of all possible data. This is perhaps due to the aforementioned fact that we had to use somewhat lower parameters than normal.

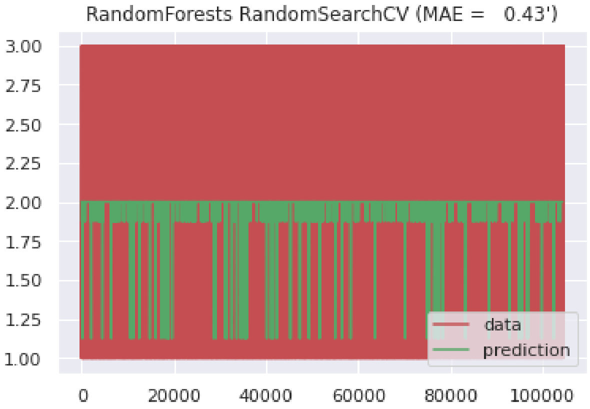


Figure 13: Graph Tree Hyperparameter

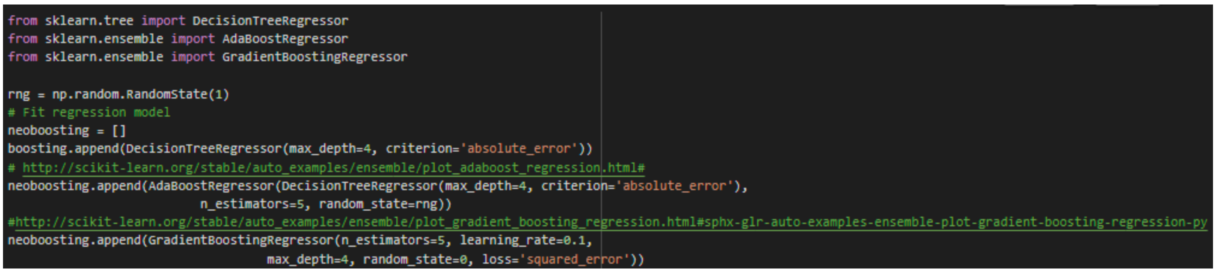
3.6 Boosting

Boosting is a machine learning technique that is used to improve the accuracy of a model by combining the predictions of multiple weaker models. It is a type of ensemble learning, which is a method that combines the predictions of multiple models to make more accurate predictions.
In boosting, weak models are trained sequentially, with each model trying to correct the mistakes made by the previous model. The final prediction is made by combining the predictions of all the weak models. Boosting can be used with any type of model, but it is most commonly used with decision trees.

There are several popular boosting algorithms, including AdaBoost, Gradient Boosting, and XGBoost. AdaBoost works by weighting the observations in the training set based on the accuracy of the previous prediction. Observations that were misclassified receive a higher weight so that the next classifier focuses more on them.

Gradient Boosting is another popular boosting algorithm that works by fitting weak models to the residual errors made by the previous model in the sequence. It is a more powerful technique than AdaBoost and is often used in competition winning solutions.

Figure 14 shows the number of algorithms that can be used in Bosting. In this case we use 3. We start with the Decision Tree algorithm indicating a maximum depth of 4, the next one we use is the AdaBoosting algorithm which uses Decision Tree with a maximum depth also of 4 and using 5 estimates. Finally, we used the Grandient Boosting algorithm which uses the same parameters as AdaBoosting, although we may have used low parameters because we did not want the execution time to be too high and thus obtain results as was the case with the radnom forest.



```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor

rng = np.random.RandomState(1)
# Fit regression model
neoboosting = []
boosting.append(DecisionTreeRegressor(max_depth=4, criterion='absolute_error'))
# http://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_regression.html#
neoboosting.append(AdaBoostRegressor(DecisionTreeRegressor(max_depth=4, criterion='absolute_error'),
                          n_estimators=5, random_state=rng))
#http://scikit-learn.org/stable/auto_examples/ensemble/plot_gradient_boosting_regression.html#sphx-glr-auto-examples-ensemble-plot-gradient-boosting-regression-py
neoboosting.append(GradientBoostingRegressor(n_estimators=5, learning_rate=0.1,
                                  max_depth=4, random_state=0, loss='squared_error'))
```

Figure 14: Boosting Code

Figure 15 shows two types of algorithms, AdaBoost and Gradient Boosting, where it can be seen that the two algorithms provide very different data, AB only generates data from 1.00 to 2.00 removing some values that manage to reach 3.00 and GB only provides a very small range of values, so the better algorithm of the two is AdaBoost. The Decision tree graph is not shown, but it is not shown again as an example is shown in previous figures.

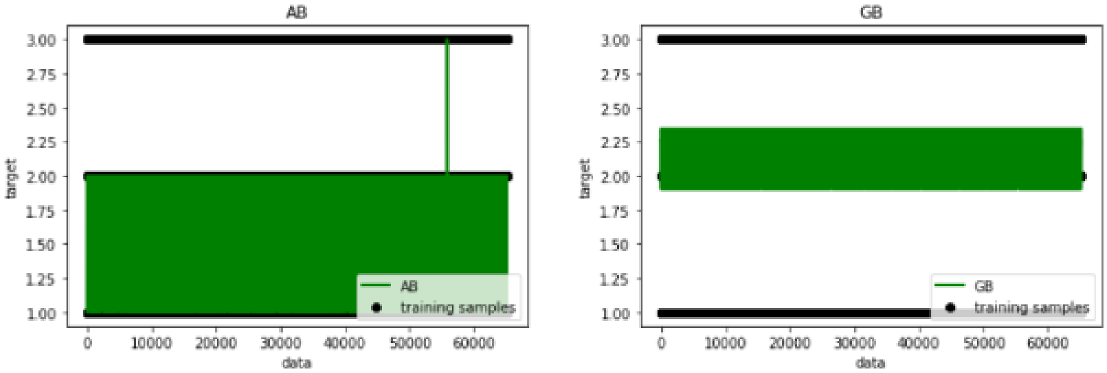


Figure 15: Graph Boosting AB and GB

References

[1] DRIVENDATAZ. Richter's predictor: Problem description. URL https://www.drivendata.org/competitions/57/nepal-earthquake/page/136/#description-1.

[2] Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020. URL https://www.researchgate.net/profile/Batta-Mahesh/publication/344717762_Machine_Learning_Algorithms_-A_Review/links/5f8b2365299bf1b53e2d243a/Machine-Learning-Algorithms-A-Review.pdf?eid=5082902844932096.