Assignment 2 – Regression using Scikit-learn

Student Name: Robert Walsh **Student ID**: 18103802 **Programme**: 4BCT

Algorithm 1: Random Forest Regressor – Overview & Detailed Description

Random Forest Regressor is one of the most popular algorithms in machine learning because it's strong, stable, and works well on many different types of datasets. The idea behind it is that instead of relying on one decision tree, it builds many of them. Each tree is trained on a random sample of the training data (which is called bootstrapping, where rows are picked with replacement) so every tree ends up seeing a slightly different version of the dataset [1].

When a tree is being built, it doesn't look at all the features at every split. Instead, it only considers a random subset of features. The randomness (random rows & random features) helps ensure that trees don't all learn the exact same patterns leading to overfitting. Because the trees are less correlated, the final model avoids overfitting and generalises better to new and unseen data.

During training, each decision tree evaluates possible splits by checking how well they reduce the variability of the target values in the resulting child nodes. For regression, this is usually done using Mean Squared Error (MSE), where the tree tests different feature thresholds and selects the split that produces child nodes with the lowest combined MSE. Lower MSE means the target values within each node are more tightly clustered which indicates the split has successfully reduced variance in the data. The tree continues splitting in this way, always choosing the feature and threshold that reduces variance the most until a stopping rule is met.

When it comes to making predictions, the process is as follows - a new input sample is sent through every tree in the forest. Each tree produces its own continuous prediction. The Random Forest then takes the average of all these predictions. This averaging effect is powerful because it reduces the impact of noisy or unreliable trees. If one tree makes a bad prediction, the other trees usually cancel it out, which makes the overall model more accurate and stable.

Why I choose this algorithm.

I chose Random Forest Regressor mainly because I wanted it to contrast with my second algorithm (KNN). Random Forest is very different in how it learns & it can capture complicated, non-linear relationships between the features without needing any scaling or heavy preprocessing. It handles mixed feature ranges naturally because tree splits are not affected by the scale of the values. Random Forest is strong at learning global patterns in the data. It looks at the bigger structure of the dataset and can find important interactions/correlationa between features. However, it doesn't always perform as well when the relationship between inputs and outputs is very local or based on small neighbourhood patterns. That is the type of situation where KNN tends to perform better. By choosing two very different algorithms, one tree based and one distance based, I felt I could get a better overall understanding of the steel dataset. Each algorithm has strengths where the other one is weaker, so comparing them should give more meaningful insights than choosing two very similar models.

Hyperparameter Details for Tuning.

Hyperparamer 1:

n_estimators

As it is one of the most important hyperparameters in a Random forest model. It controls the number of individual trees that make up the forest. Increasing this number generally improves performance because more trees mean more averaging, which usually reduces variance and creates more stable/accurate predictions. However, too many trees can increase training time without providing much extra benefit. Because it directly influences the strength and stability of the model, it is a beneficial hyperparameter to tune and compare.

Hyperparamer 2:

max_depth

Because it limits how many splits a tree can make from top to bottom. If trees are allowed to grow too deep they can memorise the training data and overfit badly. If the depth is too shallow each tree becomes too simple and underfits. This hyperparameter directly controls the model's bias/variance balance. Deeper trees lower bias but increase variance, while shallower trees do the opposite. Because of this, max_depth has a major effect on model behaviour and is an ideal hyperparameter to tune in order to observe overfitting/underfitting patterns.

Algorithm 2: KNN Regressor – Overview & Detailed Description

K-Nearest Neighbours, KNN is a simple but widely used algorithm in machine learning and can be used for both classification and regression problems/tasks. In regression, the aim is to predict a continuous target value for a new input by looking at the target values of its K closest neighbours in the training data. The prediction is just the average of those neighbour targets. The basic idea behind this is that datapoints that are close to each other in feature space should usually have similar outcomes. To measure how close points are, KNN uses distance metrics. The two most common ones are Euclidean distance and Manhattan distance, but there are other options too if the dataset demands it [4]. The choice of distance metrics can change how the algorithm behaves, especially when the input features have very different scales.

Distance plays a central role because the algorithm relies on comparing how similar each training point is to the new sample. In this assignment, Euclidean distance was used which measures the straight line distance between two points in multi dimensional space. For two data points with features $x\_1, x\_2, \ldots, x\_n$, the Euclidean distance is $d = (x1 - x1\text{^}')2 + (x2 - x2\text{^}')2 + \cdots + (xn - xn\text{^}')2$

The model computes this distance between the new sample and every row in the training set and then selects the K closest points. Because KNN is a regression algorithm in this case, the final prediction is made by averaging the target values of these K neighbours. If K is too small, the model becomes very sensitive to noise (high variance), but if K is too large, it smooths the data too much and can underfit (high bias).

One of the main hyperparameters in KNN is K, which decides how many neighbours we look at before making a prediction. This hyperparameter is essential for KNN to work & without a value for K, the algorithm has no definition of what a neighbourhood even is. If the user does not specify K, many libraries including scikit-learn will default to K= 5. When new data is introduced the algorithm calculates the distance from that new point to every point in the training set and identifies the K smallest distances and then averages their target values. The algorithm compares the feature values of the new data point to the training data.

Since KNN is completely dependent on distance calculations, preprocessing is a crucial step, especially when feature values are on different scales. In the steel dataset, features like normalising temperature can be in the hundreds, while percentages of chemical elements range from 0 to 1. Without scaling, the large-range features (like temperature) would dominate the distance metric and the algorithm would likely ignore the smaller-range features such as carbon or sulphur percentages. To prevent this, we apply StandardScaler() or another scaling method so that all features contribute fairly/equally to the distance calculations. Doing this makes the comparisons meaningful and avoids model bias towards high magnitude features.

Why I choose KNN Regressor.

I chose KNN Regressor as my second algorithm because it is very different from RandomForestRegressor, both in how it learns and in the types of patterns it tends to capture. KNN is a distance based method that works directly with numerical features and it generally performs well on small or medium-sized datasets which fits the steel data. It does not require any encoding because all of the features in this dataset are already numeric.

One of the main reasons for choosing KNN is that it has strengths where Random Forest tends to struggle. Random Forest is an ensemble of decision trees and is good at picking up complex global patterns and interactions within features but it does not always perform well when the relationship between inputs and outputs is very local or smooth. KNN works in almost the opposite way where it makes predictions based on local neighbourhoods in the data so it can capture local structure more effectively & especially after scaling.

Using two algorithms that are fundamentally different gives a more meaningful comparison than using two similar ones. Random Forest relies on many deep trees that produce piecewise splits, while KNN relies on distances and produces smoother more continuous predictions. This contrast makes it easier to analyse differences in bias and variance & to compare how each model reacts to scaling and to understand which type of structure the steel dataset supports better. Overall, the two algorithms should balance each other's weaknesses and provide clearer insights when evaluated side by side.

Hyperparameter Details for Tuning KNN Regressor.

Hyperparamer 1:

n_neighbors

K controls how many neighbours are considered when making a prediction. It is the main hyperparameter in KNN and can significantly affect model performance. If K is too small, the model becomes very sensitive to noise and can overfit. If K is too large, the model becomes too smooth and may underfit because it averages across too many points. Tuning K helps find a balance between these two extremes so it makes sense to include it in the hyperparameter search.

Hyperparamer 2:

p

The p hyperparameter determines which distance metric the model uses. When p = 2, KNN uses Euclidean distance, and when p = 1, it uses Manhattan distance. The choice of distance metric has a direct impact on how the algorithm identifies the closest neighbours, especially when the features

vary in how they scale or interact. Since KNN is entirely dependent on distances, tuning p can lead to meaningful differences in performance and is a suitable hyperparameter to tune for this dataset.

Algorithm 1 – Random Forest Regressor - Model Training, Evaluation & Result Discussion

The Random Forest Regressor model was executed on the steel data set and without the use of hyperparametrs, yielded good results as can be seen in the table below.

RandomForest Regressor (No Hyperparameters)	
Avg Train RMSE	10.934
Avg TEST RMSE	28.276
Avg TRAIN R2	0.986
Avg TEST R2	0.896

As you can see, RMSE has been chosen as a domain independent error metric while R^2 has been chosen as a domain specific error metric measure. RMSE is a commonly used metric in regression tasks especially when predictions are continuous numerical values [5].

In this instance, the algorithm has used 10 cross fold validation, where it shuffles the data, keeping rows as a complete unit, trains on 90% of the entire data set while reserving the other 10% for training. Each itteration of this will produce a RMSE score, and the average of these scores becomes our final target prediction.

RF Regressor is a robust algorithm that performs "out of the box" very well and can be quite insensitive to hyperparameteers. For the additional model trained and tested after our baseline default model, the hyperparameters stated above were introduced.

[n_estimators; $\mathbf{50, 100, 200}$ **&** $\mathbf{300}$ **] [max_depth;** $\mathit{None}, \mathbf{5, 10, 15, 20}$ **]**

The results showed that there was a very minor increase in performance, as can be seen below (mention the impact based on such a small dataset, this really makes tuning less impactful) plus all features being continuous and relevant with little to no noise. Plus the initial results were already extremely strong.

Best Choice for Hyperparameters	
Random Forest Regressor (Tuned)	
AVG TRAINS RMSE	10.667 (10.934)
AVG TEST RMSE	28.071 (28.276)
AVG TRAIN R2	0.986
AVG TEST R2	0.897 (0.896)

The results show that hyperparameter tuning produced only a very minor improvement in performance. The values in red represent the baseline model using default settings. After tuning, RMSE decreased slightly for both the training and test sets, indicating a marginal reduction in error. The R^2 score on the test set remained essentially unchanged meaning the model explained the same amount of variance before and after tuning. This minimal change is expected when using a Random Forest Regressor as the algorithm is naturally robust and tends not to be highly sensitive to

moderate hyperparameter adjustments, especially on a relatively small and structured dataset like this one.

To further validate the observation that Random Forest performance remained largely unchanged after tuning, a second hyperparameter search was conducted using a different pair of hyperparameters (**max_features** and **min_samples_split)**. This second run produced results that were almost identical to the first grid search which confirmed the conclusion that the model is relatively insensitive to moderate hyperparameter adjustments, on this dataset at least.

A grid search was again used to evaluate all combinations of the selected hyperparameters, and the results was plotted for a visual inspection. The plotted graph is included in the appendix (see *Image 1*).

Algorithm 2 – KNN Regressor - Model Training, Evaluation & Result Discussion

For the KNN model, the training process was quite simple because KNN doesn't actually learn in the normal sense. The model just stores the scaled dataset and waits until prediction time to compare distances. A pipeline was used with StandardScaler so that all features were on the same scale, since the steel dataset mixes temperatures in the 100's with chemical values much closer to 0. Without scaling, distance calculations would be completely dominated by the larger numerical features.

The baseline KNN results showed a training RMSE of around 34.31 and a test RMSE of about 42.66. The R2 scores were 0.858 for training and 0.766 for testing, see table below. These results suggest that the model can pick up some of the structure in the data but there is a clear drop in performance when moving from the training folds to the unseen folds in cross-validation. The drop in R2 also shows that the model loses some explanatory power when predicting new data. This is not surprising for KNN since its predictions depend fully on local neighbourhoods. If the dataset has noise or cases where similar feature values lead to very different strength outputs, which is the case, KNN can struggle because it just averages neighbours rather than learning a more general pattern.

KNN Regressor (Baseline model, no hyperparameters)	
AVG TRAIN RMSE	34.308
AVG TEST RMSE	42.656
AVG TRAIN R2	0.858
AVG TEST R2	0.766

These baseline results indicate the algorithm is able to explain about 76% of the variance in the target variable on unseen/new data which is decent for a distance based method.

The two main hyperparameters for KNN weere then tuned using the number of neighbours (n_neighbors) and the distance metric (p). After running a grid search with 10-fold cross-validation, the best combination was K = 3 with Manhattan distance (p = 1). These settings gave slightly better results: train RMSE dropped to 29.38, and test RMSE dropped to 42.49, with small increases in both R2 values. While these improvements were an improvement, they were small and didn't change the overall behaviour of the model. This more or less confirms that KNN hits a performance ceiling on this dataset. Even with tuning, it cannot capture the more global and non-linear relationships as well as a tree-based approach can.

See image below for results.

KNN Regressor (Best Hyperparmeters, n_neighbours : 3 , p : 1)	
AVG TRAIN RMSE	29.381 (34.308)
AVG TEST RMSE	42.495 (42.656)
AVG TRAIN R2	0.896 (0.858)
AVG TEST R2	0.771 (0.766)

Overall, the KNN model worked and produced reasonable predictions, but it wasn't as strong when it came to generalising to unseen data. Its baseline performance was decent, and tuning helped a little, but because the algorithm relies so heavily on local distances, it seems limited by the type of patterns present in the steel dataset.

Conclusions

Overall, the results showed a clear difference between the two algorithms. Random Forest performed best, with consistently lower RMSE and higher R^2 scores on both the training and test folds. It generalised well and showed no signs of serious overfitting, which makes sense because the trees in the forest average each other out. The tuned model ended up with n_estimators = 200 and max_depth = None, which worked well and didn't need much fine tuning.

KNN also worked on the dataset but it showed more signs of underfitting and didn't generalise as well. Even after tuning K and the distance metric, the improvements were small. KNN was more sensitive to hyperparameters than Random Forest, especially the value of K but even the best setting (K = 3, Manhattan distance) could not match the performance of the Random Forest model. This is likely because KNN depends heavily on local distances while the steel dataset seems to involve non-linear relationships between features.

In summary, Random Forest was the more suitable algorithm for this task. It handled the structure of the data better, showed stable performance across folds, and required less effort to tune. KNN produced reasonable predictions but was overall less effective and more limited in how much of the dataset's patterns it could capture.

Recommended Hyperparameter

KNN Regressor	
Hyperparameter	**Value**
n_neighbors	3
P	1

Random Forest Regressor	
Hyperparameter	**Value**
n_estimators	300
max_depth	None

References

1. https://www.geeksforgeeks.org/machine-learning/random-forest-algorithm-in-machine-learning/
2. https://www.geeksforgeeks.org/machine-learning/random-forest-regression-in-python/
3. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html
4. https://www.geeksforgeeks.org/machine-learning/k-nearest-neighbors-knn-regression-with-scikit-learn/
5. https://www.geeksforgeeks.org/machine-learning/regression-metrics/
6. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
7.

Report Update Log

Date	Update Description	Time Spent
08/11/25	Prepped template, changing headings and removing filler content	0.5 hours
11/11/25	Populated the template with Random Forest Regressor overview, description and ran the initial code without hyperparameters to review results prior to implementing hyperparameters.	2 Hours
18/11/25	Completed testing and training with Random Forest Regressor model, testing both the baseline model and the tuned model. Plotted the results in a graph, discussed the training, reults and evaluated the performance.	3 Hours
20/11/2025	KNN Progress. Ran with baseline model using default values for hyper params and ran model with tuned hyperparams	3 Hours
25/11/2025	KNN updates with evalution and discussion of results.	3 Hours

Images

Image 1

	param_n_estimators	param_max_depth	mean_test_rmse	mean_test_r2	rank_test_rmse
3	300	None	28.070871	0.897040	1
19	300	20	28.071908	0.897034	2
15	300	15	28.108690	0.896832	3
18	200	20	28.108866	0.896982	4
2	200	None	28.109203	0.896978	5
14	200	15	28.155930	0.896749	6
1	100	None	28.275983	0.895853	7
17	100	20	28.279979	0.895851	8
13	100	15	28.317277	0.895540	9
11	300	10	28.485846	0.894018	10
10	200	10	28.525568	0.893996	11
0	50	None	28.617331	0.893565	12
16	50	20	28.623550	0.893559	13
12	50	15	28.633997	0.893311	14
9	100	10	28.660280	0.893010	15
8	50	10	28.922321	0.891402	16
7	300	5	37.578810	0.815178	17
6	200	5	37.645658	0.814823	18
5	100	5	37.865314	0.812716	19
4	50	5	37.963800	0.811733	20

Output from RF regressor to evaluate the values from all instances and permutations of the chosen hyperparameters,

- max_depth = “None”
- n_estimators = 300

None being no limit on how large the tree can grow.