Assignment 2 – Regression using Scikit-learn

Student Name: Ayodeji Ali **Student ID**: 20343733 **Programme**: 4BCT

Algorithm 1 – Support Vector Regression

Support Vector Regression (SVR) is an extension of Support Vector Machines. It is useful for resolving regression issues. It optimizes a function by identifying a tube that approximates a continuous-valued function while minimising prediction error. SVR penalizes predictions that are far from the expected output using an e-insensitive loss function, with the width of the tube controlled by e.

Detailed Description of Support Vector Regression

The Support Vector Regressor (SVR) model is an extension of Support Vector Machines (SVM) used for regression tasks. SVR does not recursively split data, instead it tries to fit a function within a margin of the tolerance to capture the relationship between its features and its target values. This margin is known as the epsilon-tube. This is where errors are not penalized. The model's aim is to find the optimal hyperplane that best represents the data while minimizing the margin errors outside this tolerance zone.

SVR achieves this through the kernel trick, where it can handle non-linear data by mapping it into higher-dimensional space using different kernel functions. Kernels that re used the most are linear, polynomial, and radial basis function (RBF). The model then searches for a hyperplane that minimizes the margin error by optimizing for a set of support vectors. These data points are the closest ones to the hyperplane that influences the final model. Like in decision trees, the model transverses through features and tries to find the best fit, although SVR is more complex as it relies on a mathematical margin rather than discrete splits.

The model has two primary hyperparameters: C and epsilon (e). C controls the trade-off between achieving a flat, simple model and correctly predicting as many points as possible outside the margin. A high C value allows for a more complex model that tries to predict all points more accurately, possibly leading to overfitting. Epsilon sets the width of the epsilon-tube within which no penalty is given for errors, affecting the model's sensitivity to noise. Like information gain in decision trees, SVR relies on optimization of the support vectors to minimize error, but without backtracking, which can sometimes prevent it from achieving perfect optimization in all cases.

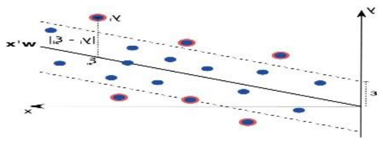


Figure 1: SVR Illustration of how it works

Why I choose this algorithm.

The reason I chose Support Vector Regressor (SVR) is because it has high level flexibility by using the kernel trick. This allows the model to handle linear and non-linear relationships within the data. This flexibility makes SVR useful when there's a combination of simple and complex interactions between features. What I found appealing about SVR is how it prioritizes the support vectors, the most influential points in the data, to define the model's boundaries. This makes it more resilient to noise. SVR also gives a good balance between simplicity and accuracy, which makes it possible to tune the model to generalize well. It's more complex than a basic linear model, but the kernel functions allow it to fit more challenging patterns, making it a good choice to achieve accurate predictions without an overly complex structure.

Hyperparameter Details for Tuning.

Hyperparamer 1: This would be *C* which controls trade-off between producing a smooth decision boundary and correctly classifying training points.

Hyperparamer 2: This would be *Epsilon* which defines the margin of tolerance where errors are not penalized.

Algorithm 2 – Gradient boosting Regression

Gradient boosting Regression is a variant of ensemble methods where the model creates multiple weak models and combine them to get better performance.

Detailed Description of Algorithm 2.

The Gradient Boosting Regressor (GBR) method is an ensemble learning method that uses multiple decision trees in a sequence, each tree corrects the errors made by the previous ones. Unlike standalone decision trees that independently classify or predict data, GBR uses a series of weaker learners that are usually shallow decision trees, that each focus on the errors of the previous trees. It creates a strong predictive model through incremental improvements.

The GBR algorithm works by minimizing the loss function, which measures how well or poorly the model fits the data. Each new tree attempts to decrease the model's residual error by focusing on the examples where previous trees performed poorly, optimizing the loss reduction in each step. The process continues for a specific number of boosting rounds or until adding more trees provides little to no performance benefit. The goal is to achieve a balance where each step corrects for the errors of the previous models, without overfitting.

GBR has two main hyperparameters: learning rate and n_estimators. Learning rate controls the contribution of each tree to the final model; lower values make the model more robust but may require a larger number of trees. n_estimators is the number of boosting rounds or trees to add to the model; more estimators improve accuracy but can lead to overfitting if not regulated by the learning rate. Like information gain in decision trees, GBR's sequential process ensures the model improves over each iteration, but without backtracking. Once added, each tree's adjustments are fixed, making it crucial to set hyperparameters carefully to balance performance and prevent overfitting.

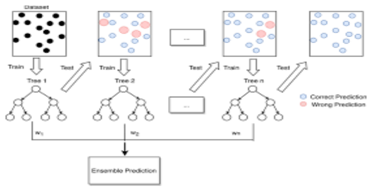


Figure 2: Illustration of how Gradient boosting regression works

Why I choose this algorithm.

I chose Gradient Boosting Regressor (GBR) because of its incremental approach. It improves the model little by little by correcting its own mistakes with each new tree it adds. This makes GBR quite powerful for capturing complex, non-linear relationships, as it creates an ensemble of decision trees that combine to create a more accurate prediction. I like how GBR focuses on the residual errors and continues to refine the model, which means it's naturally equipped to handle difficult datasets that other models might struggle with. It may be more resource-intensive to train, but the accuracy and robustness it achieves make it well worth the effort. With GBR, the model is constantly improving, but I still have control over the process with hyperparameters like the learning rate and the number of estimators, making it a great choice for anyone who wants a solid balance between accuracy and interpretability.

Hyperparameter Details for Tuning.

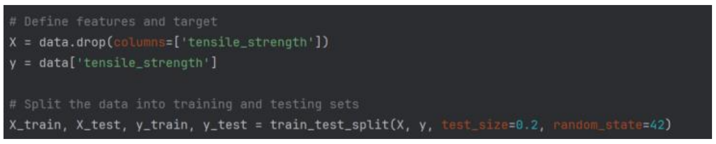
Hyperparamer 1: This would be *Learning Rate* which controls the contribution of each tree to the overall model.

Hyperparamer 2: This would be *n_estimators* which specifies the number of trees (iterations) to add to the ensemble.

Algorithm 1 - - Model Training and Evaluation

Data Preprocessing and Visualisation

Data Preprocessing
I split the dataset into features (X_Train, X_Test) and the target variable (Y_Train, Y_Test). The features included multiple parameters that could affect tensile strength, which was the target variable. Splitting the data allowed the model to learn from the training set (X_Train and Y_Train) and then predict the tensile strength using the test set (X_Test and Y_Test).



```python
# Define features and target
X = data.drop(columns=['tensile_strength'])
y = data['tensile_strength']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 3: Code snippet showing how the data was split for training and testing.

By separating the data, the SVR model was able to identify relationships between the features and tensile strength without overfitting to the data.

Visualization
To evaluate the performance of the SVR model, I used visualizations to compare errors and explore the effect of hyperparameters:
I plotted a bar chart to show the Mean Squared Error (MSE) for both the default and tuned SVR models, using both training and test data. This allowed me to see how adjustments in hyperparameters influenced the accuracy of the model.

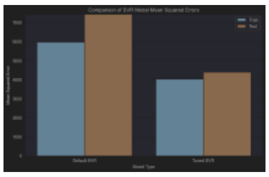


Figure 4: Bar chart displaying the training and testing errors of the SVR model before and after tuning.

I visualized the impact of changing the C and epsilon parameters by using a heatmap. This heatmap provided a clear view of how different combinations of hyperparameters affected the MSE. It made it easier to determine the optimal settings for the SVR model.

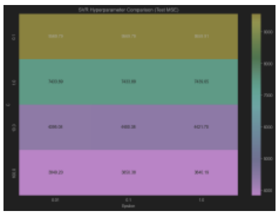


Figure 3: A heatmap showing how the MSE varied with different C and epsilon values.

Training and Evaluation Details

Training

The default SVR used its standard parameters, with no adjustments to control complexity. The model was not tuned to capture complex patterns, this resulted with the ability to fit the training data but with limitations in generalization.

In the tuned model, the parameters C and epsilon were adjusted to better handle the complexities in the data better. This make the model more accurate while still being sensitive to errors.

Evaluation

The default SVR struggled with generalization, it showed a big enough gap between the training and test performance. This shows that the model captured some patterns but failed to handle unseen data effectively, this means the model was underfitting the data.

The tuned SVR, showed a big improvement with the test performance, it was better at capturing the underlying relationships but doesn't overfit. It achieved a balance between training and test performance, showing more of a robust model that generalizes effectively. The increased R^2 for the tuned model shows an improvement in the ability to explain the variance in the data and reduced MSE making less prediction errors altogether.

Discussion of results

The default SVR model demonstrated moderate predictive ability, with a lower training R^2 of 0.25 and a test R^2 of 0.24. This indicated that the model captured some underlying trends but struggled with generalization, suggesting underfitting. The Mean Squared Error (MSE) was relatively high for both training and test sets, reflecting prediction inaccuracies.

After tuning, the SVR's performance improved significantly. The training R^2 increased to 0.49, and the test R^2 rose to 0.55, indicating better alignment with the data patterns. The reduced MSE in the tuned model demonstrated fewer prediction errors, suggesting that adjusting hyperparameters enhanced the model's ability to learn and generalize without overfitting.

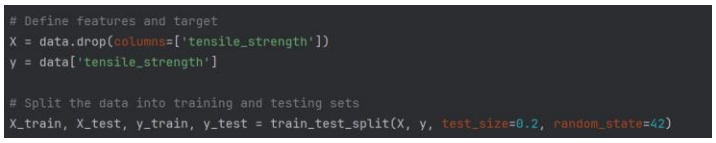
Conclusions

The tuned SVR performed better than the default model by capturing more relevant patterns and improving predictive accuracy. This shows the impact of carefully selecting hyperparameters, allowing the SVR to balance fitting the training data while generalizing well to unseen data.

Algorithm 2 – Support Vector Regression - Model Training and Evaluation

Data Preprocessing and Visualisation

Data Preprocessing
Like the SVR model, I split the dataset into X_Train and X_Test for the features and Y_Train and Y_Test for the target variable. This split ensured that the GBR model had a dedicated training set to learn from while keeping a separate test set to validate its predictions. Using a consistent data split allowed for reliable comparisons between models.



```
# Define features and target
X = data.drop(columns=['tensile_strength'])
y = data['tensile_strength']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 4: Code snippet showing how the dataset was split for the GBR model

The consistent structure in data splitting ensured that I could directly compare the results across models, focusing on the GBR's ability to predict tensile strength.

Visualization
I generated a bar chart to compare the MSE between the default and tuned GBR models. The chart displayed MSE values for both the training and testing datasets, allowing me to see the difference before and after hyperparameter tuning.

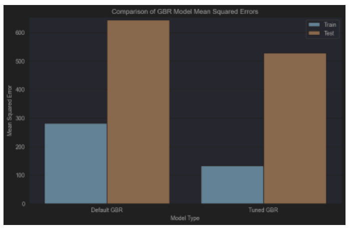


Figure 5: A bar chart showcasing the MSE of the GBR model for both training and testing datasets

I used a heatmap to explore how variations in n_estimators (number of boosting rounds) and learning_rate impacted the model's MSE. This visualization provided a clear understanding of how the model's performance changes with different parameter settings, helping to find the best configuration.

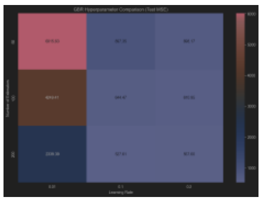


Figure 6: A heatmap illustrating the impact of n_estimators and learning_rate on the MSE

Training and Evaluation Details

Training
The default GBR model was trained without any constraints, this led to overfit the training data by capturing even minor patterns. It performed well on the training set, which is normal for GBR.

The tuned GBR had constraints set, a lower learning_rate and a controlled number of trees (n_estimators). These adjustments were made to prevent the model from overfitting, this allowd it to learn more slowly and focus on significant trends.

Evaluation
The default GBR had high training accuracy, but the gap between the training and test scores showed it was overfitting. It performed well on training data but showed a less accuracy when evaluated on the test data, it captured noise as well as the important patterns.

The tuned GBR model displayed a better generalization with more consistent training and test results. This shows that the tuned model didn't capture any unnecessary details, it focuses on the important patterns. The higher R^2 on the test data displays improved explanatory power and the more balanced MSE values shows that the model handled errors better through different data sets.

Discussion of results

The default GBR model displayed a high training accuracy with an R^2 of 0.96, but a noticeable drop in test accuracy to 0.93, this indicates a slight overfitting. The low training MSE reflected and exact fitting to the training data, but the higher test MSE shows that some noise had been captured, this reduces generalization.

Tuning the GBR improved its performance, it got an R^2 of 0.98 for training and 0.95 for testing. This shows us that the model, while still fitting the training data closely, it has managed to generalize better with less overfitting. The reduction in MSE for both training and test sets improved the predictive accuracy and handled errors better.

Conclusions

The tuned GBR minimized overfitting and increased test performance, showing the advantage of tuning hyperparameters like learning rate and tree depth. The adjustments allowed the model to maintain a high predictive power on the training set, improving accuracy on test data, confirming the benefits of model optimization.

Comparative Analysis of Algorithm Performances

Comparative Analysis of Algorithm Performances
The Decision Tree model achieved high training accuracy but overfitted, resulting in a test accuracy of 0.86. After tuning, the test accuracy improved to 0.88, showing better generalization. The Random Forest model, on the other hand, remained stable, with a test accuracy of 0.84 even after tuning.

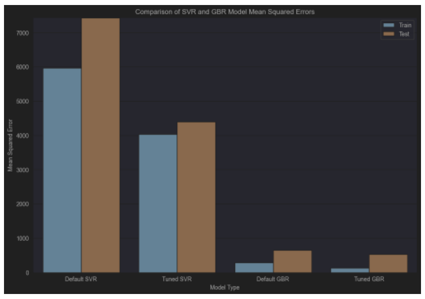


Figure 10: Graphed comparison of results from the two algorithms

This graph illustrates the test accuracy comparison, highlighting the improvement seen in the Decision Tree after tuning.

Recommended Hyperparameter Values Based on Results

Decision Tree:
max_depth=5: Helps reduce overfitting and improves generalization.
min_samples_split=10: Prevents the tree from becoming too complex.
Random Forest:
n_estimators=100: Provides a good balance between performance and efficiency.
max_depth=5: Limits overfitting and enhances generalization.

Concluding Remarks

The Decision Tree model showed clear overfitting with high training accuracy but improved its test accuracy to 0.88 after tuning. The Random Forest model maintained a stable test accuracy of 0.84. By using the recommended hyperparameters—max_depth=5 and n_estimators=100—both models can achieve better generalization. This analysis emphasizes the importance of tuning hyperparameters for improved model performance.

References

1. **Verma, N.**, 2020. *An introduction to Support Vector Regression (SVR) in Machine Learning*. Medium. Available at: https://medium.com/@nandiniverma78988/an-introduction-to-support-vector-regression-svr-in-machine-learning-681d541a829a [Accessed 31 Oct. 2024].
2. **Aggarwal, M.**, 2019. *All you need to know about Gradient Boosting Algorithm — Part 1 (Regression)*. Towards Data Science. Available at: https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502 [Accessed 31 Oct. 2024].
3. **scikit-learn**, 2024. *Support Vector Machines — SVM Regression*. Available at: https://scikit-learn.org/stable/modules/svm.html#svm-regression [Accessed 31 Oct. 2024].
4. **scikit-learn**, 2024. *sklearn.ensemble.GradientBoostingRegressor*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor [Accessed 31 Oct. 2024].

5. **scikit-learn**, 2024. *Gradient Boosting Regression Example*. Available at: https://scikit-learn.org/1.5/auto_examples/ensemble/plot_gradient_boosting_regression.html [Accessed 31 Oct. 2024].
6. **Boeckler, F.**, 2014. *Support-vector regression (SVR) Illustration of an SVR regression function*. Available at: https://www.researchgate.net/profile/Frank-Boeckler/publication/248396465/figure/fig1/AS:213987712081920@1428030054349/Support-vector-regression-SVR-Illustration-of-an-SVR-regression-function-represented.png [Accessed 31 Oct. 2024].
7. **Boeckler, F.**, 2014. *Flow diagram of gradient boosting machine learning method*. Available at: https://www.researchgate.net/figure/Flow-diagram-of-gradient-boosting-machine-learning-method-The-ensemble-classifiers_fig1_351542039 [Accessed 31 Oct. 2024].