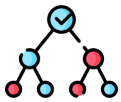
Impact of Hyperparameter tuning (Tree Depth and Number of Trees) on accuracy in Ensemble Trees

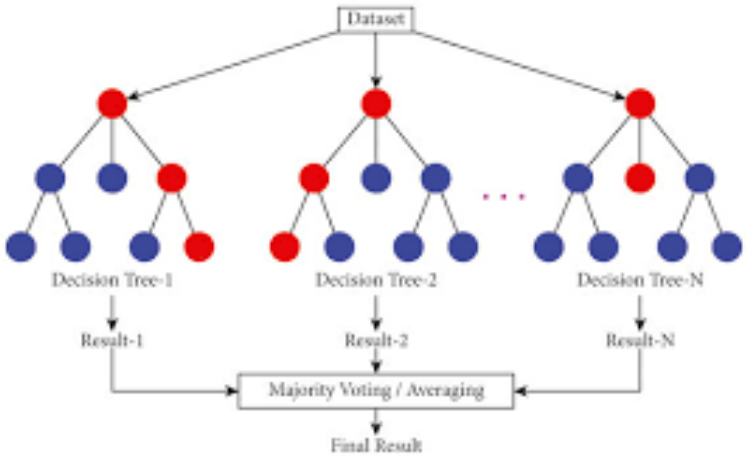
Introduction

Machine learning is a branch of artificial intelligence that focuses on enabling computers to learn patterns from data and make predictions. In supervised learning, a model is trained using a dataset containing both input features (independent variables) and a target outcome (dependent variable). The objective is to develop a function that effectively maps inputs to their corresponding outputs. The training process continues until the model reaches a satisfactory level of accuracy. Some widely used supervised learning methods for regression analysis include decision trees, random forests, k-nearest neighbors (KNN), and logistic regression.

Decision trees 

Decision Trees (DTs) are a widely used non-parametric supervised learning method for classification and regression. They operate by learning a hierarchy of simple decision rules from input features to predict a target variable. Their tree-like structure mirrors human decision-making, making them intuitive and easy to interpret—especially for newcomers to machine learning. Each node represents a decision, branches represent outcomes, and leaves provide final predictions. In regression, trees approximate the target function as piecewise constants. While decision trees are valued for their clarity and interpretability, they can suffer from overfitting or underfitting when used individually (Assegie and Elaraby, 2023). **Ensemble methods** such as Random Forests and Gradient Boost address these limitations by combining multiple decision trees to enhance accuracy and generalization. These techniques aggregate the strengths of individual trees, improving model robustness and performance. Let's now explore one such model called Random Forest in more detail (Scikit Learn, 2012).

Random Forest is an ensemble learning technique that constructs multiple decision trees and aggregates their outputs to produce more accurate and robust predictions. By combining the results of many individual trees—each trained on a random subset of data and features—it reduces the risk of overfitting that can occur with a single decision tree. This collective approach enhances generalization on unseen data. Moreover, Random Forest provides useful feature importance metrics, allowing practitioners to identify which variables contribute most significantly to the model's prediction (Scikit Learn, 2012).



Model efficiency

The efficiency of a decision tree is crucial in machine learning, especially in ensemble methods. Proper **tree depth** ensures a balance between overfitting—where deep trees capture noise—and underfitting, where shallow trees fail to learn meaningful patterns. Additionally, tree depth affects computational cost and interacts with other hyperparameters, such as the **number of trees** in an ensemble, influencing both accuracy and interpretability. Efficient decision trees contribute to the overall performance of ensemble models, enhancing predictive power while maintaining scalability (Scikit Learn, 2012).

Noise

In a perfect world, machine learning models would analyze data and give us super accurate results every time. But in reality, things aren't always that smooth—one big reason is **noise** in the dataset. Noise refers to random or irrelevant variations in the data that can hide real patterns and mess with model performance. It can come from things like measurement errors, typos during data entry, or just natural randomness in the real world. For example, a sensor might record slightly different values under the same conditions, or the target outcome might be influenced by factors we didn't capture. Too much noise can throw off decision trees, making them either latch onto meaningless patterns (overfitting) or miss important ones (underfitting).

To combat this, The algorithm can be instructed or tuned to provide optimal accuracy by adjusting the hyperparameters. Hyperparameters are configuration settings external to the learning algorithm that define how the model is trained, such as tree depth, learning rate, or the number of estimators, and they must be set before the training process begins. Effective hyperparameter tuning helps mitigate these effects. By optimizing these parameters, machine learning models can filter out noise, improve

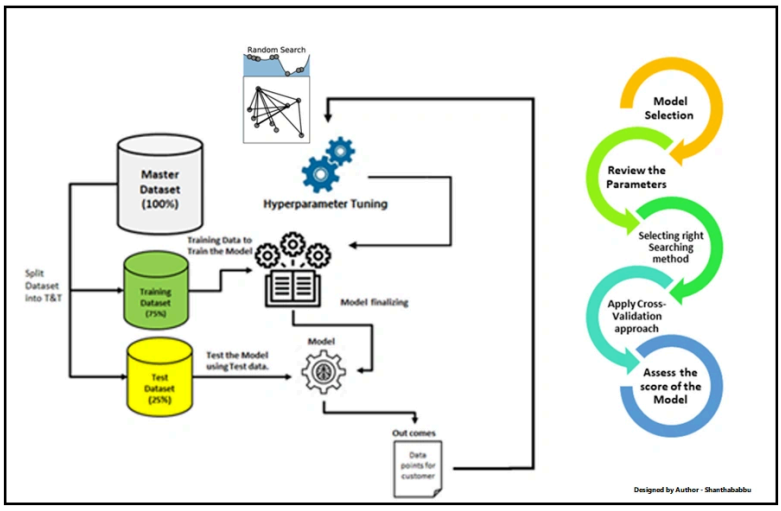
generalization, and enhance predictive accuracy. The next section explores how hyperparameter tuning plays a key role in addressing noise-related challenges (Scikit Learn, 2012).

Hyperparameter tuning

Importance and how it works

Hyperparameter tuning is the process of optimizing model parameters that are not learned directly from the data but significantly impact performance. In ensemble tree-based models, key hyperparameters such as tree depth (max_depth) and the number of trees (n_estimators) influence the model's ability to generalize. A shallow tree may lead to underfitting, failing to capture complex patterns, while an excessively deep tree risks overfitting by memorizing noise in the training data. Similarly, using too few trees results in high variance and unstable predictions, whereas too many trees increase computational cost without substantial performance gains. Effective hyperparameter tuning finds a balance, ensuring that the model generalizes well to unseen data while maintaining efficiency (Mantovani, 2018).

When it comes to fine-tuning machine learning models, there are two main approaches to adjusting hyperparameters: manual and automated tuning (Weerts, Mueller and Vanschoren, 2020)



Manual Tuning:

A hands-on approach where you iteratively test hyperparameter combinations, tracking results (metrics, logs, system data) to guide adjustments.

Automated Tuning:

Uses algorithms to search the hyperparameter space efficiently. You define:

1. Parameters and their value ranges (e.g., in dictionaries)
2. The algorithm then tests combinations and returns the optimal set.

Hyperparameter strategies

Hyperparameter tuning is a critical aspect of optimizing machine learning models, and several strategies exist for this purpose, each with its strengths and limitations:

Grid Search

Grid Search is the simplest method, where you test every possible combination of hyperparameter values in a predefined grid. It's like a brute-force approach to finding the best settings. While it's easy to implement and guarantees thoroughness, it's not the most efficient, especially for complex models or large datasets (Scikit Learn, 2012).

Pros:

- **Easy to Use**: Simple to set up, no fancy algorithms needed.
- **Thorough**: Tests all combinations, ensuring nothing is missed.

Cons:

- **Computational Overload**: Can get very slow and resource-heavy for large spaces.
- **Exponential Growth**: As you add more hyperparameters, the number of combinations grows out of control.

Random Search

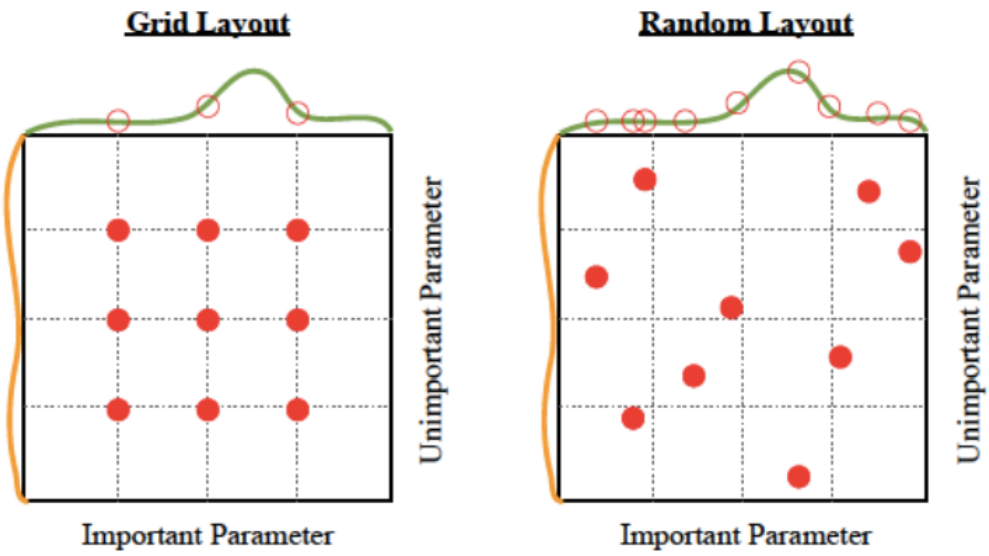
Random Search picks hyperparameters randomly from a range. It's more efficient than Grid Search because it explores the space more freely, but it still doesn't guarantee you'll find the best combination (Scikit Learn, 2012).

Pros:

- **More Efficient**: It often finds good solutions faster than Grid Search by covering a broader range.
- **Simple and Flexible**: Easy to implement with minimal setup.

Cons:

- **Still Resource-Heavy**: While better than Grid Search, it can still be quite computationally expensive.
- **Hits or Misses**: Random selections mean you might miss the optimal hyperparameters.
- **No Prior Knowledge**: Doesn't use what it's learned from previous tests.



Bayesian Optimization

Bayesian Optimization uses past results to intelligently explore the hyperparameter space. It's like having a smarter search that gets better with each trial. This makes it ideal for expensive or time-consuming models, as it can find the best configuration with fewer evaluations (scikit-optimize.github.io).

Pros:

- **Efficient**: Requires fewer trials to find the optimal solution.
- **Smart Search**: Adapts based on what it has already learned.
- **Great for Complex Problems**: Works well when dealing with high-dimensional spaces or expensive evaluations.

Cons:

- **Trickier to Set Up**: More complex to implement.
- **Extra Overhead**: Building the probabilistic model takes time and resources.

Existing literature

In machine learning, particularly for complex applications like predicting medical outcomes, careful tuning of model parameters is crucial for optimal performance. Several studies have highlighted key factors that influence model accuracy. For example, Assegie and Elaraby (2023) emphasize the importance of decision tree (DT) depth in predicting heart failure (HF) mortality, showing that the depth of the tree directly affects model accuracy. Their work suggests that tuning this parameter is essential for achieving precise predictions in medical contexts. Similarly, Weerts et al. (2020) propose a methodology for assessing the significance of hyperparameter tuning, using non-inferiority tests and tuning risk. They argue that in some cases, base models may outperform tuned ones, highlighting the need for careful evaluation of whether hyperparameter optimization justifies the computational cost. This

points to the importance of a balanced approach to tuning, particularly when it may add unnecessary complexity.

Example

To understand how this process works, we will take an example.

Step 1: Importing Libraries

We import key libraries for data handling (pandas, numpy), visualization (matplotlib, seaborn), and machine learning (RandomForestRegressor, xgboost). We also include tools for splitting data (train_test_split) and hyperparameter tuning (GridSearchCV, RandomizedSearchCV, BayesSearchCV), along with evaluation metrics (MAE, MSE, R^2).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from skopt import BayesSearchCV
import xgboost as xgb
```

Step 2: Generating Data

The generate_data function creates a synthetic dataset with 5 features and a target variable. The target y is based on mathematical operations on the features plus some added noise. The dataset is returned as a Pandas DataFrame. For simplicity and to better illustrate the concept, we generate data rather than using a real-world dataset.

```
def generate_data(n_samples=1000, noise=0.4):
    np.random.seed(42)
    X = np.random.rand(n_samples, 5) * 10
    y = (
        3 * np.sin(X[:, 0]) +
        2 * np.log1p(X[:, 1]) +
        1.5 * np.sqrt(X[:, 2]) -
        2.5 * X[:, 3] +
        0.8 * X[:, 4] +
        noise * np.random.randn(n_samples)
    )
    df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(1, 6)])
    df['Target'] = y
    return df
```

Step 3: Data Preprocessing

In this step, we separate the target variable y from the features X. The dataset is then split into training and testing sets using train_test_split. 80% of the data is used for training, and 20% for testing, ensuring the model is evaluated on unseen data. The random_state ensures reproducibility of the split.

```
y = df['Target']
X = df.drop(columns=['Target'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 4: Training and Evaluating the Baseline Model

In this step, we train a baseline Random Forest Regressor model with 10 estimators and a maximum depth of 2. The model is fit to the training data (X_train, y_train). After training, we predict the target values on the test set (X_test). The model's performance is then evaluated using three metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and R^2 score.

```
baseline_rf = RandomForestRegressor(n_estimators=10, max_depth=2, random_state=42)
baseline_rf.fit(X_train, y_train)
y_pred_rf = baseline_rf.predict(X_test)

print("Random Forest Baseline Model Performance:")
print("MAE:", mean_absolute_error(y_test, y_pred_rf))
print("MSE:", mean_squared_error(y_test, y_pred_rf))
print("R2 Score:", r2_score(y_test, y_pred_rf))
```

When we run the baseline model, the result yields an MAE of 3.20, an MSE of 14.85, and an R^2 score of 0.75, indicating a reasonable model fit with room for improvement. Let's see if Hyperparameter tuning strategies can improve the accuracy of the model.

Step 5: Hyperparameter Tuning with GridSearchCV

In this step, we use **GridSearchCV** to optimize the Random Forest model's hyperparameters. A parameter grid is defined for n_estimators, max_depth, and min_samples_split, and GridSearchCV evaluates all combinations using 5-fold cross-validation (cross-validation) and the R^2 scoring metric. The best model is then used to predict on the test set, and its performance is assessed using MAE, MSE, and R^2 score.

```
# Gridsearch
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10]
}
grid_search_rf = GridSearchCV(RandomForestRegressor(random_state=42),
param_grid_rf, cv=5, scoring='r2', n_jobs=-1)
grid_search_rf.fit(X_train, y_train)
y_grid_pred_rf = grid_search_rf.best_estimator_.predict(X_test)

print("Grid Search Tuned Model Performance:")
print("MAE:", mean_absolute_error(y_test, y_grid_pred_rf))
print("MSE:", mean_squared_error(y_test, y_grid_pred_rf))
print("R2 Score:", r2_score(y_test, y_grid_pred_rf))
```

Following GridSearchCV, We apply **RandomizedSearchCV**, which samples a fixed number of hyperparameter combinations from the grid (param_dist_rf) over 10 iterations. It uses 5-fold cross-validation with R^2 scoring to find the best model, which is then evaluated using MAE, MSE, and R^2 score.

```
# Random Search
param_dist_rf = {
    'n_estimators': np.arange(50, 300, 50),
    'max_depth': np.arange(5, 50, 5),
    'min_samples_split': np.arange(2, 15, 2)
}
random_search_rf = RandomizedSearchCV(RandomForestRegressor(random_state=42),
param_distributions=param_dist_rf, n_iter=10, cv=5, scoring='r2', n_jobs=-1, random_state=42)
random_search_rf.fit(X_train, y_train)
y_random_pred_rf = random_search_rf.best_estimator_.predict(X_test)

print("Random Tuned Model Performance:")
print("MAE:", mean_absolute_error(y_test, y_random_pred_rf))
print("MSE:", mean_squared_error(y_test, y_random_pred_rf))
print("R2 Score:", r2_score(y_test, y_random_pred_rf))
```

We apply **Bayesian Optimization** with **BayesSearchCV**, which models the objective function to select the most promising hyperparameter combinations (n_estimators, max_depth, min_samples_split). After 10 iterations with 5-fold cross-validation and R^2 scoring, the best model is evaluated using MAE, MSE, and R^2 score.

```
# Bayesian Optimization
bayes_search_rf = BayesSearchCV(RandomForestRegressor(random_state=42), {
    'n_estimators': (50, 300),
    'max_depth': (5, 50),
    'min_samples_split': (2, 15)
}, n_iter=10, cv=5, scoring='r2', n_jobs=-1, random_state=42)
bayes_search_rf.fit(X_train, y_train)
y_bayes_pred_rf = bayes_search_rf.best_estimator_.predict(X_test)

print("Bayesian Tuned Model Performance:")
print("MAE:", mean_absolute_error(y_test, y_bayes_pred_rf))
print("MSE:", mean_squared_error(y_test, y_bayes_pred_rf))
print("R2 Score:", r2_score(y_test, y_bayes_pred_rf))
```

Step 6: Comparing Model Performance

Comparing the performance of the three tuning methods, we observe that all models show similar performance, with Grid Search slightly outperforming the others. When compared to the **baseline model** (MAE = 3.20, MSE = 14.85, R^2 = 0.75), the tuned models significantly improve accuracy and predictive power.

```
Random Tuned Model Performance:
MAE: 1.5386655790258927
MSE: 3.589007006491372
R2 Score: 0.9404440628566181
```

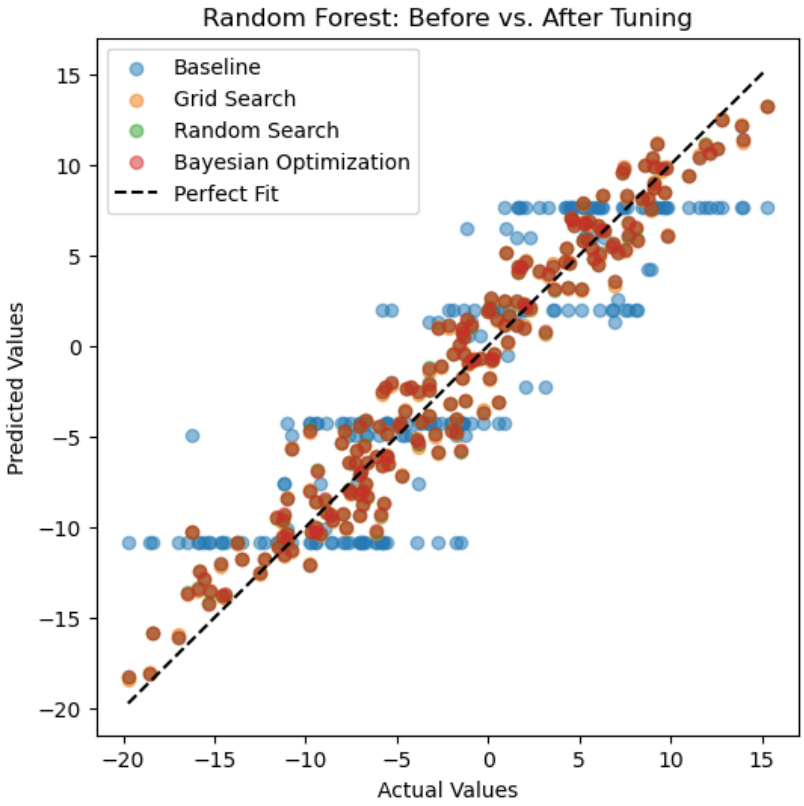
```
Grid Search Tuned Model Performance:
MAE: 1.530175655795297
MSE: 3.546202022532574
R2 Score: 0.9411543682222702
```

```
Bayesian Tuned Model Performance:
MAE: 1.5392397007044532
MSE: 3.5838682872144916
R2 Score: 0.9405293346996927
```

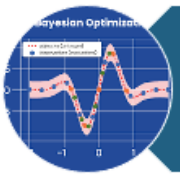
Step 7: Interpreting Results Through Scatter Plot

The scatter plot compares actual vs. predicted values for the baseline and tuned models. Points closer to the black dashed line (perfect fit) indicate better model performance.

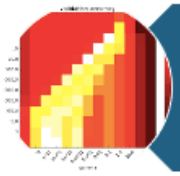
```
plt.figure(figsize=(6, 6))
plt.scatter(y_test, y_pred_rf, alpha=0.5, label='Baseline')
plt.scatter(y_test, y_grid_pred_rf, alpha=0.5, label='Grid Search')
plt.scatter(y_test, y_random_pred_rf, alpha=0.5, label='Random Search')
plt.scatter(y_test, y_bayes_pred_rf, alpha=0.5, label='Bayesian Optimization')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', label='Perfect Fit')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Random Forest: Before vs. After Tuning")
plt.legend()
plt.show()
```



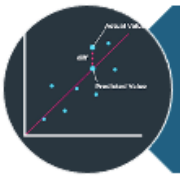
Key Takeaways



Hyperparameter tuning is crucial for enhancing model performance over a baseline model



Grid search, Random search and Bayesian optimization were employed



MAE decreased from 3.20 to 1.53, MSE from 14.85 to 3.55 and R2 increased from 0.75 to 0.94

References

Assegie, T.A. and Elaraby, A. (2023). Optimal Tree Depth in Decision Tree Classifiers for Predicting Heart Failure Mortality. *Healthcraft Frontiers*, 1(1), pp.58–66

scikit-optimize.github.io. (n.d.). *Bayesian optimization with skopt — scikit-optimize 0.8.1 documentation*. [online] Available at: https://scikit-optimize.github.io/stable/auto_examples/bayesian-optimization.html

Mantovani, R. G. (2018) "An empirical study on hyperparameter tuning of decision trees," ArXiv. ArXiv

Salman, H.A., Kalakech, A. and Steiti, A. (2024). Random Forest Algorithm Overview. *Deleted Journal*, 2024, pp.69–79. Salman, Kalakech and Steiti, 2024

Scikit Learn (2012). *3.2. Tuning the hyper-parameters of an estimator — scikit-learn 0.22 documentation*. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/grid_search.html, (Scikit Learn, 2012)

Weerts, H.J.P., Mueller, A.C. and Vanschoren, J. (2020). *Importance of Tuning Hyperparameters of Machine Learning Algorithms*. [online]