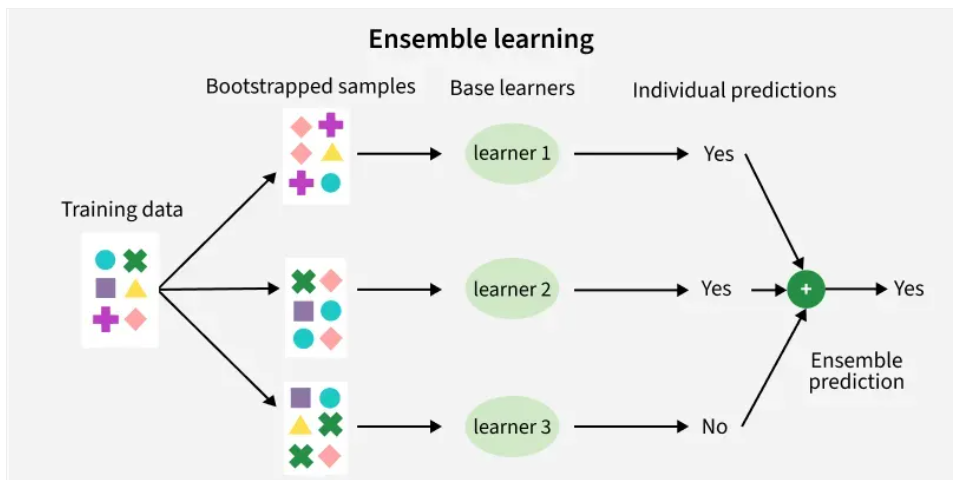
**Machine Learning Algorithms Part3**

**What is Ensemble Learning:**

<https://www.geeksforgeeks.org/a-comprehensive-guide-to-ensemble-learning/>

Ensemble learning is a method where we use many small models instead of just one. Each of these models may not be very strong on its own, but when we put their results together, we get a better and more accurate answer. It's like asking a group of people for advice instead of just one person—each one might be a little wrong, but together, they usually give a better answer.



**Types of Ensembles Learning in Machine Learning**

There are three main types of ensemble methods:

1. **Bagging (Bootstrap Aggregating):**  
   Models are trained independently on different random subsets of the training data. Their results are then combined—usually by averaging (for regression) or voting (for classification). This helps reduce variance and prevents overfitting.
2. **Boosting:**  
   Models are trained one after another. Each new model focuses on fixing the errors made by the previous ones. The final prediction is a weighted combination of all models, which helps reduce bias and improve accuracy.
3. **Stacking (Stacked Generalization):**  
   Multiple different models (often of different types) are trained, and their predictions are used as inputs to a final model, called a meta-model. The meta-model learns how to best combine the predictions of the base models, aiming for better performance than any individual model.

**1. Bagging Algorithm**

[Bagging classifier](https://www.geeksforgeeks.org/ml-bagging-classifier/) can be used for both regression and classification tasks. Here is an overview of Bagging classifier algorithm**:**

* **Bootstrap Sampling:** Divides the original training data into ‘N’ subsets and randomly selects a subset with replacement in some rows from other subsets. This step ensures that the base models are trained on diverse subsets of the data and there is no class imbalance.
* Base Model Training: For each bootstrapped sample we train a base model independently on that subset of data. These weak models are trained in parallel to increase computational efficiency and reduce time consumption. We can use different base learners i.e. different ML models as base learners to bring variety and robustness.
* **Prediction Aggregation:** To make a prediction on testing data combine the predictions of all base models. For classification tasks it can include majority voting or weighted majority while for regression it involves averaging the predictions.
* **Out-of-Bag (OOB) Evaluation**: Some samples are excluded from the training subset of particular base models during the bootstrapping method. These “out-of-bag” samples can be used to estimate the model’s performance without the need for cross-validation.
* **Final Prediction:** After aggregating the predictions from all the base models, Bagging produces a final prediction for each instance.

**Python pseudo code for Bagging Estimator implementing libraries:**

**1. Importing Libraries and Loading Data**

* **BaggingClassifier:** for creating an ensemble of classifiers trained on different subsets of data.
* **DecisionTreeClassifier:** the base classifier used in the bagging ensemble.
* **load\_iris:** to load the Iris dataset for classification.
* **train\_test\_split:** to split the dataset into training and testing subsets.
* **accuracy\_score**: to evaluate the model’s prediction accuracy.

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

**2. Loading and Splitting the Iris Dataset**

* **data = load\_iris():**loads the Iris dataset, which includes features and target labels.
* **X = data.data:** extracts the feature matrix (input variables).
* **y = data.target:** extracts the target vector (class labels).
* **train\_test\_split(...):**splits the data into training (80%) and testing (20%) sets, with random\_state=42 to ensure reproducibility.

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**3. Creating a Base Classifier**

**Decision tree is chosen as the base model. They are prone to overfitting when trained on small datasets making them good candidates for bagging.**

* **base\_classifier = DecisionTreeClassifier()**: initializes a Decision Tree classifier, which will serve as the base estimator in the Bagging ensemble.

base\_classifier = DecisionTreeClassifier()

**4. Creating and Training the Bagging Classifier**

* A **BaggingClassifier**is created using the decision tree as the base classifier.
* **n\_estimators = 10** specifies that 10 decision trees will be trained on different bootstrapped subsets of the training data.

bagging\_classifier = BaggingClassifier(base\_classifier, n\_estimators=10, random\_state=42)

bagging\_classifier.fit(X\_train, y\_train)

**5. Making Predictions and Evaluating Accuracy**

* The trained bagging model predicts labels for test data.
* The accuracy of the predictions is calculated by comparing the predicted labels (**y\_pred**) to the actual labels (**y\_test**).

y\_pred = bagging\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

**Output:**

*Accuracy: 1.0*

**2. Boosting Algorithm**

[Boosting](https://www.geeksforgeeks.org/boosting-in-machine-learning-boosting-and-adaboost/) is an ensemble technique that combines multiple weak learners to create a strong learner. Weak models are trained in series such that each next model tries to correct errors of the previous model until the entire training dataset is predicted correctly. One of the most well-known boosting algorithms is [AdaBoost (Adaptive Boosting).](https://www.geeksforgeeks.org/implementing-the-adaboost-algorithm-from-scratch/) Here is an overview of Boosting algorithm:

* **Initialize Model Weights**: Begin with a single weak learner and assign equal weights to all training examples.
* **Train Weak Learner**: Train weak learners on these dataset.
* **Sequential Learning**: Boosting works by training models sequentially where each model focuses on correcting the errors of its predecessor. Boosting typically uses a single type of weak learner like decision trees.
* **Weight Adjustment**: Boosting assigns weights to training datapoints. Misclassified examples receive higher weights in the next iteration so that next models pay more attention to them.

**Python pseudo code for boosting Estimator implementing libraries:**

**1. Importing Libraries and Modules**

* **AdaBoostClassifier from sklearn.ensemble:** for building the AdaBoost ensemble model.
* **DecisionTreeClassifier from sklearn.tree:** as the base weak learner for AdaBoost.
* **load\_iris from sklearn.datasets:**to load the Iris dataset.
* **train\_test\_split from sklearn.model\_selection:**to split the dataset into training and testing sets.
* **accuracy\_score from sklearn.metrics:**to evaluate the model’s accuracy.

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

**2. Loading and Splitting the Dataset**

* **data = load\_iris(): loads the Iris dataset, which includes features and target labels.**
* **X = data.data: extracts the feature matrix (input variables).**
* **y = data.target: extracts the target vector (class labels).**
* **train\_test\_split(...): splits the data into training (80%) and testing (20%) sets, with random\_state=42 to ensure reproducibility.**

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**3. Defining the Weak Learner**

We are creating the base classifier as a decision tree with maximum depth 1 (a decision stump). This simple tree will act as a weak learner for the**AdaBoost algorithm**, which iteratively improves by combining many such weak learners.

base\_classifier = DecisionTreeClassifier(max\_depth=1)

**4. Creating and Training the AdaBoost Classifier**

* **base\_classifier: The weak learner used in boosting.**
* **n\_estimators = 50: Number of weak learners to train sequentially.**
* **learning\_rate = 1.0: Controls the contribution of each weak learner to the final model.**
* **random\_state = 42: Ensures reproducibility.**

adaboost\_classifier = AdaBoostClassifier(

base\_classifier, n\_estimators=50, learning\_rate=1.0, random\_state=42

)

adaboost\_classifier.fit(X\_train, y\_train)

**5. Making Predictions and Calculating Accuracy**

We are calculating the accuracy of the model by comparing the true labels **y\_test**with the predicted labels **y\_pred**. The accuracy\_score function returns the proportion of correctly predicted samples. Then, we print the accuracy value.

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

**Output:**

*Accuracy: 1.0*

**Benefits of Ensemble Learning in Machine Learning**

Ensemble learning is a versatile approach that can be applied to machine learning model for: -

* **Reduction in Overfitting**: By aggregating predictions of multiple model's ensembles can reduce overfitting that individual complex models might exhibit.
* **Improved Generalization**: It generalizes better to unseen data by minimizing variance and bias.
* **Increased Accuracy**: Combining multiple models gives higher predictive accuracy.
* **Robustness to Noise**: It mitigates the effect of noisy or incorrect data points by averaging out predictions from diverse models.
* **Flexibility**: It can work with diverse models including decision trees, neural networks and support vector machines making them highly adaptable.
* **Bias-Variance Tradeoff**: Techniques like bagging reduce variance, while boosting reduces bias leading to better overall performance.

There are various ensemble learning techniques we can use as each one of them has their own pros and cons.

**Ensemble Learning Techniques**

| **Technique** | **Category** | **Description** |
| --- | --- | --- |
| **Random Forest** | Bagging | [Random forest](https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/) constructs multiple decision trees on bootstrapped subsets of the data and aggregates their predictions for final output, reducing overfitting and variance. |
| **Random Subspace Method** | Bagging | Trains models on random subsets of input features to enhance diversity and improve generalization while reducing overfitting. |
| **Gradient Boosting Machines (GBM)** | Boosting | [Gradient Boosting Machines](https://www.geeksforgeeks.org/ml-gradient-boosting/) sequentially builds decision trees, with each tree correcting errors of the previous ones, enhancing predictive accuracy iteratively. |
| **Extreme Gradient Boosting (XGBoost)** | Boosting | [XGBoost](https://www.geeksforgeeks.org/xgboost/) do optimizations like tree pruning, regularization, and parallel processing for robust and efficient predictive models. |
| **AdaBoost (Adaptive Boosting)** | Boosting | [AdaBoost](https://www.geeksforgeeks.org/implementing-the-adaboost-algorithm-from-scratch/) focuses on challenging examples by assigning weights to data points. Combines weak classifiers with weighted voting for final predictions. |
| **CatBoost** | Boosting | [CatBoost](https://www.geeksforgeeks.org/catboost-ml/) specialize in handling categorical features natively without extensive preprocessing with high predictive accuracy and automatic overfitting handling. |

* + **1.1.1.4 Random Forest Regressor**

Code - <https://github.com/campusx-official/100-days-of-machine-learning/tree/main/day65-random-forest>

* **What is random forest regressor Algorithm**
* **Intuition behind the random forest regressor Algorithm**
* **How Random forest perform so well Bias and variance trade off:**
* **Bagging vs random forest**
* **Random forest Hyper parameters**
* **Random forest’s Hyperparameters tuning using grid search cv and RandomizedSearchCV**

**The Intuition behind Random Forest! Explained with example.**

If you have heard about the **Decision tree**, then you are not very far from understanding what random forests are. There are two keywords here - **random** and **forests**. Let us first understand what forest means. A **random forest** is a collection of many decision trees. Instead of relying on a single decision tree, you build many decision trees say 100 of them. And you know what a collection of trees is called - a forest. So you now understand why is it called a forest.

**Why is it called random then?**

Say our dataset has 1,000 rows and 30 columns. There are two levels of randomness in this algorithm:

* **At row level:** Each of these decision trees gets a random sample of the training data (say 10%) i.e. each of these trees will be trained independently on 100 randomly chosen rows out of 1,000 rows of data. Keep in mind that each of these decision trees is getting trained on 100 randomly chosen rows from the dataset i.e they are different from each other in terms of predictions.
* **At column level:** The second level of randomness is introduced at the column level. Not all the columns are passed into training each of the decision trees. Say we want only 10% of columns to be sent to each tree. This means a randomly selected 3 column will be sent to each tree. So for the first decision tree, maybe column C1, C2, and C4 were chosen. The next DT will have C4, C5, C10 as chosen columns, and so on.

Let us now understand how an interview selection process resembles a random forest algorithm. Each panel in the interview process is actually a decision tree. Each panel gives a result of whether the candidate is a pass or fail and then a majority of these results are declared as final. Say there were 5 panels, 3 said yes and 2 said no. The final verdict will be yes.

Something similar happens in the random forest as well. The results from each of the trees are taken and the final result is declared accordingly. **Voting** and **averaging** are used to predict in case of **classification** and **regression** respectively.

With the advent of huge computational power at our disposal, we hardly think for even a second before we apply random forests. And very conveniently our predictions are made. Let us try to understand other aspects of this algorithm.

***When is a random forest a poor choice relative to other algorithms?***

1. *Random forests don't train well on smaller datasets as it fails to pick on the pattern*. To simplify, say we know that 1 pen costs INR 1, 2 pens cost INR 2, 3 pens cost INR 6. In this case, linear regression will easily estimate the cost of 4 pens but random forests will fail to come up with a good estimate.
2. *There is a problem of interpretability with random forest*. You can't see or understand the relationship between the response and the independent variables. Understand that a random forest is a predictive tool and not a descriptive tool. You get variable importance but this may not suffice in many analyses of interests where the objective might be to see the relationship between response and the independent features.
3. The time taken to train random forests may sometimes be too huge as you train multiple decision trees. Also, in the case of a categorical variable, the time complexity increases exponentially. *For a categorical column with n levels, RF tries split at 2^n -1 points to find the maximal splitting point.*
4. In the case of a regression problem, *the range of values the response variable can take is determined by the values already available in the training dataset*. Unlike linear regression, decision trees and hence random forest can't take values outside the training data.

***What are the advantages of using random forest?***

* Since we are using multiple decision trees, the bias remains the same as that of a single decision tree. However, the variance decreases and thus we decrease the chances of overfitting.
* When all you care about is the predictions and want a quick and dirty way out, the random forest comes to the rescue. You don't have to worry much about the assumptions of the model or linearity in the dataset.
* **How Random forest perform so well Bias and variance trade off:**

The Bias–Variance Tradeoff Challenge  
High-variance, low-bias models (e.g., decision trees, KNN, SVM with complex kernels) can capture complex patterns but overfit easily. High-bias, low-variance models (e.g., linear or logistic regression) generalize stably but underfit when relationships are non‑linear.  
  
Traditionally, we had to choose between these extremes there was no single algorithm that balanced both low bias and low variance.  
  
How Random Forest Strikes the Balance:  
Random Forest combines many decision trees (each low-bias, high-variance) in two key ways:  
  
Bootstrap Aggregation (Bagging):  
Each tree is trained on a different random bootstrap sample of the data. This decorrelates the trees: noisy or outlier data points are unlikely to appear in the same way in every sample. When we average (or vote) across all trees, the individual overfitting “noise” tends to cancel out, reducing variance substantially.  
  
Random Feature Subsampling:  
At each split, a random subset of features is considered.  
This further decorrelates trees, since different trees see different features at different splits.  
It prevents any single strong predictor from dominating every tree, lowering variance without increasing bias much.  
  
Key Takeaways:  
Because each tree remains a deep, flexible model, bias stays low. But by averaging many weakly correlated trees, variance also drops giving Random Forest its reputation for robust, high‑accuracy performance without much tuning.

* **Bagging vs random forest**

The fundamental difference is that in Random forests, only a subset of features are selected at random out of the total and the best split feature from the subset is used to split each node in a tree, unlike in bagging where all features are considered for splitting a node.

**What exactly happens in Bagging?**

Bagging in general is an acronym-like work that is a portmanteau of Bootstrap and aggregation. In general, if you take a bunch of bootstrapped samples of your original dataset, fit models, and then average all the model predictions this is bootstrap aggregation i.e. Bagging.

This is done as a step within the Random forest model algorithm. The random forest creates bootstrap samples and across observations and for each fitted decision tree a random subsample of the covariates/features/columns are used in the fitting process.

The selection of each covariate is done with uniform probability in the original bootstrap paper. So if you had 100 covariates you would select a subset of these features each has a selection probability of 0.01. If you only had 1 covariate/feature you would select that feature with probability 1.

Overall RF provides more randomness over the Bagging and hence there are fewer chances of overfitting.

Please add if there are any other thoughts on this.

**Deeper Understanding**

**✅ What is Bagging?**

* “Bagging” = **Bootstrap Aggregating**.
* You build multiple models (usually decision trees) on **bootstrapped** (random with replacement) datasets.
* Final prediction = **average (regression)** or **majority vote (classification)**.

🔁 **Each tree sees a slightly different dataset**, but the same feature set is available for each split.

**✅ What is Random Forest?**

* Random Forest = **Bagging + Random Feature Subset Selection**
* At **each split in the tree**, only a **random subset of features** is considered — not all features.
* This reduces **correlation** among trees → makes the ensemble more diverse → reduces variance more effectively.
* **Random forest Hyper parameters**

<https://www.kaggle.com/code/nargisbegum82/hyperparameter-tuning-in-random-forests>

Hyperparameters are configurations that cannot be learnt from the regular data that we provide to the algorithm, these are inbuilt to the algorithm and each algorithm has its own predefined set of hyperparameters.  Hyperparameters are often tuned for increasing model accuracy, and we can use various methods such as GridSearchCV, RandomizedSearchCV as explained in the article <https://www.geeksforgeeks.org/hyperparameter-tuning/>.

A deep understanding of hyperparameters is required because they are responsible for deciding how quickly a model can fit onto the data to produce accurate results. On the other hand, not finding the optimal values of hyperparameters can also result in less accuracy because of overfitting issue. Therefore, we will be having a closer look at the hyperparameters of random forest classifier to have a better understanding of the inbuilt hyperparameters:

* **n\_estimators:**We know that a random forest is nothing but a group of many decision trees, the n\_estimator parameter controls the number of trees inside the classifier. We may think that using many trees to fit a model will help us to get a more generalized result, but this is not always the case. However, it will not cause any overfitting but can certainly increase the time complexity of the model. **The default number of estimators is 100**in scikit-learn.
* **max\_depth:**It governs the maximum height upto which the trees inside the forest can grow. It is one of the most important hyperparameters when it comes to increasing the accuracy of the model, as we increase the depth of the tree the model accuracy increases upto a certain limit but then it will start to decrease gradually because of overfitting in the model. It is important to set its value appropriately to avoid overfitting. **The default value is set to** **None,**None specifies that the nodes inside the tree will continue to grow until all leaves become pure or all leaves contain less than **min\_samples\_split** (another hyperparameter).
* **min\_samples\_split:**It specifies the minimum amount of samples an internal node must hold in order to split into further nodes. If we have a very low value of min\_samples\_splits then, in this case, our tree will continue to grow and start overfitting. By increasing the value of min\_samples\_splits we can decrease the total number of splits thus limiting the number of parameters in the model and thus can aid in reducing the overfitting in the model. However, the value should not be kept very large that a number of parameters drop extremely causing the model to underfit. We generally keep min\_samples\_split value between 2 and 6. **However, the default value is set to 2.**
* **min\_samples\_leaf:** It specifies the minimum amount of samples that a node must hold after getting split. It also helps to reduce overfitting when we have ample amount of parameters. Less number of parameters can lead to overfitting also, we should keep in mind that increasing the value to a large number can lead to less number of parameters and in this case model can underfit also. **The default value is set to 1.**
* **max\_features:**Random forest takes random subsets of features and tries to find the best split.  max\_features helps to find the number of features to take into account in order to make the best split. It can take four values "**auto**", "**sqrt**", "**log2**" and **None**.
  + In case of auto: considers max\_features = sqrt(n\_features)
  + In case of sqrt: considers max\_features = sqrt(n\_features), it is same as auto
  + In case of log2: considers max\_features = log2(n\_features)
  + In case of None: considers max\_features = n\_features
* **max\_leaf\_nodes:**It sets a limit on the splitting of the node and thus helps to reduce the depth of the tree, and effectively helps in reducing overfitting. If the value is set to None, the tree continues to grow infinitely.
* **max\_samples:**This hyperparameter helps to choose maximum number of samples from the training dataset to train each individual tree.

These are the major hyperparameters that are present implicitly in the random forest classifier which is required to be tuned in order to increase the accuracy of our training model.

* **Random forest’s Hyperparameters tuning using grid search cv and RandomizedSearchCV**

<https://www.geeksforgeeks.org/random-forest-hyperparameter-tuning-in-python/>

[Random Forest](https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/) is one of the most popular machine learning algorithms used for both classification and regression tasks. It works by building multiple decision trees and combining their outputs to improve accuracy and control overfitting. While Random Forest is a robust model, fine-tuning its hyperparameters such as the number of trees, maximum depth and feature selection can improve its prediction and performance and in this article we will learn how we can do it.

Since we are talking about Random Forest Hyperparameters let us see what different Hyperparameters can be Tuned:

**1. n\_estimators:**It defines the number of trees in the forest. More trees typically improve model performance but increase computational cost. In the below example it takes 100 trees.

***By default:*** *n\_estimators=100*

**2. max\_features:** Limits the number of features to consider when splitting a node. This helps control overfitting.

***By default:*** *max\_features="sqrt" [available: ["sqrt", "log2", None}]*

* **sqrt**: Selects the square root of the total features. This is a common setting to reduce overfitting and speed up the model.
* **log2:** This option selects the base-2 logarithm of the total number of features. It provide more randomness and reduce overfitting more than the square root option.
* **None:** If None is chosen the model uses all available features for splitting each node. This increases the model's complexity and may cause overfitting, especially with many features.

**3. max\_depth**: Controls the maximum depth of each tree. A shallow tree may underfit while a deep tree may overfit. So choosing right value of it is important.

***By default:*** *max\_depth=None*

**4. max\_leaf\_nodes:** Limits the number of leaf nodes in the tree hence controlling its size and complexity. None means it takes an unlimited number of nodes.

***By default:*** *max\_leaf\_nodes = None*

**5. max\_sample**: Apart from the features, we have a large set of training datasets. max\_sample determines how much of the dataset is given to each individual tree. None means data.shape[0] is taken.

***By default:*** *max\_sample = None*

**6. min\_sample\_split:** Specifies the minimum number of samples required to split an internal node. In the below example every node has 2 subnodes.

***By default:*** *min\_sample\_split = 2*

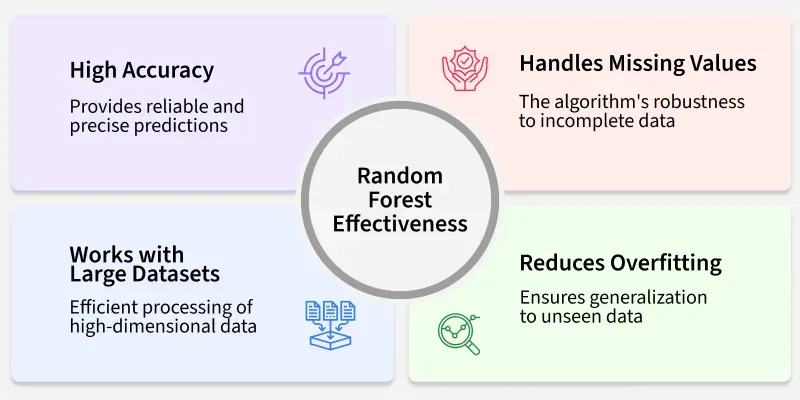
**4) Random Forest Regressor**

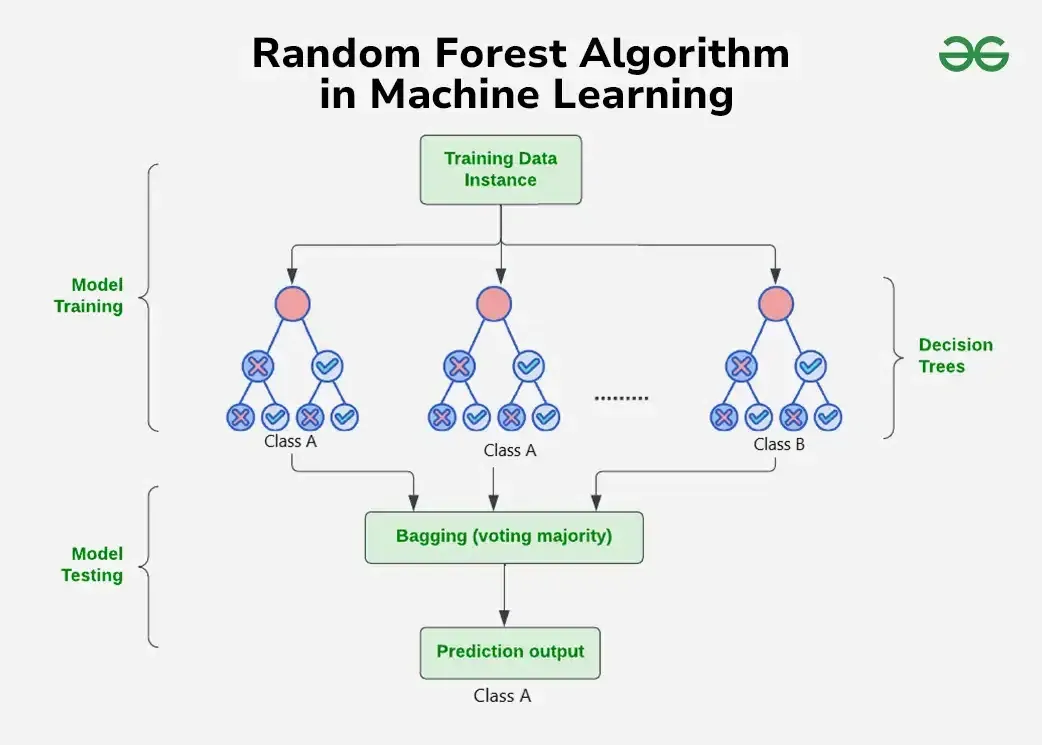
**📘 What is it?**

A **Random Forest Regressor** is an **ensemble learning method** that builds multiple decision trees and averages their outputs to improve predictive accuracy and reduce overfitting.

It's part of the **Bagging** family (Bootstrap Aggregation).

Random Forest is a machine learning algorithm that uses many decision trees to make better predictions. Each tree looks at different random parts of the data and their results are combined by voting for classification or averaging for regression. This helps in improving accuracy and reducing errors.





**Working of Random Forest Algorithm**

* **Create Many Decision Trees:** The algorithm makes many [decision trees](https://www.geeksforgeeks.org/decision-tree/) each using a random part of the data. So every tree is a bit different.
* **Pick Random Features:** When building each tree it doesn’t look at all the features (columns) at once. It picks a few at random to decide how to split the data. This helps the trees stay different from each other.
* **Each Tree Makes a Prediction:** Every tree gives its own answer or prediction based on what it learned from its part of the data.
* **Combine the Predictions:**
  + For **classification** we choose a category as the final answer is the one that most trees agree on i.e majority voting.
  + For **regression** we predict a number as the final answer is the average of all the trees predictions.
* **Why It Works Well:** Using random data and features for each tree helps avoid overfitting and makes the overall prediction more accurate and trustworthy.

*Random forest is also a ensemble learning technique which you can learn more about from:* [*Ensemble Learning*](https://www.geeksforgeeks.org/a-comprehensive-guide-to-ensemble-learning/)

**Key Features of Random Forest**

* **Handles Missing Data:** It can work even if some data is missing so you don’t always need to fill in the gaps yourself.
* **Shows Feature Importance:** It tells you which features (columns) are most useful for making predictions which helps you understand your data better.
* **Works Well with Big and Complex Data:** It can handle large datasets with many features without slowing down or losing accuracy.
* **Used for Different Tasks:** You can use it for both **classification** like predicting types or labels and **regression** like predicting numbers or amounts.

**Assumptions of Random Forest**

* **Each tree makes its own decisions**: Every tree in the forest makes its own predictions without relying on others.
* **Random parts of the data are used**: Each tree is built using random samples and features to reduce mistakes.
* **Enough data is needed**: Sufficient data ensures the trees are different and learn unique patterns and variety.
* **Different predictions improve accuracy**: Combining the predictions from different trees leads to a more accurate final result.

**Advantages of Random Forest Regression**

* **Handles Non-Linearity**: It can capture complex, non-linear relationships in the data that other models might miss.
* **Reduces Overfitting**: By combining multiple decision trees and averaging predictions it reduces the risk of overfitting compared to a single decision tree.
* **Robust to Outliers**: Random Forest is less sensitive to outliers as it aggregates the predictions from multiple trees.
* **Works Well with Large Datasets**: It can efficiently handle large datasets and high-dimensional data without a significant loss in performance.
* **Handles Missing Data**: Random Forest can handle missing values by using surrogate splits and maintaining high accuracy even with incomplete data.
* **No Need for Feature Scaling**: Unlike many other algorithms Random Forest does not require normalization or scaling of the data.

**Disadvantages of Random Forest Regression**

* **Complexity**: It can be computationally expensive and slow to train especially with a large number of trees and high-dimensional data. Due to this it may not be suitable for real-time predictions especially with a large number of trees.
* **Less Interpretability**: Since it uses many trees it can be harder to interpret compared to simpler models like linear regression or decision trees.
* **Memory Intensive**: Storing multiple decision trees for large datasets require significant memory resources.
* **Overfitting on Noisy Data**: While Random Forest reduces overfitting, it can still overfit if the data is highly noisy especially with a large number of trees.
* **Sensitive to Imbalanced Data**: It may perform poorly if the dataset is highly imbalanced like one class is significantly more frequent than another.

**🔍 Typical Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| 📈 Predicting house prices | Complex nonlinear patterns in features |
| 🌳 Environmental modelling | Predict temperature, rainfall, pollution |
| 💰 Stock price forecasting | Time series with multiple features |
| ⚙️ Industrial manufacturing | Predictive maintenance, defect prediction |
| 📦 E-commerce | Sales prediction, demand forecasting |

**🛠️ Full Real-World Code (California Housing Dataset)**

We’ll predict **median house prices** using a single feature (AveRooms) for visualization.

python

CopyEdit

# 📌 Step 1: Import Libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import fetch\_california\_housing

from sklearn.ensemble import RandomForestRegressor

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

# 📌 Step 2: Load Dataset

data = fetch\_california\_housing()

X = data.data

y = data.target

# Use only 1 feature (AveRooms)

X = X[:, [3]] # Feature at index 3: Average rooms per household

# 📌 Step 3: Train/Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 📌 Step 4: Train Random Forest Regressor

model = RandomForestRegressor(n\_estimators=100, max\_depth=10, random\_state=42)

model.fit(X\_train, y\_train)

# 📌 Step 5: Predictions

y\_pred = model.predict(X\_test)

# 📌 Step 6: Evaluate Model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("✅ Mean Squared Error (MSE):", mse)

print("✅ R^2 Score:", r2)

# 📌 Step 7: Visualization

plt.figure(figsize=(10,6))

plt.scatter(X\_test, y\_test, color='blue', alpha=0.5, label='Actual')

plt.scatter(X\_test, y\_pred, color='green', alpha=0.5, label='Predicted (RF)')

plt.xlabel('Average Rooms per Household')

plt.ylabel('Median House Value')

plt.title('Random Forest Regressor - California Housing')

plt.legend()

plt.grid(True)

plt.show()

**🧪 Sample Output (will vary)**

java

CopyEdit

✅ Mean Squared Error (MSE): 0.392

✅ R^2 Score: 0.71

Random Forest performs significantly better than a single decision tree and linear regression due to ensemble learning and ability to model complex relationships.