Table of contents

Introduction to Random Forest

In my last article I talked about `Decision Trees` and I think it's time to talk about a closely related algorithm and my favorite algorithm called `Random Forest`. As always I'm `Md. Rishat Talukder`. Let's get started.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Introduction

`Random Forest` is more like a method of training a `Decision Tree` rather than a `stand alone` model.

Random Forest is a `ensemble` of `Decision Trees` and it's a `supervised learning` algorithm.

What is a `ensemble`?

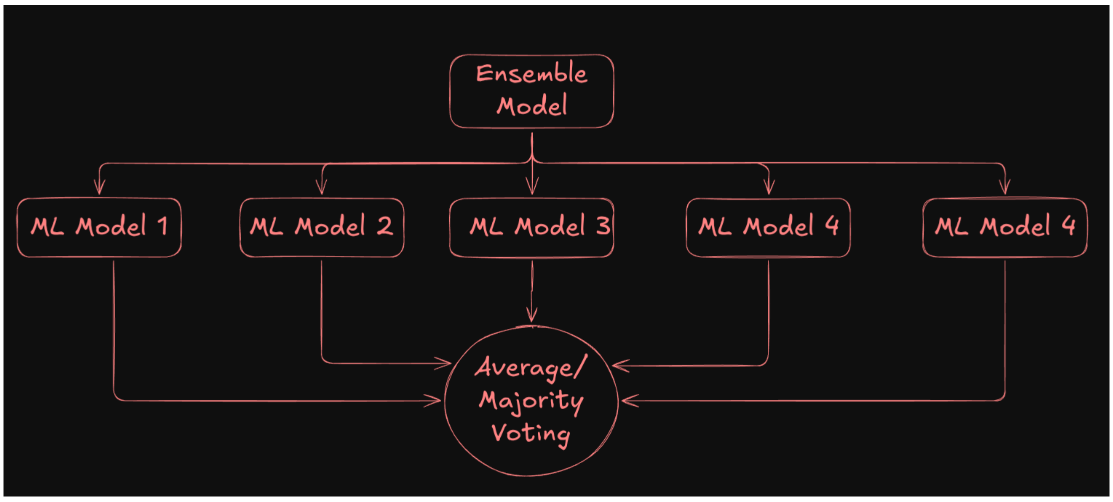
Ensemble

`Ensemble` refers to a `collection` of `models` that are trained together and then `combined` to make a `final prediction`.

Let's say you have a `K Nearest Neighbor` model and you train it on a certain dataset and then you take a bunch of `K Nearest Neighbor` models and train them on different `subsets` of the same dataset. You combine the predictions of all the `K Nearest Neighbor` models and then you make a final prediction.

> There is a slight chance that the predictions of the `K Nearest Neighbor` models trained on different subsets of the dataset will be better than a single `K Nearest Neighbor` model trained on the whole dataset.

This is the main idea of `ensemble` algorithms.



> For classification problems `majority voting` is used to combine the predictions and for regression problems `averaging` is used to combine the predictions.

There are 3 types of ensemble algorithms:

- `Bagging`
- `Boosting`
- `Stacking`

Below are **clear, descriptive, article-ready notes** on **Bagging, Boosting, and Stacking**. They're written to flow naturally after decision trees / random forests and to emphasize **intuition first, math second**.

Ensemble Learning Overview

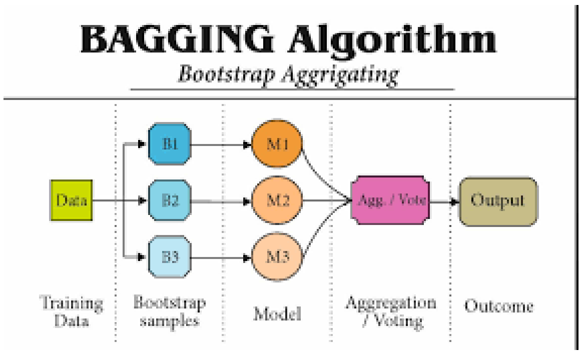
Ensemble methods combine multiple models to produce a single, stronger predictor. The core idea is simple:

> **A group of imperfect models can outperform a single perfect-looking one.**

The three most important ensemble strategies are **Bagging**, **Boosting**, and **Stacking**.

Bagging (Bootstrap Aggregating)

`Bagging` is an ensemble technique where multiple models are trained **independently** on different **bootstrap samples** of the same dataset, and their predictions are **aggregated**.



- How it works:

1. Create multiple datasets by **sampling with replacement**

2. Train a separate model on each dataset

3. Combine predictions:

    - Classification → majority vote
    - Regression → average

Each model sees **slightly different data**, which introduces diversity.

This method:

- Reduces **variance**
- Prevents overfitting
- Stabilizes unstable models

Works best with **high-variance models**, such as decision trees.

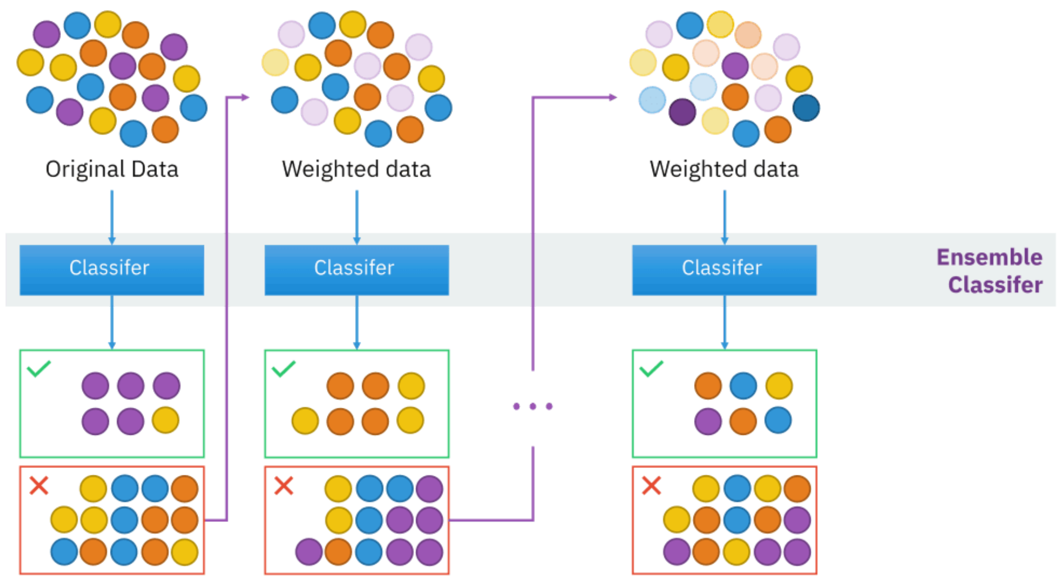
Models are trained **in parallel**, no dependency between models, same algorithm used for all models, bias remains roughly the same. `Bagged Decision Trees` and `Random Forest (bagging + feature randomness)` uses this method.

If dataset is small or noisy, If the model overfits easily `bagging` can work like a charm.

> Bagging reduces variance by training many independent models on different views of the same data.

Boosting

`Boosting` is another ensemble method where models are trained **sequentially**, and each new model focuses on correcting the mistakes made by previous ones.



How it works (conceptually):

1. Train a weak model
2. Increase importance of misclassified samples
3. Train the next model on harder cases
4. Repeat
5. Combine all models into a weighted prediction

Each model learns from the **errors of the previous models**.

It's a little more complex then bagging.

Why Boosting works

This model helps reducing **bias**, converts weak learners into a strong learner, forces the model to focus on difficult patterns. It is especially effective when the model is **underfitting**.

Even though the models are trained **sequentially** each model depends on previous ones and sensitive to noise and outliers and often requires regularization

Some popular `boosting` algorithms are:

- AdaBoost
- Gradient Boosting
- XGBoost
- LightGBM
- CatBoost

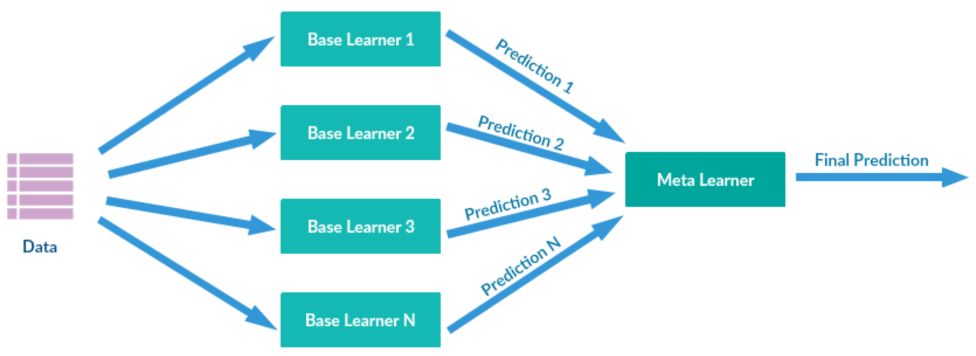
When to use Boosting?

- Bias is high
- Dataset is reasonably clean
- Complex patterns exist

> Boosting improves performance by forcing models to learn from their own mistakes.

Stacking (Stacked Generalization)

Stacking is an ensemble technique where **different types of models** are combined, and a **meta-model** learns how to best combine their predictions.



How it works:

1. Train multiple **base models** (e.g., tree, SVM, logistic regression)
2. Generate predictions from these models
3. Use those predictions as features
4. Train a **meta-learner** on top of them

> *Main Idea is which model to trust, and when.*

This method leverages strengths of different model families, captures complementary patterns, more flexible than bagging or boosting because of the meta-learner.

This method is not perfect. The models can be heterogeneous and requires careful cross-validation, more complex and harder to debug. This model has high risk of data leakage if done incorrectly

> Stacking learns how to combine models, instead of assuming all models are equally useful.

Comparison of Ensemble Methods

Method	Main Goal	Training Style	Reduces	Risk
Bagging	Stability	Parallel	Variance	Underfitting

Method	Main Goal	Training Style	Reduces	Risk
Boosting	Accuracy	Sequential	Bias	Overfitting to noise
Stacking	Flexibility	Multi-layer	Both	Data leakage

> Bagging, boosting, and stacking represent three different philosophies of ensemble learning: reducing variance through independence, reducing bias through correction, and reducing error through intelligent combination. Understanding when and why each approach works is far more important than memorizing their algorithms.

Now let's talk about `Random Forest`.

Random Forest

`Random Forest` Algorithm is a `ensemble` of `Decision Trees`. Main concept is to train multiple `Decision Trees` on different subsets of the dataset and then combine their predictions to make a final prediction.

Hmmmm?! This sound familiar?

It's the bagging technique applied to `Decision Trees` buuuuut there some catches.

How it works

- `Step 1: Start with the original dataset`

Assume we have:

- A feature matrix `X`
- A target variable `y`

This dataset may contain:

- Noise
- Correlated features
- Limited samples

> A single decision tree trained on this data is likely to **overfit**.

- `Step 2: Create bootstrap samples (row sampling)`

For each tree in the forest:

- Randomly sample data points **with replacement**.
- The sample size is usually equal to the original dataset size.
- Some points may appear multiple times.

- Some points may not appear at all.

These unused samples are called **Out-of-Bag (OOB)** samples.

> Almost 37% of the data is left out for each tree.

This introduces variation between trees and at the same time prevents all trees from learning the same noise

- `Step 3: Train a decision tree on each bootstrap sample`

Each bootstrap sample is used to train **one decision tree**.

Key property:

- Trees are usually grown **deep and unpruned**
- Each tree has **low bias but high variance**

At this stage, each tree is intentionally allowed to overfit.

- `Step 4: Random feature selection at each split`

When a tree tries to split a node:

- It does **not** consider all features.
- It randomly selects a subset of features.
  - Classification: `√(number of features)`
  - Regression: `number of features / 3`

The best split is chosen **only from this subset**.

Purpose:

- Reduces correlation between trees
- Prevents dominant features from controlling every tree
- Encourages diverse decision boundaries

- `Step 5: Repeat to build many trees`

Steps 2–4 are repeated:

Let's say you are making 100 or a 1000 decision trees.

Each tree:

- Sees different data
- Uses different features
- Makes different mistakes

- `Step 6: Aggregate predictions from all trees`

Once all trees are trained:

- For classification:

- Each tree votes for a class
- The final prediction is the **majority vote**

- For regression:

- Each tree outputs a numeric value
- The final prediction is the **average**

> Even though the tree are intentionally gets overfitted this aggregation smooths out individual tree errors.

- `Step 7: Evaluate using Out-of-Bag samples (optional)`

Since each tree leaves out almost 37% of the data:

- These OOB samples can act as a **validation set**.
- Predictions are made only by trees that did not see the sample.

This provides:

- An unbiased estimate of generalization performance
- No need for a separate validation split

Why Random Forest Works

Random Forest works because it combines:

- **Low-bias models** (deep trees)
- **Randomness** (data + features)
- **Averaging** (variance reduction)

Mathematically:

- Variance decreases as tree correlation decreases.
- Bias remains relatively low.

> A Random Forest does not try to be clever with a single model. Instead, it lets many simple models disagree—and then trusts the consensus.

As always we are going to use `scikit-learn` to build a `Random Forest` model. So, I think I should introduce you guys with some of the `key hyperparameters`.

- `n_estimators` → number of trees
- `max_depth` → controls overfitting
- `min_samples_split` → prevents noisy splits
- `max_features` → controls feature randomness

> Having understood how Random Forest reduces variance through controlled randomness, we can now appreciate why ensemble

> methods often outperform single models on real-world data.

Now let's see how we implement it.

A simple model using `Random Forest`

Well let's follow the steps of `Machine Learning Pipeline`.

Data collection

For example I'll be using the `iris` dataset from `sklearn` Just so we I can show you if there is a difference between `Decision Tree` and `Random Forest`.

In [8]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris(as_frame=True)
```

This will return a `dictionary` like object that will have both `feature` and `target` variables.

So, I will just extract the `feature` and `target` variables from the `iris` object.

In [9]:
```
features = iris.data
target = iris.target
```

In [12]:
```
features.head()
```

Out[12]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Now, we can use the data to build a `Random Forest` model.

Splitting and training

Before splitting I want to see the target value distribution.

In [11]:
```
target.value_counts()
```

Out[11]:

```
target
0    50
1    50
2    50
Name: count, dtype: int64
```

As you can see, there is `3` classes in the target variable each with `50` samples. So, the data is balanced and we can expect the model to learn each class well.

So, let's split.

In [14]:

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    features, target, test_size=0.2, random_state=42, stratify=target
)
```

> I turned the `stratify` parameter on to ensure that the split is stratified(all the classes have the same distribution in the `training` and `testing` sets).

Now, to inplement the `Random Forest` we can import the `Random Forest` model from the `ensemble` module of `scikit-learn` and train the model.

In [ ]:

```python
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=50)
```

> `n_estimators` is the number of trees in the forest. By default, it is set to 100. However, you can adjust this value to change the number of trees in the forest. You can also set the random state for the random sampling process to ensure reproducibility.

Now, we fit the model to the training data.

In [16]:

```python
model.fit(X_train, y_train)
```

Out[16]:

```
▾ RandomForestClassifier ⓘ ?

▸ Parameters
```

And it's done let's do some predictions with the `predict` method.

In [17]:

```python
predictions = model.predict(X_test)
predictions
```

Out[17]:

```
array([0, 2, 1, 1, 0, 1, 0, 0, 2, 1, 2, 2, 2, 1, 0, 0, 0, 1, 1, 2, 0, 2,
       1, 1, 2, 2, 1, 0, 2, 0])
```

Well, prediction is done let's see the `classification report`

In [18]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       0.90      0.90      0.90        10
           2       0.90      0.90      0.90        10

    accuracy                           0.93        30
   macro avg       0.93      0.93      0.93        30
weighted avg       0.93      0.93      0.93        30
```

Ow wow! It's a wonderful result. `93%` accuracy with `93%` `macro F1-score`. Amazing!

With such a small dataset and also a small number of trees, we can see how `Random Forest` works.

Let's see if the same can be said for a `Decision Tree` model.

In [19]:
```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

dt_predictions = dt.predict(X_test)

print(classification_report(y_test, dt_predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       0.90      0.90      0.90        10
           2       0.90      0.90      0.90        10

    accuracy                           0.93        30
   macro avg       0.93      0.93      0.93        30
weighted avg       0.93      0.93      0.93        30
```

WHAAAT! Decision tree model also does a wonderful job. `93%` accuracy with `93%` `macro F1-score`. Exactly same as the `Random Forest` model.

Hmmm... Somethings wrong. Wasn't it supposed to be the `Random Forest` model doing a better job?

This confusion can be debunked if we see the performance of the models on the whole dataset.

Let's cross validate and see the which one actually does a better job.

> I'll do a 5-fold cross validation. I've already made a article talking about cross validation before so you can check it out if you don't understand the fundamentals of cross validation.

In [22]:
```
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

accuracy_scores_RF = cross_val_score(model, features, target, cv=cv, scori
print(f'Accuracy scores for each fold on Random Forest: {accuracy_scores_R
print(f'Average accuracy for Random Forest: {accuracy_scores_RF.mean()}')
```

```
Accuracy scores for each fold on Random Forest: [1.         0.96666667 0.93
333333 1.         0.9        ]
Average accuracy for Random Forest: 0.9600000000000002
```

Again, wow! Average accuracy for Random Forest is `96%` which is amazing and if you look closely for some of the folds the accuracy is `100%`.

Let's see if we can say the same for the `Decision Tree` model.

In [23]:
```
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

accuracy_scores_RF = cross_val_score(dt, features, target, cv=cv, scoring=
print(f'Accuracy scores for each fold on Decision Tree: {accuracy_scores_R
print(f'Average accuracy for Decision Tree: {accuracy_scores_RF.mean()}')
```

```
Accuracy scores for each fold on Decision Tree: [1.         0.96666667 0.93
333333 0.96666667 0.9        ]
Average accuracy for Decision Tree: 0.9533333333333335
```

Hmmmm... It's close but the average accuracy for Decision Tree is `95.3%` which is not bad and very very close to the `Random Forest` model. But you can clearly see that the `Random Forest` model does a better job on the whole dataset.

But The difference is not that big and sometimes `decision tree` can out perform `Random Forest` on the whole dataset.

`So, why is it so close even though the` Random Forest `model is built to out perform` Decision Tree `on the whole dataset?`

Although Random Forest is designed to outperform a single Decision Tree by reducing variance, the performance gap on the Iris dataset is naturally very small. This is because Iris is a **clean, low-noise, low-dimensional dataset** where a single decision tree is already capable of learning the underlying patterns with minimal overfitting. In such scenarios, the variance reduction offered by Random Forest provides only a marginal benefit. Additionally, both models rely on randomness—Decision Trees through data ordering and Random Forest through bootstrapping and feature sampling—so small fluctuations in cross-validation splits can cause either model to appear slightly better in a given run. This does not contradict the theory behind Random Forest; rather, it highlights that ensemble methods show their true advantage on **noisier, higher-dimensional, or more complex datasets**, where single trees tend to overfit and variance becomes a dominant issue.

> When the problem is simple, the ensemble has little variance left to reduce.

> Before ending here's a small task for you, cross validate the `macro f1` scores and see which one does better.

Final Words

We are same same but **DIFFERENT**