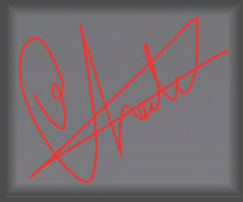
Ensemble Learning
+
Random Forest



Ensemble Learning

Ensemble Learning is a machine learning technique that combines multiple individual models (called base learners or weak learners) to create a stronger more robust predictive model. The core idea is that a group of models working together can often achieve better performance than any single model alone.

Types of Ensemble Learning:

1. Bagging (Bootstrap Aggregation)
   - Trains multiple models on different subsets of training data
   - Each subset is created through bootstrap sampling
   - final prediction combines results from all models
   - Ex. Random Forest, Extra trees

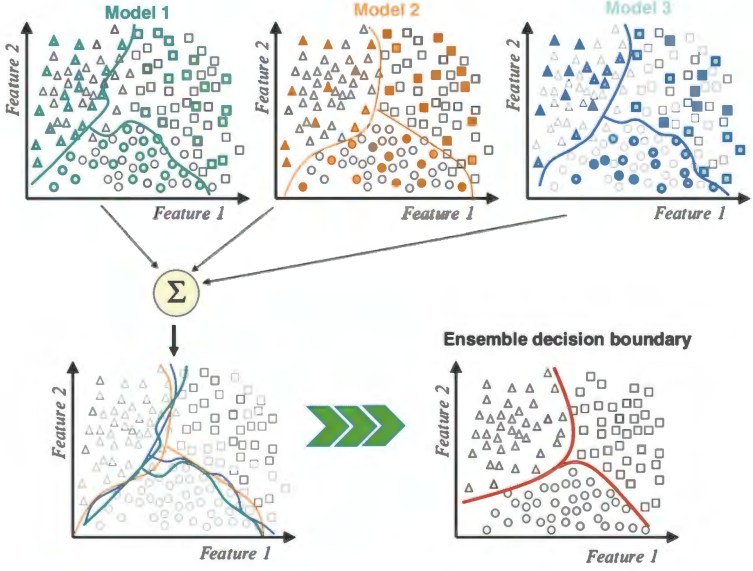
2. Boosting
   - Trains model sequentially, where each new model focuses on correcting the errors of previous models.
   - Models weighted based on their performance
   - Ex. Ada Boost, Gradient Boosting, XGBoost
   - Reduces bias and can convert weak learners into strong ones

Stacking (Stacked Generalization)

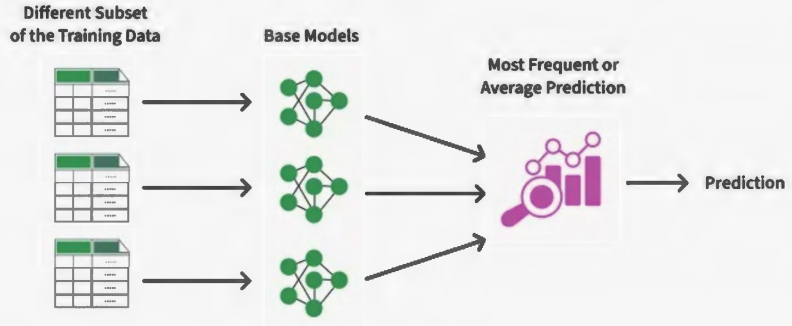
- Uses a meta-learner (blender) to learn how to best combine predictions from multiple base models.
- Base models make predictions, then meta-learner uses these predictions as features
- More complex but potentially more powerful than other methods.

Benefits of Ensemble Learning

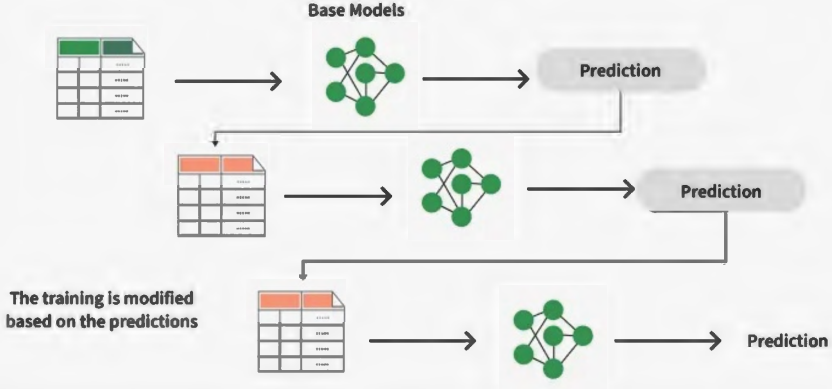
Improved Accuracy: Multiple models can capture different patterns in the data, leading to better overall performance than individual models.
Reduced Overfitting: By combining multiple models, ensemble methods can generalize better to unseen data, especially with bagging approaches.
Increased Robustness: Less sensitive to outliers and noise since errors from individual models can be averaged out.
Better Handling of Complex Relationships: Different models may capture different aspects of the data, providing a more comprehensive understanding.
Reduced Variance: Particularly with bagging, the variance of predictions is reduced compared to single models.



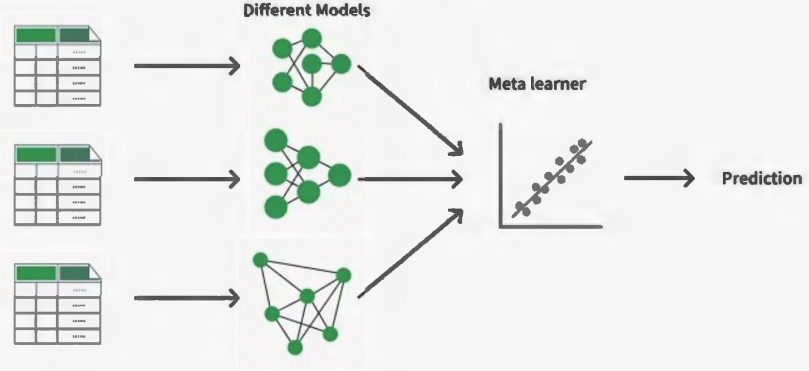
Bagging



Boosting



Stacking



Bagging (Bootstrap Aggregated)

Bagging is an ensemble method that combines multiple models trained on different subsets of the training data to reduce variance and improve generalization. The name comes from Bootstrap sampling + Aggregating Prediction.

Simply, Different kind of data subsets from same data is given to multiple models of the same type and than the majority occuring output is selected. Random Forest is a popular Bagging algorithm.

Data Sampling Methods.

Basic Types

With Replacement: Put the ball back in the bag after picking it (can pick same item multiple times) **Without Replacement**: Don't put the ball back (each item picked only once)

Common Methods

Random: Pick randomly from all data **Stratified**: Keep the same class ratios (if 70% cats, 30% dogs, sample keeps this ratio) **Bootstrap**: Random sampling with replacement - used in bagging

Imbalanced Data

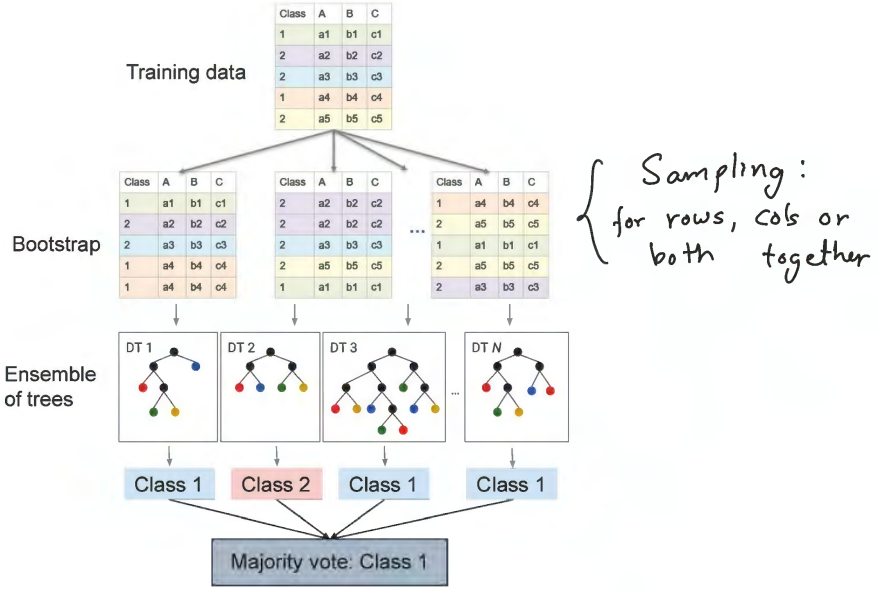
Oversampling: Make more copies of rare class (minority) **Undersampling**: Remove some common class (majority) **SMOTE**: Create fake examples of rare class by mixing real ones

Key Point

- **With replacement** = Bootstrap = Bagging
- **Without replacement** = Regular train/test splits
- **Stratified** = Keep proportions same
- **Balance techniques** = Fix unequal class sizes

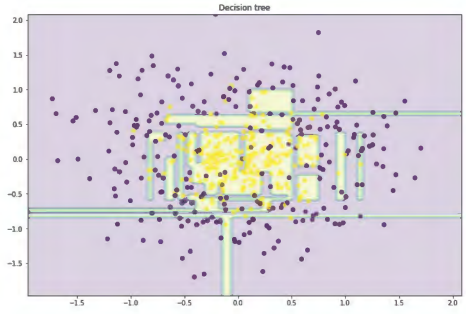
Random Forest

Random Forest is an ensemble learning method used for classification and regression

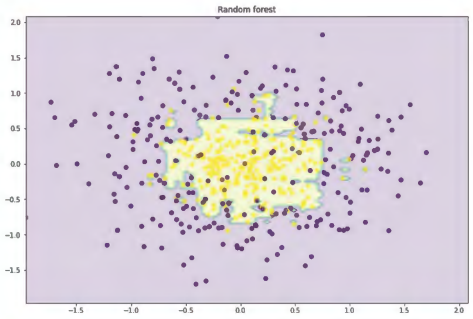


Random Forest works because of a universal law known as "Wisdom of Crowd". Because we are training each Decision tree of different subsets of same data, it gives a better accuracy.

Notice the two figures below, what actually happens is that: let's say we have 100 decision tree model. Each model will have mistakes at some places. But a single mistake by few decision trees will not be mistake for majority of the trees, which is why when we average all decision trees, despite each having their own flaws, will predict correctly.



1 DT



After 100 DT

Random Forest has unique ability to convert a low bias high variance model to low bias low variance models, this is because of "wisdom of crowd".
(Decision Trees are by default low bias)

Difference between Random Forest and Bagging (with using Decision trees)

Two differences.

1. Bagging can take any model as base estimator. (meaning we can take only SVM, or only KNN etc,) but in Random Forest we only have decision tree as the base estimator.
2. The biggest difference due to which random forest outperforms Bagging: even though decision trees are used as base estimators in bagging:

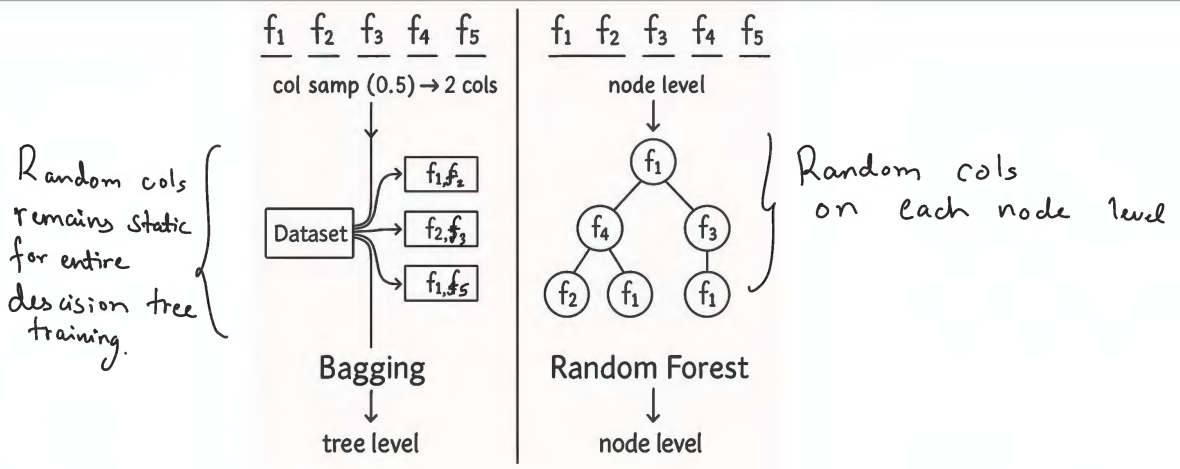
Bagging: The randomness happens once at the beginning. If you randomly select certain columns, each decision tree gets its own different random subset of columns, but then uses those same columns throughout its entire training process. So Tree 1 might get columns A, C, D and Tree 2 might get columns B, D, E, but each tree sticks with its assigned columns.

Random Forest: The randomness happens continuously. At every single node when the tree is deciding how to split, it randomly picks which columns to consider. So even within one tree, different nodes might use different random subsets of columns. For example, Tree 1's root node might randomly consider columns A, C, D, but its next node might randomly consider columns B, D, E, and so on at each decision point.

Simple analogy:

- Bagging = Give each tree a different fixed set of tools at the start, and they keep those same tools throughout
- Random Forest = At every decision point, each tree randomly picks which tools to use from the full toolbox

This extra randomness in Random Forest makes the trees more diverse and usually leads to better performance.



Due to the randomness in node level as well, there's a lot of randomness, and that's why due to wisdom of crowd Random forest outperforms.

Feature Importance

If we recall how feature importance is done in DT, Gini Importance formula is used to calculate it.

In Random forest, each feature importance given from each Decision trees is averaged.

Gini Importance Calculation in Random Forest

Step-by-Step Process:

For Each Tree:

1. **At each node split:** Calculate Gini impurity decrease
   - Gini decrease = (weighted_left × gini_left) + (weighted_right × gini_right) - gini_parent
   - Weight = number of samples in node / total samples
2. **For each feature:** Sum up all Gini decreases where that feature was used for splitting
   - Feature_importance = Σ(Gini decrease for all nodes using this feature)
3. **Normalize within tree:** Divide each feature's importance by total importance in that tree

Across All Trees:

4. **Average across forest:** Take the mean of each feature's importance across all trees
   - Final_importance(feature) = Mean(importance_tree1, importance_tree2, ..., importance_treeN)

Example:

- Tree 1: Feature A used at 3 nodes → Gini decreases: 0.1, 0.05, 0.02 → Total: 0.17
- Tree 2: Feature A used at 2 nodes → Gini decreases: 0.08, 0.03 → Total: 0.11
- Tree 3: Feature A used at 1 node → Gini decrease: 0.06 → Total: 0.06
- **Final Feature A importance = (0.17 + 0.11 + 0.06) / 3 = 0.113**

The final scores are normalized so all feature importances sum to 1.0.

Random Forest Hyperparameters Reference Guide

Forest Level Parameters

n_estimators Number of trees in the forest Default: 100 | Range: 10-1000+ | More trees = better performance but slower

bootstrap Use bootstrap sampling for training Default: True | Options: True/False | Adds randomness to prevent overfitting

max_samples Number of samples per tree (if bootstrap=True) Default: None | Type: int/float/None | Lower = more randomness

oob_score Calculate out-of-bag score for validation Default: False | Options: True/False | Free validation estimate

n_jobs Number of parallel jobs (-1 = all cores) Default: None | Type: int/None | Higher = faster training

random_state Seed for reproducible results Default: None | Type: int/None | Use int for reproducibility

verbose Control training output verbosity Default: 0 | Type: int | Higher = more progress info

warm_start Add more trees to existing model Default: False | Options: True/False | Enables incremental learning

class_weight Weights for imbalanced classes Default: None | Options: dict/'balanced'/None | Handles class imbalance

Tree Level Parameters

criterion Function to measure split quality Default: "gini"/"squared_error" | Options: gini/entropy/log_loss/squared_error/etc.

max_depth Maximum depth of trees Default: None | Range: 3-20 | Lower = less overfitting, higher = more complex

min_samples_split Minimum samples required to split node Default: 2 | Type: int/float | Higher = prevents overfitting

min_samples_leaf Minimum samples required at leaf node Default: 1 | Type: int/float | Higher = smoother model

min_weight_fraction_leaf Minimum weighted fraction at leaf Default: 0.0 | Range: 0.0-0.5 | Useful for weighted samples

max_leaf_nodes Maximum number of leaf nodes Default: None | Type: int/None | Alternative to max_depth

min_impurity_decrease Minimum impurity decrease for split Default: 0.0 | Type: float | Higher = fewer splits

ccp_alpha Complexity parameter for pruning Default: 0.0 | Type: float | Higher = more pruning

Feature Selection Parameters

max_features Number of features considered for best split Default: "sqrt"/"1.0" | Options: int/float/"sqrt"/"log2"/None | Lower = more randomness

Quick Reference Summary:

- **Forest Size**: n_estimators (100), bootstrap (True), max_samples (None)
- **Per**
- Handle Imbalance: class_weight='balanced'**formance**: oob_score (False), n_jobs (None), verbose (0)
  Reproducibility: random_state (None), warm_start (False)
  Class Balance: class_weight (None)
  Tree Structure: max_depth (None), criterion ("gini"/"squared_error")
  Split Control: min_samples_split (2), min_samples_leaf (1)
  Pruning: min_impurity_decrease (0.0), ccp_alpha (0.0)
  Feature Selection: max_features ("sqrt"/"1.0")
  Common Tuning Combinations:
  Prevent Overfitting: ↑max_depth, ↑min_samples_split, ↑min_samples_leaf
  Improve Performance: ↑n_estimators, tune max_features
- Speed Up Training: ↑n_jobs, ↓n_estimators

"OOB" in Machine Learning

"OOB" stands for **"out-of-bag"**. In the context of machine learning, an out-of-bag score is a method of measuring the prediction error of random forests, bagging classifiers, and other ensemble methods that use bootstrap aggregation (bagging) when sub-samples of the training dataset are used to train individual models.

How it works:

1. **Bootstrap Sampling and Out-of-Bag Samples**
   Each tree in the ensemble is trained on a distinct bootstrap sample of the data. By the nature of bootstrap sampling, some samples from the dataset will be left out during the training of each tree. These samples are called "out-of-bag" samples.
2. **Validation Using OOB Samples**
   The out-of-bag samples can then be used as a validation set. We can pass them through the tree that didn't see them during training and obtain predictions.
3. **Computing the OOB Score**
   These predictions are then compared to the actual values to compute an "out-of-bag score", which can be thought of as an estimate of the prediction error on unseen data.

Advantages

One of the advantages of the out-of-bag score is that it allows us to estimate the prediction error without needing a separate validation set. This can be particularly useful when the dataset is small and partitioning it into training and validation sets might leave too few samples for effective learning.

Extremely Randomized Trees

Extra Trees is short for "Extremely Randomized Trees". It's a modification of the Random Forest algorithm that changes the way the splitting points for decision tree branches are chosen.

In traditional decision tree algorithms (and therefore in Random Forests), the optimal split point for each feature is calculated, which involves a degree of computation. For a given node, the feature and the corresponding optimal split point that provide the best split are chosen. On the other hand, in the Extra Trees algorithm, for each feature under consideration, a split point is chosen completely at random. The best-performing feature and its associated random split are then used to split the node. This adds an extra layer of randomness to the model, hence the name "Extremely Randomized Trees".

Because of this difference, Extra Trees tend to have more branches (be deeper) than Random Forests, and the splits are made more arbitrarily. This can sometimes lead to models that perform better, especially on tasks where the data may not have clear optimal split points. However, like all models, whether Extra Trees will outperform Random Forests (or any other algorithm) depends on the specific dataset and task.

Advantages

- **Robustness to Overfitting:** Random Forests are less prone to overfitting compared to individual decision trees, because they average the results from many different trees, each of which might overfit the data in a different way.
- **Handling Large Datasets:** They can handle large datasets with high dimensionality effectively.
- **Less Pre-processing:** Random Forests can handle both categorical and numerical variables without the need for scaling or normalization. They can also handle missing values.
- **Variable Importance:** They provide insights into which features are most important in the prediction.
- **Parallelizable:** The training of individual trees can be parallelized, as they are independent of each other. This speeds up the training process.
- **Non-Parametric:** Random Forests are non-parametric, meaning they make no assumptions about the functional form of the transformation from inputs to output. This makes them very flexible and able to model complex, non-linear relationships.

Disadvantages

- **Model Interpretability:** One of the biggest drawbacks of Random Forests is that they lack the interpretability of simpler models like linear regression or decision trees. While you can rank features by their importance, the model as a whole is essentially a black box.
- **Performance with Unbalanced Data:** Random Forests can be biased towards the majority class when dealing with unbalanced datasets. This can sometimes be mitigated by balancing the dataset prior to training.
- **Predictive Performance:** Although Random Forests generally perform well, they may not always provide the best predictive performance. Gradient boosting machines, for instance, often outperform Random Forests. If the relationships within the data are linear, a linear model will likely perform better than a Random Forest.
- **Inefficiency with Sparse Data:** Random Forests might not be the best choice for sparse data or text data where linear models or other algorithms might be more suitable.
- **Parameters Tuning:** Although Random Forests require less tuning than some other models, there are still several parameters (like the number of trees, tree depth, etc.) that can affect model performance and need to be optimized.
- **Difficulty with High Cardinality Features:** Random Forests can struggle with high cardinality categorical features (features with a large number of distinct values). These types of features can lead to trees that are biased towards the variables with more levels, and may cause overfitting.
- **Can't Extrapolate:** This is because they do not predict beyond the range of the training data, and they may not predict as accurately as other regression models.