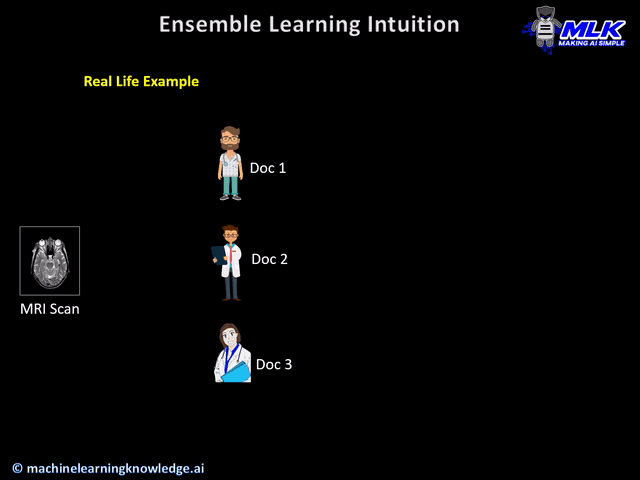
**Ensemble Learning**

**https://www.datacamp.com/tutorial/ensemble-learning-python**

**What is 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.

**Ensemble Learning:**



**Why Do We Need Ensemble Learning?**

There are several reasons why we need ensemble learning. In this article, we will explore some of the key reasons why ensemble learning is important.

**1. Reducing Overfitting**

One of the main benefits of ensemble learning is that it can help to reduce overfitting. Overfitting occurs when a model is too complex and fits the training data too closely, resulting in poor generalization to new data.

Ensemble learning can help to reduce overfitting by combining multiple models that have been trained on different subsets of the data or with different algorithms. This helps to reduce the variance of individual models and improve the generalization performance of the ensemble.

from sklearn.ensemble import BaggingClassifier  
from sklearn.tree import DecisionTreeClassifier  
  
# Create a decision tree classifier  
tree = DecisionTreeClassifier()  
  
# Create a bagging classifier  
bag = BaggingClassifier(base\_estimator=tree, n\_estimators=10, random\_state=42)  
  
# Train the bagging classifier  
bag.fit(X\_train, y\_train)  
  
# Make predictions on the test data  
y\_pred = bag.predict(X\_test)

In this example, we create a decision tree classifier and use it as the base estimator for the bagging classifier. We set the number of estimators to 10, which means we will train 10 models on different subsets of the data.

**2. Improving Accuracy**

Ensemble learning can also help to improve the accuracy of models. By combining multiple models, we can take advantage of their individual strengths and weaknesses.

For example, one model may be good at capturing linear relationships in the data, while another model may be good at capturing non-linear relationships. By combining these models, we can create a more accurate model that takes advantage of the strengths of each individual model.

from sklearn.ensemble import VotingClassifier  
from sklearn.linear\_model import LogisticRegression  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.tree import DecisionTreeClassifier  
  
# Create the individual classifiers  
lr = LogisticRegression(random\_state=42)  
knn = KNeighborsClassifier()  
tree = DecisionTreeClassifier(random\_state=42)  
  
# Create the ensemble classifier  
ensemble = VotingClassifier(estimators=[('lr', lr), ('knn', knn), ('tree', tree)], voting='hard')  
  
# Train the ensemble classifier  
ensemble.fit(X\_train, y\_train)  
  
# Make predictions on the test data  
y\_pred = ensemble.predict(X\_test)

In this example, we create three different classifiers (logistic regression, k-nearest neighbors, and decision tree) and use them as the base estimators for the voting classifier. We set the voting parameter to ‘hard’, which means that the predictions are based on a simple majority vote.

**3. Handling Noisy Data**

Ensemble learning can also be useful when working with noisy data. Noisy data can make it difficult to train accurate models, as the noise can introduce errors and biases into the model.

Ensemble learning can help to reduce the impact of noise by combining multiple models that have been trained on different subsets of the data or with different algorithms. This helps to reduce the impact of noise on individual models and improve the overall accuracy of the ensemble.

from sklearn.ensemble import RandomForestClassifier  
  
# Create a random forest classifier  
rf = RandomForestClassifier(n\_estimators=100, max\_features='sqrt', random\_state=42)  
  
# Train the random forest classifier  
rf.fit(X\_train, y\_train)  
  
# Make predictions on the test data  
y\_pred = rf.predict(X\_test)

In this example, we create a random forest classifier with 100 trees and use the square root of the number of features as the maximum number of features to consider at each split. We then train the classifier on the training data and make predictions on the test data.

**4. Handling Imbalanced Data**

Ensemble learning can also be useful when working with imbalanced data. Imbalanced data occurs when one class is much more prevalent than the others. This can make it difficult to train accurate models, as the model may be biased towards the majority class.

Ensemble learning can help to address this issue by combining multiple models that have been trained on different subsets of the data or with different algorithms. This helps to reduce the bias towards the majority class and improve the accuracy of the model for all classes.

from imblearn.ensemble import BalancedBaggingClassifier  
from sklearn.tree import DecisionTreeClassifier  
  
# Create a decision tree classifier  
tree = DecisionTreeClassifier()  
  
# Create a balanced bagging classifier  
bag = BalancedBaggingClassifier(base\_estimator=tree, n\_estimators=10, random\_state=42)  
  
# Train the balanced bagging classifier  
bag.fit(X\_train, y\_train)  
  
# Make predictions on the test data  
y\_pred = bag.predict(X\_test)

In this example, we create a decision tree classifier and use it as the base estimator for the balanced bagging classifier. We set the number of estimators to 10, which means we will train 10 models on different subsets of the data. The balanced bagging classifier uses a resampling strategy to balance the class distribution in each subset of the data.

**5. Handling Large Datasets**

Ensemble learning can also be useful when working with large datasets. Large datasets can be computationally expensive to train and may require specialized hardware or distributed computing systems.

Ensemble learning can help to reduce the computational cost of training models by allowing us to train multiple models in parallel on different subsets of the data or with different algorithms.

from dask\_ml.ensemble import RandomForestClassifier  
  
# Create a random forest classifier  
rf = RandomForestClassifier(n\_estimators=100, max\_features='sqrt', random\_state=42)  
  
# Train the random forest classifier using dask  
rf.fit(X\_train\_dask, y\_train\_dask)  
  
# Make predictions on the test data using dask  
y\_pred\_dask = rf.predict(X\_test\_dask)

In this example, we create a random forest classifier with 100 trees and use the square root of the number of features as the maximum number of features to consider at each split. We train the classifier using dask\_ml, which is a library for distributed machine learning using Dask. This allows us to train the model on large datasets using multiple processors or even multiple machines.

Machine learning models are not like traditional software solutions.  These models need constant updates as new data becomes available for accurate and reliable predictions. In complex and sensitive scenarios, relying on a single model may  not be sufficient to generate an optimal result,  and this is where ensemble modeling can help.

This conceptual tutorial covers what ensemble modeling in machine learning is and how it can improve your overall model performance. Then, we’ll provide an overview of various ensemble methods before diving into the illustration of a real-world scenario using a step-by-step implementation with Python.

Our [**Ensemble Learning in R with SuperLearner**](https://www.datacamp.com/tutorial/ensemble-r-machine-learning) tutorial explains how to boost your machine learning results and based on ensemble learning approach using the SuperLearner package in R.

**Ensemble Models in Machine Learning**

Let’s imagine a music manager participating in an international competition.  They have access to a wide variety of musicians with different expertise:

* Classical musicians with the ability to compose traditional pieces.
* Electronic musicians who are experts in using electronic instruments.
* Jazz musicians with a great sense of improvisation.
* Soloist musicians who can perform complex solos and highlight their technical abilities.

Given the broad spectrum of musical expertise and background, the manager can combine all of them to create a unique and memorable performance.

Think of ensemble models as an orchestra of musicians, where each person specializes in a specific instrument like piano, trumpet, drum, and more. The combination of those skills creates a harmonious melody.

Ensemble learning uses the same logic:

It combines multiple algorithms to obtain better predictive performance than the one from a single model. There is no predefined number of models to consider, and some business goals may require more models than others.

**Model Error and Reducing this Error with Ensembles**

The error emerging from any machine model can be broken down into three components mathematically:

**Bias + Variance + Irreducible error**

Why is this important in the current context? To understand what goes on behind an ensemble model, you need first to know what causes an error in the model.

Let’s look at these errors:

**Bias error**

This is useful to quantify how much, on an average, the predicted values are different from the actual value. A high bias error means we have an underperforming model that keeps missing essential trends.

**Variance**

On the other side, Variance quantifies how the predictions made on the same observation differ. A high variance model will over-fit on your training population and perform poorly on any observation beyond training. The following diagram will give you more clarity (assume that the red spot is the real value, and the blue dots are predictions):

Typically, as you increase the complexity of your model, you will see a reduction in error due to lower bias in the model. However, this only happens until a particular point. As you continue to make your model more complex, you end up over-fitting your model, and hence your model will start suffering from the high variance.

Now that you are familiar with the basics of ensemble learning let's look at different ensemble learning techniques:

**Types of Ensemble Methods**

There are different types of ensemble methods, and each one brings a set of advantages and disadvantages. This section covers those aspects to help you make the right choice for your use cases.

Before diving into each method, let’s understand what meta and base learners are for a better understanding of the next concepts.

* **Base learners** are the first level of an ensemble learning architecture, and each one of them is trained to make individual predictions.
* **Meta learners**, on the other hand, are in the second level, and they are trained on the output of the base learners.

**Bagging ensemble learning**

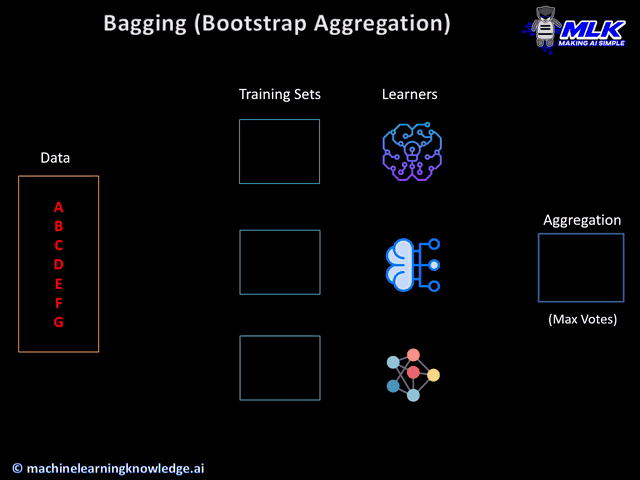
[Bagging classifier](https://www.geeksforgeeks.org/machine-learning/What-is-Bagging-classifier/) can be used for both regression and classification tasks. Here is an overview of Bagging classifier algorithm**:**

* **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.

Bagging is also known as bootstrap aggregation.  This technique is similar to [**random forest**](https://www.datacamp.com/tutorial/random-forests-classifier-python), but it uses all the predictors, whereas random forest uses only a subset of predictors in each tree.

In bagging, a random sample of data from the training set is selected with replacement, which enables the duplication of sample instances in a set. Below are the main steps involved in bagging:

* Generation of multiple bootstrap resamples.
* Running an algorithm on each resample to make predictions.
* Combining the predictions by taking the average of the predictions or taking the majority vote (for classification).



Bagging presents several key advantages and disadvantages when used for classification or regression tasks.

**Advantages**

* The modeling process is straightforward and does not require any deep mathematical concepts, and can handle missing values.
* The scikit-learn package makes it easy to implement the underlying logic. It contains all the modules to combine the predictions of each model, also known as base learners.
* It has a significant effect on reducing variance on high-variance classifiers. This is helpful when dealing with high-dimensional data, preventing the model from accurately generalizing on new data.
* Bagging provides an unbiased estimate of the out-of-bag error, which corresponds to the average error/loss that all these classifiers yield.

**Disadvantages**

* Bagging is computationally expensive due to the use of several models.
* The averaging involved across predictions makes it difficult to interpret the final result.
* Here are the **main techniques** used in bagging and its variations like **pasting**:

**✅ 1. Bagging (Bootstrap Aggregating)**

* **Sampling with replacement** (bootstrapping)
* Each base learner gets a random subset of the training data (with replacement)
* Learners are trained independently in parallel
* Final prediction is made via:
  + Majority voting (for classification)
  + Averaging (for regression)

Ex. code

BaggingClassifier(DecisionTreeClassifier(), bootstrap=True)

📌 Used in: **Random Forest**, BaggingClassifier (Scikit-learn)

**✅ 2. Pasting**

* Same as bagging, **but without replacement**
* Subsets are drawn randomly, **without** putting back samples
* Good when you have **enough data** and want more diverse models

📌 Used in: BaggingClassifier(..., bootstrap=False) in Scikit-learn

Ex. code

BaggingClassifier(DecisionTreeClassifier(), bootstrap=False)

**✅ 3. Random Subspaces**

* Each base learner is trained on a **random subset of features**, not data
* Increases diversity among models
* Especially useful in high-dimensional spaces

📌 Often used with decision trees or SVMs

Ex. code

BaggingClassifier(DecisionTreeClassifier(), bootstrap=False, bootstrap\_features=True, max\_features=0.5)

**✅ 4. Random Patches**

* Combines both:
  + **Random subset of instances**
  + **Random subset of features**
* Most general form of bagging
* Improves generalization by injecting maximum randomness

📌 This is what **Random Forests** essentially do

Ex. code

BaggingClassifier(DecisionTreeClassifier(), bootstrap=True, bootstrap\_features=True, max\_samples=0.5, max\_features=0.5)

**✅ 5. Bootstrapped Boostrap (Advanced) –** also called **Meta-Bagging**

* Meta-bootstrapping techniques where bootstraps themselves are aggregated
* Rare in practical ML pipelines, but used in advanced statistical ensembles

Applies bagging inside another bagging – a simple simulation of hierarchical bootstrapping.

Ex. code

inner\_bag = BaggingClassifier(DecisionTreeClassifier(), n\_estimators=5)

outer\_bag = BaggingClassifier(inner\_bag, n\_estimators=3)

**✅ 6. Bootstrap with Stratification**

* Ensures that each bootstrap sample maintains **class proportions**
* Useful in imbalanced classification problems

📌 Controlled by StratifiedKFold or StratifiedShuffleSplit when combined with ensembles

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import StratifiedShuffleSplit

from sklearn.metrics import accuracy\_score

from scipy.stats import mode

# Load Iris dataset

X, y = load\_iris(return\_X\_y=True)

# Create Stratified splits

sss = StratifiedShuffleSplit(n\_splits=5, test\_size=0.3, random\_state=42)

# Store predictions from each model

all\_preds = []

# Base model

def get\_base\_model():

return DecisionTreeClassifier()

# Training on each stratified split

for train\_index, test\_index in sss.split(X, y):

X\_train, X\_test = X[train\_index], X[test\_index]

y\_train = y[train\_index]

model = get\_base\_model()

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

preds = model.predict(X[test\_index])

all\_preds.append(preds)

# Convert to shape (n\_models, n\_samples)

all\_preds = np.array(all\_preds)

# Transpose to (n\_samples, n\_models) for voting

all\_preds\_T = all\_preds.T

# Majority voting across predictions

final\_preds, \_ = mode(all\_preds\_T, axis=1)

# Final predictions after voting

final\_preds = final\_preds.ravel()

# Ground truth from last test split (for simplicity)

\_, test\_index = list(sss.split(X, y))[-1]

y\_true = y[test\_index]

# Accuracy

acc = accuracy\_score(y\_true, final\_preds)

print(f"Stratified Bagging Accuracy: {acc:.4f}")

**📊 Summary Table**

| **Technique** | **Data Sampling** | **Feature Sampling** | **With Replacement?** | **Use Case** |
| --- | --- | --- | --- | --- |
| **Bagging** | ✅ | ❌ | ✅ | General |
| **Pasting** | ✅ | ❌ | ❌ | Large datasets |
| **Random Subspaces** | ❌ | ✅ | N/A | High-dimensional |
| **Random Patches** | ✅ | ✅ | ✅ / ❌ | High diversity |
| **Stratified Bagging** | ✅ (stratified) | ❌ | ✅ | Imbalanced data |

**Boosting ensemble learning**

[Boosting](https://www.geeksforgeeks.org/machine-learning/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/machine-learning/implementing-the-adaboost-algorithm-from-scratch/) Here is an overview of Boosting algorithm:

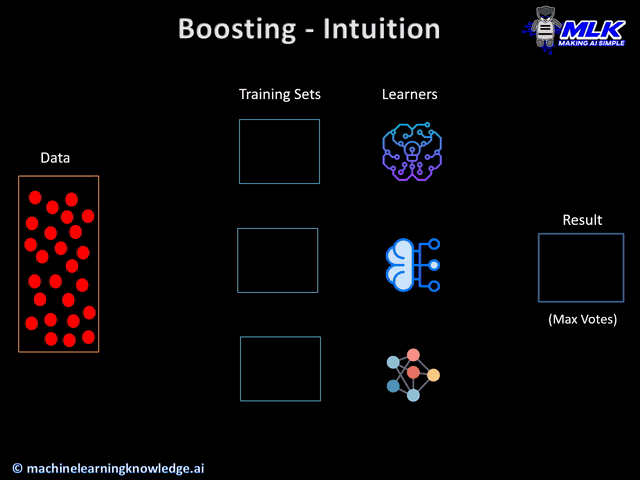
* **Initialize Model Weights**: Begin with a single weak learner and assign equal weights to all training examples.
* **Train Weak Learner**: Train weak learners on these dataset.
* **Sequential Learning**: Boosting works by training models sequentially where each model focuses on correcting the errors 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.

**Boosting** is an ensemble technique that converts a collection of **weak learners** into a **strong learner**.

* A **weak learner** is a model that performs only slightly better than random guessing (e.g., a shallow decision tree).
* A **strong learner** is a model that achieves high accuracy.

The **core idea** is to train models sequentially, where **each new model corrects the errors** made by the previous ones.

Boosting adopts a sequential approach, where the prediction of the current model is transferred to the next one. Each model iteratively focuses attention on the observations that are misclassified by its predecessors. We’ve outlined the general process below:



**Advantages**

* Similarly to bagging, boosting is easy to understand and implement. Furthermore, it does not require any preprocessing and can handle missing values in the data.
* It efficiently reduces bias.
* Boosting algorithms prioritize features that increase overall accuracy during the training. This process reduces the dimensionality of the data, hence reducing the computation time.

**Disadvantages**

* The sequential approach of boosting makes the next models correct the mistakes of its predecessor. This makes the overall model vulnerable to outliers.
* Boosting is not scalable for the same reason as the sequential aspect.

**⚙️ How Does Boosting Work?**

Let’s go step-by-step through the **general process** of boosting:

**1. Initialize Sample Weights (for Classification)**

In the beginning:

* All training samples are assigned **equal weights**.

For example, if there are 100 samples, each one has a weight of 1/100.

**2. Train First Weak Learner**

* A **weak learner** (e.g., a small decision tree) is trained on the data using these weights.
* The model tries to **minimize weighted error** (i.e., it focuses equally on all samples at first).

**3. Evaluate Weak Learner Performance**

* Identify which samples the model predicted **incorrectly**.
* Compute a **weighted error rate**:

Error=∑i=1nwi⋅I(yi≠h(xi))\text{Error} = \sum\_{i=1}^{n} w\_i \cdot I(y\_i \neq h(x\_i))Error=i=1∑n​wi​⋅I(yi​=h(xi​))

where:

* + wiw\_iwi​ is the weight of sample iii,
  + yiy\_iyi​ is the true label,
  + h(xi)h(x\_i)h(xi​) is the prediction,
  + III is the indicator function (1 if wrong, 0 otherwise).

**4. Increase Focus on Misclassified Samples**

* Misclassified samples are **given higher weights**.
* Correctly classified ones get **lower weights**.

This forces the next learner to **focus more on the "hard" cases**.

This is a key difference from bagging — boosting is **sequential and adaptive**.

**5. Train Next Weak Learner**

* The new learner is trained on the updated (reweighted) data.
* It attempts to correct the errors made by the previous learner.

**6. Combine All Learners**

Each weak learner is assigned a **vote or weight** based on its performance.

* More accurate models get **more influence** in the final prediction.

In classification:

* Final prediction is made using **weighted majority vote**.

In regression:

* Final prediction is a **weighted average**.

**Limitations of Boosting**

* **Sensitive to noise** (because it focuses on hard cases, which might be noisy).
* **Slower training** than bagging (due to sequential nature).
* Risk of **overfitting** if not regularized properly (especially in deeper trees).
* **can each weak learner have different algorithms for training data**

In **Boosting**, typically all weak learners are **of the same type of algorithm**, but they are trained **sequentially** on different versions of the data — each one learning from the mistakes of the previous. Here's how it works and why mixing algorithms is not common:

**🔁 General Boosting Flow**

1. **Start with an initial weak learner** (often a shallow decision tree).
2. Use it to predict.
3. Evaluate the **errors** — identify which instances were misclassified or poorly predicted.
4. **Adjust weights** of those instances so that the next learner focuses more on the difficult examples.
5. Add the next weak learner, which tries to correct the previous one’s mistakes.
6. **Repeat** this process, combining all learners (typically via weighted majority vote or sum).

**🤔 Why same algorithms are used?**

* **Consistency**: All learners must be able to accept **weighted data**, as boosting often assigns more weight to misclassified points.
* **Efficiency**: Algorithms like decision trees are fast, interpretable, and easy to adjust for weighted samples.
* **Mathematical Formulation**: Boosting frameworks (e.g., AdaBoost, Gradient Boosting) are designed assuming a base learner of fixed type, often a **decision stump** or small tree.

**🧠 Can we use different algorithms?**

In theory: **Yes**, you *can* mix algorithms. But in practice, this is:

* **Not Boosting** in its strict definition.
* More like **Stacking** or **Ensemble of Ensembles**, where diverse models are combined, often using a meta-learner.

**🔄 Boosting vs Bagging**

| **Feature** | **Boosting** | **Bagging** |
| --- | --- | --- |
| **Training** | Sequential | Parallel |
| **Goal** | Reduce bias | Reduce variance |
| **Focus** | Hard samples (adaptive) | All samples equally (random) |
| **Model Type** | Usually same base (e.g., stumps) | Usually same base |

**Categories of Boosting Techniques**

**1. AdaBoost (Adaptive Boosting)**

* **Core Idea**: Increases the weights of misclassified instances so that the next model focuses more on those.
* **Base Learner**: Weak models (usually decision stumps).
* **Libraries**: sklearn.ensemble.AdaBoostClassifier

from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier(n\_estimators=50)

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

**2. Gradient Boosting**

* **Core Idea**: Trains each new model to predict the residual errors (gradients) of previous models.
* **Base Learner**: Decision trees.
* **Libraries**: sklearn.ensemble.GradientBoostingClassifier

python

CopyEdit

from sklearn.ensemble import GradientBoostingClassifier

model = GradientBoostingClassifier(n\_estimators=100)

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

**3. XGBoost (Extreme Gradient Boosting)**

* **Core Idea**: Enhanced gradient boosting with regularization and optimizations like tree pruning, column sampling.
* **Advantages**: Fast, regularized, handles missing values.
* **Library**: xgboost

python

CopyEdit

from xgboost import XGBClassifier

model = XGBClassifier(n\_estimators=100, learning\_rate=0.1)

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

**4. LightGBM (Light Gradient Boosting Machine)**

* **Core Idea**: Uses histogram-based algorithm and leaf-wise tree growth for faster computation.
* **Advantages**: Very fast on large datasets.
* **Library**: lightgbm

python

CopyEdit

from lightgbm import LGBMClassifier

model = LGBMClassifier(num\_leaves=31)

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

**5. CatBoost**

* **Core Idea**: Specially optimized for categorical features; uses ordered boosting to reduce overfitting.
* **Advantages**: Handles categorical variables without one-hot encoding.
* **Library**: catboost

python

CopyEdit

from catboost import CatBoostClassifier

model = CatBoostClassifier(verbose=0)

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

**6. Stochastic Gradient Boosting**

* **Core Idea**: Adds randomness by sampling a subset of data/features to prevent overfitting.
* **Implemented in**: GradientBoostingClassifier with subsample < 1.0

python

CopyEdit

from sklearn.ensemble import GradientBoostingClassifier

model = GradientBoostingClassifier(subsample=0.8)

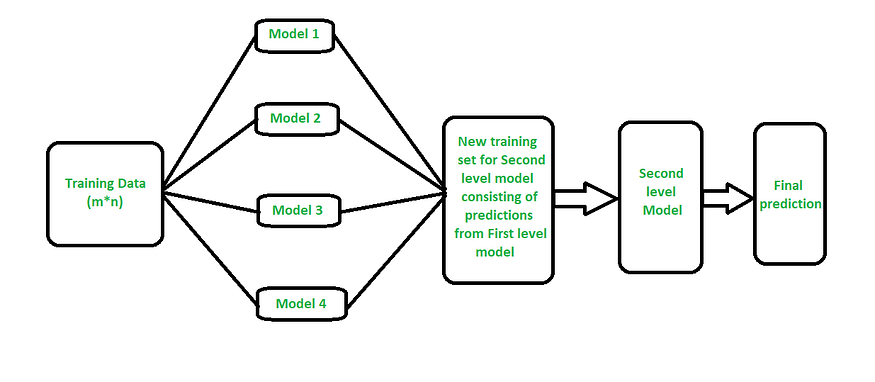
model.fit(X\_train, y\_train)

**Summary Table**

| **Technique** | **Regularization** | **Handles Cat Vars** | **Fast on Large Data** | **Handles Missing** |
| --- | --- | --- | --- | --- |
| AdaBoost | No | No | No | No |
| Gradient Boosting | Basic | No | Medium | No |
| XGBoost | Yes | Medium | Yes | Yes |
| LightGBM | Yes | Medium | ✅ Very Fast | Yes |
| CatBoost | Yes | ✅ Yes | Fast | Yes |
| Stochastic GB | Partial | No | Medium | No |

**Stacking ensemble learning**

Stacking is pretty similar to boosting. The predictions from the base learners are stacked together and are used as the input to train the meta learner to produce more robust predictions. The meta learner is then used to make final predictions. Bagging and boosting typically use homogeneous base learners, whereas stacking tends to include heterogeneous ones.



**Advantages**

* It leverages the strengths of multiple high-performing models for both classification and regression tasks.
* Similar to other ensemble models, stacking helps build a more accurate model than individual models used alone.

**Disadvantages**

* Using complex basic models can increase the risk of overfitting.
* Different levels of a model’s training can make the stacking architecture complex to implement.

**Stacking (or Stacked Generalization)** is an ensemble learning technique that combines predictions from **multiple base models (also called level-0 models)** using a **meta-model (or level-1 model)**. The goal is to improve generalization by **learning how to best combine the outputs** of diverse models.

**🔍 Why Use Stacking?**

* Different models make different types of errors.
* By combining them smartly, we can reduce bias **and** variance.
* Unlike bagging and boosting, stacking learns **how to combine** base models using a separate learner, rather than averaging or voting.

**🏗️ How Stacking Works — Step-by-Step**

**🔹 1. Train Base Learners (Level-0 models)**

* Select multiple machine learning algorithms (e.g., Decision Tree, SVM, KNN).
* Train each of them on the **original training dataset**.

**🔹 2. Generate Predictions for Training Meta-Model**

* To prevent **data leakage**, we cannot use predictions from models trained on the same data.
* So, we use **K-fold cross-validation**:
  + Split the training data into K folds.
  + For each fold:
    - Train base learners on K−1 folds.
    - Predict on the held-out fold.
* This creates a **new dataset of predictions** (one column per base learner), known as the **level-1 input dataset**.

**🔹 3. Train Meta-Learner (Level-1 model)**

* The meta-model (e.g., Logistic Regression or Gradient Boosting) is trained on the level-1 input dataset.
* Input: base models' predictions
* Target: true labels (same as original problem)

**🔹 4. Final Prediction**

* On **new test data**:
  + Base learners generate predictions.
  + These predictions are passed as input features to the meta-learner.
  + The meta-learner outputs the final prediction.

**💡 Key Concepts**

| **Concept** | **Explanation** |
| --- | --- |
| **Level-0 models** | Base learners (e.g., Random Forest, SVM, etc.) |
| **Level-1 model (meta-model)** | Learns to combine base learners’ predictions |
| **Out-of-fold predictions** | Predictions made on data not used during training of base models (avoids leakage) |
| **Generalization** | Meta-model captures when and how each base learner performs well or poorly |

**✅ Advantages of Stacking**

* **Highly flexible**: Can combine different algorithms.
* **Handles different error types**: Reduces generalization error.
* **Often more accurate** than bagging or boosting in many real-world tasks.

**❌ Disadvantages**

* **Slower**: Requires training multiple models.
* **Complexity**: More difficult to implement and tune.
* **Overfitting Risk**: Especially if meta-model is too complex or if out-of-fold procedure isn't followed carefully.

**1. Standard Stacking (Single-layer)**

* All base learners are trained on the same data.
* A meta-learner is trained on the predictions of these base learners.

**Based on Meta-Model Type**

| **Category** | **Description** |
| --- | --- |
| **Linear Meta-Model** | Uses linear regression, logistic regression, or ridge regression to blend base learners' outputs. |
| **Non-linear Meta-Model** | Uses tree-based models (e.g., random forest, gradient boosting), SVMs, or even neural networks. |
| **Trainable Neural Net Blender** | Deep learning model as the meta-model — can learn complex relations between base predictions. |

from sklearn.ensemble import StackingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

base\_learners = [

('dt', DecisionTreeClassifier()),

('svc', SVC(probability=True))

]

meta\_learner = LogisticRegression()

stack\_model = StackingClassifier(estimators=base\_learners, final\_estimator=meta\_learner)

**🔹 2. Blending (Variant of Stacking)**

* Uses a **validation set** to train the meta-learner rather than cross-validation.
* Simpler but may underfit due to limited meta-training data.

**Based on Base Learner Diversity**

| **Category** | **Description** |
| --- | --- |
| **Homogeneous Stacking** | All base models are of the same type (e.g., multiple decision trees with different hyperparams). |
| **Heterogeneous Stacking** | Base models are of different algorithms (e.g., SVM, Random Forest, Logistic Regression). Most common in real-world stacks. |

# Split original train set into train and validation

# Base learners on training set, meta learner on validation predictions

**🔹 3. Multi-layer Stacking (Deep Stacking)**

* Outputs of first-level stacking are input to second-level stacking (stack of stacks).
* Can increase accuracy but adds complexity.

**Based on Training Strategy**

| **Category** | **Description** |
| --- | --- |
| **Two-Level Stacking** | Basic setup: level-0 base learners and level-1 meta-learner. |
| **Multi-Level (Hierarchical) Stacking** | More than two levels, where output of meta-models feeds into another level. Useful for large ensembles. |
| **Blending** | A variant of stacking where meta-model is trained on a **holdout set**, rather than cross-validation folds. Faster but less robust. |

python

CopyEdit

# Level 1

stack\_1 = StackingClassifier(

estimators=[('svc', SVC(probability=True)), ('dt', DecisionTreeClassifier())],

final\_estimator=LogisticRegression()

)

# Level 2

stack\_2 = StackingClassifier(

estimators=[('stack1', stack\_1), ('rf', RandomForestClassifier())],

final\_estimator=GradientBoostingClassifier()

)

**🔹 4. Cross-validated Stacking**

* Instead of using predictions on training data, we use **out-of-fold (OOF)** predictions during cross-validation to train the meta-learner.
* Prevents overfitting.

**Based on Data Splitting & Cross-Validation**

| **Category** | **Description** |
| --- | --- |
| **K-Fold Cross-Validated Stacking** | Each base learner is trained on k-1 folds and predictions are made on the remaining fold. Prevents data leakage. |
| **Holdout-Based Stacking (Blending)** | Reserve a small part of the data for training the meta-learner. Simpler but might underfit due to less training data. |

stack\_model = StackingClassifier(

estimators=base\_learners,

final\_estimator=meta\_learner,

cv=5 # enables OOF stacking

)

**🔹 5. Heterogeneous vs Homogeneous Stacking**

* **Heterogeneous**: Different types of base models (SVM, Tree, Logistic, etc.)
* **Homogeneous**: All base models are of same type but with different hyperparameters.

**Based on Output Type**

| **Category** | **Description** |
| --- | --- |
| **Hard Voting / Classification** | Meta-model combines class labels from base models. |
| **Soft Voting / Probabilistic** | Combines predicted class probabilities instead of final labels. |
| **Regression Output** | Used for regression problems — meta-model combines numerical predictions. |

# Homogeneous stacking

base\_learners = [

('rf1', RandomForestClassifier(n\_estimators=10)),

('rf2', RandomForestClassifier(n\_estimators=100))

]

**🔹 6. Stacking with Feature Engineering**

* Instead of just using predictions, add meta-features (like confidence scores, class probabilities, or engineered features).

python

CopyEdit

# Use predict\_proba outputs as input to meta-learner

StackingClassifier(..., passthrough=True)

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.tree import DecisionTreeClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import StackingClassifier, RandomForestClassifier

from sklearn.metrics import accuracy\_score

# Load and prepare dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Standardize for PCA

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Reduce to 2D using PCA for visualization

pca = PCA(n\_components=2)

X\_2d = pca.fit\_transform(X\_scaled)

# Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_2d, y, test\_size=0.3, random\_state=42, stratify=y)

# --- Stacking Classifier Setup ---

base\_models = [

('knn', KNeighborsClassifier(n\_neighbors=5)),

('dt', DecisionTreeClassifier(max\_depth=3)),

('nb', GaussianNB())

]

meta\_model = LogisticRegression()

stack\_model = StackingClassifier(

estimators=base\_models,

final\_estimator=meta\_model,

cv=5

)

# --- Comparison Model ---

rf\_model = RandomForestClassifier(n\_estimators=100, random\_state=42)

# --- Fit models ---

stack\_model.fit(X\_train, y\_train)

rf\_model.fit(X\_train, y\_train)

# --- Predict ---

y\_pred\_stack = stack\_model.predict(X\_test)

y\_pred\_rf = rf\_model.predict(X\_test)

print(f"Stacking Accuracy: {accuracy\_score(y\_test, y\_pred\_stack):.4f}")

print(f"Random Forest Accuracy: {accuracy\_score(y\_test, y\_pred\_rf):.4f}")

# --- Plotting Decision Boundaries ---

def plot\_decision\_boundary(model, X, y, title, subplot\_index):

h = .02 # step size

x\_min, x\_max = X[:, 0].min() - .5, X[:, 0].max() + .5

y\_min, y\_max = X[:, 1].min() - .5, X[:, 1].max() + .5

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),

np.arange(y\_min, y\_max, h))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.subplot(1, 2, subplot\_index)

plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)

scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k', cmap=plt.cm.coolwarm)

plt.title(title)

plt.xlabel('PCA 1')

plt.ylabel('PCA 2')

plt.legend(\*scatter.legend\_elements(), title="Classes", loc='upper right')

# --- Plot both models side by side ---

plt.figure(figsize=(14, 6))

plot\_decision\_boundary(stack\_model, X\_test, y\_test, "Stacking Classifier Decision Boundary", 1)

plot\_decision\_boundary(rf\_model, X\_test, y\_test, "Random Forest Decision Boundary", 2)

plt.tight\_layout()

plt.show()

**Blending ensemble learning**

Blending is similar to Stacking. In blending, the structure of the data is made of training, hold-out, and test data. The meta learners are trained on the training data. Then their predictions are combined with the hold-out data to build the final meta model, which uses the test data to make the final predictions.

**Advantages**

* Like stacking and many other ensemble methods, blending can boost the final model’s performance in many cases.
* This use of this technique has been successful in winning many competitions.

**Disadvantages**

* Dividing the original data according to the blending architecture can limit the use of data for training base models, hence resulting in poor performance.
* Like other ensemble models, the final model interpretability is reduced due to increased complexity, hence making it difficult to draw any crucial business insights.

**Blending Ensemble Learning: working explanation**

Blending is a simplified variant of **stacking** and shares the same motivation:

Combine multiple base learners to improve prediction performance by learning from their aggregated output.

**🔹 1. Basic Concept**

Blending uses:

* A **training set** to train multiple base models (level-0 models).
* A **holdout validation set** (a small part of training data) to **generate predictions** from those base models.
* These predictions become the input for a **meta-model** (level-1 model), which learns how to best combine them.

**🔹 2. Step-by-Step Workflow**

**Step 1: Split the training data**

Split original training set into:

* Train set (e.g., 80%): used to train the base learners.
* Validation set (e.g., 20%): used to make predictions from those base learners for training the meta-model.

**Step 2: Train base learners**

Train multiple different models (e.g., logistic regression, decision tree, SVM) on the **train set**.

**Step 3: Generate meta-features**

Use the **base learners** to predict on the **validation set** (not the train set), and collect those predictions:

* For regression: direct predicted values.
* For classification: class labels or probabilities.

This forms the **meta-feature matrix** (rows = validation samples, columns = predictions from each model).

**Step 4: Train the meta-learner**

Train a meta-model (e.g., logistic regression or XGBoost) on this meta-feature matrix using true labels from the **validation set**.

**Step 5: Final Prediction (Test phase)**

* Base learners predict on the **test set**.
* Meta-model uses these base predictions to give the final prediction.

**🔹 3. Key Characteristics of Blending**

| **Feature** | **Description** |
| --- | --- |
| **Holdout** | Uses a fixed validation set instead of cross-validation. |
| **Simplicity** | Easier and faster to implement than stacking. |
| **Overfitting Risk** | Can overfit if validation set is small or not representative. |
| **No Data Leakage** | Unlike stacking without OOF predictions, blending avoids training meta-learner on the same data used for base learners' training. |
| **Speed** | No cross-validation; generally faster. |

**🔹 4. Blending vs Stacking**

| **Feature** | **Stacking** | **Blending** |
| --- | --- | --- |
| Meta-learner training data | OOF predictions (cross-validated) | Predictions on validation set |
| Risk of overfitting | Lower (uses more data via CV) | Higher (validation set is small) |
| Implementation complexity | More complex | Simpler |
| Speed | Slower (due to CV) | Faster |

**🔹 5. Example Analogy**

Imagine you have 3 weather apps predicting tomorrow’s temperature:

* You observe their performance over the last few days (validation set).
* You learn that App A is mostly right when App B is wrong, and App C is usually off.
* Based on these patterns, you train yourself (meta-learner) to weight their forecasts accordingly for better accuracy.

**Categories of Blending Techniques with Code Snippets**

Blending techniques vary by **how they split data** and **how they use predictions to train the meta-learner**.

**✅ 1. Simple Blending (Holdout-Based Blending)**

**📌 Idea:**

Use a fixed validation set to collect base learner predictions, then train the meta-learner on that.

**🧩 Code Snippet:**

python

CopyEdit

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

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.datasets import make\_classification

from sklearn.metrics import accuracy\_score

import numpy as np

# Step 1: Prepare data

X, y = make\_classification(n\_samples=1000, n\_features=20, random\_state=42)

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

# Step 2: Holdout split for blending

X\_base\_train, X\_blend\_val, y\_base\_train, y\_blend\_val = train\_test\_split(X\_train\_full, y\_train\_full, test\_size=0.3, random\_state=42)

# Step 3: Base learners

clf1 = RandomForestClassifier(n\_estimators=50, random\_state=1)

clf2 = SVC(probability=True, random\_state=1)

# Train on base\_train

clf1.fit(X\_base\_train, y\_base\_train)

clf2.fit(X\_base\_train, y\_base\_train)

# Predict on blend\_val to generate meta-features

meta\_X = np.column\_stack([

clf1.predict\_proba(X\_blend\_val)[:, 1],

clf2.predict\_proba(X\_blend\_val)[:, 1]

])

# Step 4: Train meta-learner

meta\_clf = LogisticRegression()

meta\_clf.fit(meta\_X, y\_blend\_val)

# Final test prediction

meta\_X\_test = np.column\_stack([

clf1.predict\_proba(X\_test)[:, 1],

clf2.predict\_proba(X\_test)[:, 1]

])

final\_preds = meta\_clf.predict(meta\_X\_test)

print("Accuracy:", accuracy\_score(y\_test, final\_preds))

**✅ 2. Weighted Blending**

**📌 Idea:**

Manually assign weights to each model based on validation performance.

**🧩 Code Snippet:**

python

CopyEdit

# Assume models already trained and we are on test set

pred1 = clf1.predict\_proba(X\_test)[:, 1]

pred2 = clf2.predict\_proba(X\_test)[:, 1]

# Weighted average of predictions

final\_pred = (0.7 \* pred1 + 0.3 \* pred2) > 0.5

print("Weighted Blend Accuracy:", accuracy\_score(y\_test, final\_pred))

**✅ 3. Blending with Probability Averaging (Soft Voting)**

**📌 Idea:**

Instead of training a meta-model, simply average the predicted probabilities of base models.

**🧩 Code Snippet:**

python

CopyEdit

# Soft Voting

avg\_probs = (pred1 + pred2) / 2

final\_pred\_soft = avg\_probs > 0.5

print("Soft Voting Accuracy:", accuracy\_score(y\_test, final\_pred\_soft))

**✅ 4. Blending with Rank Averaging**

**📌 Idea:**

Instead of raw probabilities, average the **rank** of predictions.

**🧩 Code Snippet:**

python

CopyEdit

from scipy.stats import rankdata

rank1 = rankdata(pred1)

rank2 = rankdata(pred2)

avg\_rank = (rank1 + rank2) / 2

final\_pred\_rank = avg\_rank > np.median(avg\_rank)

print("Rank Averaging Accuracy:", accuracy\_score(y\_test, final\_pred\_rank))

**✅ 5. Blending with Neural Net as Meta-Learner**

**📌 Idea:**

Use a small neural net instead of logistic regression for combining meta-features.

**🧩 Code Snippet:**

python

CopyEdit

from sklearn.neural\_network import MLPClassifier

meta\_nn = MLPClassifier(hidden\_layer\_sizes=(5,), max\_iter=500, random\_state=42)

meta\_nn.fit(meta\_X, y\_blend\_val)

final\_pred\_nn = meta\_nn.predict(meta\_X\_test)

print("NN Meta Accuracy:", accuracy\_score(y\_test, final\_pred\_nn))

**Voting ensemble learning**

In voting, multiple models are trained independently, and their predictions are combined to make a final prediction using either hard voting, soft voting, or weighted voting:

* In hard voting, the final prediction is the most common prediction from all the models.
* For soft voting, each model generates a probability distribution instead of a binary prediction. Then, the class with the highest probability is the one predicted.
* Finally, in weighted voting, there is an assumption that some models have more skill than other,s and those models are assigned with more contribution when making predictions.

**Advantages**

* Voting architecture is simple to implement compared to stacking, and blending. Also, it does not require complex fine-tuning.
* Using multiple base learners in the voting makes it less susceptible to the influence of individual models, which contributes to making more stable and reliable predictions.

**Disadvantages**

* It can be difficult to deal with models’ prediction conflict, which makes it hard to make the final decision in a meaningful way.
* Adding more models to the ensemble voting model does not necessarily improve the final performance.

**Cascading**

Cascading uses a stacking approach but with only one model in each layer. The first model is trained on the whole training data, and the next model is trained on the output of the model before. The goal of using the strategy is to learn complex patterns from the data, hence allow the model to make better predictions.

**Advantages**

* The next model is specialized in the output of the previous model, which reduces noise in the data.
* More robust to overfitting and can perform well on real-world data.

**Disadvantages**

* Implementing the cascading can be complex since it involves training each of the modes in the sequence.
* Getting the optimal cascading architecture can require a lot of experimentation and fine-tuning.

**Theoretical Explanation: internal working**

**Voting** is an ensemble technique where predictions from multiple models (often called **base learners**) are combined to make a final prediction. This method is mainly used for classification problems.

There are **two main types of voting**:

**✅ 1. Hard Voting**

* In **Hard Voting**, each base classifier predicts a class label.
* The final prediction is the **class that gets the majority vote**.
* It’s like a democracy—**the class that more models agree on wins**.

**Example:**

If three models predict the following:

* Model 1 → Class A
* Model 2 → Class B
* Model 3 → Class A  
  Final result → Class A (because A got 2 votes)

**✅ 2. Soft Voting**

* In **Soft Voting**, each classifier provides a **probability** for each class instead of a hard label.
* The final prediction is made by **averaging the predicted class probabilities** and selecting the class with the highest average probability.
* Soft voting often performs **better than hard voting**, especially when base models are well-calibrated.

**📊 When to Use Voting?**

* When you have **diverse models** that capture different data characteristics.
* Works well when base learners are **independent and not highly correlated**.
* Useful as a **simple and interpretable** ensemble method.

**⚖️ Summary: Hard vs Soft Voting**

| **Feature** | **Hard Voting** | **Soft Voting** |
| --- | --- | --- |
| Based on | Predicted class | Predicted probabilities |
| Output | Majority class | Class with highest avg prob |
| Requirement | Just predictions | Class probabilities |
| Performance | Often less accurate | Often more accurate |

**Voting Ensemble Learning Techniques with Code Snippets**

We'll use the VotingClassifier from scikit-learn.

**1. Hard Voting Classifier**

Each model votes for a class label, and the most common label is chosen.

python

CopyEdit

from sklearn.ensemble import VotingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

voting\_clf\_hard = VotingClassifier(

estimators=[

('lr', LogisticRegression()),

('dt', DecisionTreeClassifier()),

('svc', SVC())

],

voting='hard' # majority class

)

**2. Soft Voting Classifier**

Each model gives class probabilities; the probabilities are averaged.

python

CopyEdit

voting\_clf\_soft = VotingClassifier(

estimators=[

('lr', LogisticRegression(probability=True)),

('dt', DecisionTreeClassifier()),

('svc', SVC(probability=True))

],

voting='soft' # average predicted probs

)

**3. Weighted Soft Voting**

Give more weight to more accurate classifiers.

python

CopyEdit

voting\_clf\_weighted = VotingClassifier(

estimators=[

('lr', LogisticRegression(probability=True)),

('dt', DecisionTreeClassifier()),

('svc', SVC(probability=True))

],

voting='soft',

weights=[1, 1, 2] # more weight to SVC

)

**4. Voting with Heterogeneous Models**

Mixing completely different types of models to boost diversity.

python

CopyEdit

from sklearn.naive\_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

voting\_clf\_heterogeneous = VotingClassifier(

estimators=[

('gnb', GaussianNB()),

('knn', KNeighborsClassifier(n\_neighbors=5)),

('svc', SVC(probability=True))

],

voting='soft'

)

**5. Voting with GridSearchCV**

Used to optimize the best parameters for each base estimator before voting.

python

CopyEdit

from sklearn.model\_selection import GridSearchCV

param\_grid = {'svc\_\_C': [0.1, 1, 10]}

grid\_voting = GridSearchCV(

estimator=voting\_clf\_soft,

param\_grid=param\_grid,

cv=5

)

**6. Voting within a Pipeline**

Used when you want to preprocess data before feeding it into the voting ensemble.

python

CopyEdit

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

pipeline = Pipeline([

('scaler', StandardScaler()),

('voting', voting\_clf\_soft)

])

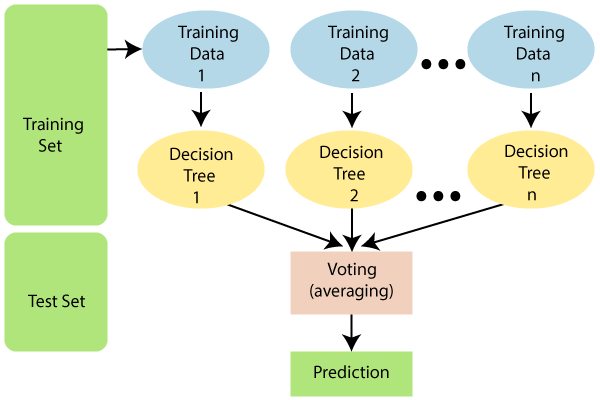
**Overview of Ensemble Algorithms**

The previous sections covered the different types of ensemble models. Now, let’s have a brief overview of some popular models.

**Random Forest**

Random Forest is a commonly used model that can solve both classification and regression problems. A random forest is made up of many decision trees that are trained using bagging. Its outcome is determined by taking the average of the prediction of individual trees.

The node size, the number of trees, and number of features sampled are the three main hyperparameters that need to be set before training the random forest.



**What about randomness in a random forest?**

Feature bagging, also known as feature randomness, creates a random sample of features to ensure low correlation among decision trees. This approach sets apart random forests from decision trees which consider all the possible feature splits, whereas random forests consider only a subset of those features. Read in our [**random forest classification tutorial**](https://www.datacamp.com/tutorial/random-forests-classifier-python).

**XGBoost**

Extreme Gradient Boosting, or XGBoost for short, is used for both classification and regression. XGBoost is designed to be scalable and highly efficient, and it implements the gradient boosting decision trees framework.

It is suitable for processing large-scale data sets, and is compatible with major distributed environments such as Hadoop, MPI (Message Passing Inference), and SGI (Sun Grid Engine). You can learn more in our **[XGBoost tutorial](https://www.datacamp.com/tutorial/xgboost-in-python" \t "_blank)**.

**AdaBoost**

Adaptive Boosting, or AdaBoost, is one of the first ensemble boosting classifiers for successful boosting algorithms for binary classification. It is adaptive as the weights are re-assigned to every instance, and misclassified instances are assigned with higher weights. Read more about [**AdaBoost Classification in Python**](https://www.datacamp.com/tutorial/adaboost-classifier-python) in our separate tutorial.

**A Step-by-Step Implementation of Ensemble Models**

Now you understand the main idea of ensemble modeling and how some models work, this section will focus on the technical implementation using Python.

**Setting up the environment**

To begin, you will need to have[**Python**](https://www.python.org/downloads/) installed on your computer, along with the following libraries:

* pandas for loading the data frame.
* Scikit-learn to use the existing machine learning algorithms.
* Seaborn and matplotlib for visualization purposes.

You can install these libraries using pip, the Python package manager, as follows:

pip install scikit-learn

pip install pandas

pip install matplotlib seaborn

[**Powered By**](https://www.datacamp.com/datalab)

To better illustrate the use case, we will be using the Loan Data available on **[DataLab](https://www.datacamp.com/workspace/datasets/dataset-r-loans" \t "_blank)**. The code for this tutorial is also available on DataLab in [**this workbook**](https://app.datacamp.com/workspace/w/6c0ec223-3bb2-4a0f-a4a8-9f35bccb2673/edit); you can create your own workbook copy and edit and run the code in your browser without needing to install anything on your computer.

**Understanding the data**

The data set has 9,500 loans with information on the loan structure, the borrower, and whether the loan was paid back in full, represented by the not.fully.paid column. The first five observations are shown below:

import pandas as pd

loan\_data = pd.read\_csv("loan\_data.csv")

loan\_data.head()

[**Powered By**](https://www.datacamp.com/datalab)

Using the info() function generates the following relevant information about the data:

* 9,578 rows (from 0 to 9577), and 14 columns of numerical types except for purpose, which is object and gives the textual description of the purpose of the load.
* The non-null tag for each column means that there is no missing information in that specific column in none of the columns.

loan\_data.info()

[**Powered By**](https://www.datacamp.com/datalab)

We notice an imbalanced data scenario, where there are many more observations with class 0 (8045 observations) than there are for class 1 (only 1533 observations).

print(loan\_data['not.fully.paid'].value\_counts())

loan\_data['not.fully.paid'].value\_counts().plot(kind='barh')

[**Powered By**](https://www.datacamp.com/datalab)

Using such imbalanced data for model training affects the overall performance.

One way of tackling this issue is using the undersampling approach, meaning we are reducing the number of the majority observations (class 0). In this scenario, there will be as many observations for class 0 as there are for class 1, and the process is as follows:

* First get the number of class 1.
* Select N random observations from class 0, where N is the size of the dataframe of class 1.
* Concatenate the previous two dataframes.

loan\_data\_class\_1 = loan\_data[loan\_data['not.fully.paid'] == 1]

number\_class\_1 = len(loan\_data\_class\_1)

loan\_data\_class\_0 = loan\_data[loan\_data['not.fully.paid'] == 0].sample(number\_class\_1)

final\_loan\_data = pd.concat([loan\_data\_class\_1,

loan\_data\_class\_0])

print(final\_loan\_data.shape)

[**Powered By**](https://www.datacamp.com/datalab)

The [**Diving Deep with Imbalanced Data**](https://www.datacamp.com/tutorial/diving-deep-imbalanced-data) tutorial provides more resources to learn the techniques to deal with an imbalance dataset.

After the undersampling, we get:

* Overall 3066 observations and 14 columns.

And the final balanced distribution is shown below.

**Prepare the data for training**

By looking at the data set, we notice that not all the features have the same scale, and some machine learning models such as KNN are sensitive to such a scaling. This issue can be addressed by normalizing the ranges of the features to the same scale using the ***MinMaxScaler*** module. In this case, between 0 and 1.

Also, we will remove the purpose column for simplicity’s sake since it requires additional preprocessing.

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature\_range=(0, 1))

# Remove unwanted 'purpose' column and get the data

final\_loan\_data.drop('purpose', axis=1, inplace=True)

X = final\_loan\_data.drop('not.fully.paid', axis=1)

normalized\_X = scaler.fit\_transform(X)

[**Powered By**](https://www.datacamp.com/datalab)

Like in any predictive task, we need to split the data into training and testing sets. In this case, we will use 77% for training and the remaining 33% for testing.

* Random state attribute is initialized with a random value (2023 in our case) to ensure the reproducibility of the results.

***Stratify*** is used to ensure that the y target value is evenly distributed in both training and testing data.

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

y = final\_loan\_data['not.fully.paid']

r\_state = 2023

t\_size = 0.33

X\_train, X\_test, y\_train, y\_test = train\_test\_split(normalized\_X, y,

test\_size=t\_size,

random\_state=r\_state,

stratify=y)

[**Powered By**](https://www.datacamp.com/datalab)

All the models will follow the sequence of training on the training data, make the prediction on the testing data and evaluate the models’ performance.

**Bagging model**

Random Forest is the bagging model used in this section. We will use the default parameters for simplicity.

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import cross\_val\_score

# Define the model

random\_forest\_model = RandomForestClassifier()

# Fit the random search object to the data

random\_forest\_model.fit(X\_train, y\_train)

[**Powered By**](https://www.datacamp.com/datalab)

After training the model, the performance is generated using the **accuracy\_score** function as follows:

# Make predictions

y\_pred = random\_forest\_model.predict(X\_test)

# Get the performance

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

print("Accuracy:", accuracy)

[**Powered By**](https://www.datacamp.com/datalab)

The previous **print**statement generates 0.6156 which is 61.56%

Now let's compare this score to the performance of a blending aggregation approach.

**Blending Model**

For blending, we will use two base models: a decision tree and a K-Nearest Neighbors classifier. A final regression model is used to make the final predictions.

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.linear\_model import LogisticRegression

[**Powered By**](https://www.datacamp.com/datalab)

The original training data is split into a new training data set and a validation data set. The base models are trained on the training data.

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

X\_train, y\_train,

test\_size=t\_size,

random\_state=r\_state)

[**Powered By**](https://www.datacamp.com/datalab)

Two dataframes are then generated from the predictions of those base models:

* First dataframe  with the predictions on the validation data concatenated with the original validation data.
* Second dataframe with the predictions on the test data concatenated with the original test data.

# Decision Tree Model

dt\_model = DecisionTreeClassifier()

dt\_model.fit(X\_train, y\_train)

dt\_model\_pred\_val = dt\_model.predict(X\_val)

dt\_model\_pred\_test= dt\_model.predict(X\_test)

dt\_model\_pred\_val = pd.DataFrame(dt\_model\_pred\_val)

dt\_model\_pred\_test = pd.DataFrame(dt\_model\_pred\_test)

# KNN Model

knn\_model = KNeighborsClassifier()

knn\_model.fit(X\_train,y\_train)

knn\_model\_pred\_val = knn\_model.predict(X\_val)

knn\_model\_pred\_test = knn\_model.predict(X\_test)

knn\_model\_pred\_val = pd.DataFrame(knn\_model\_pred\_val)

knn\_model\_pred\_test = pd.DataFrame(knn\_model\_pred\_test)

[**Powered By**](https://www.datacamp.com/datalab)

The final logistic regression model is built using the concatenated validation data, and evaluated using the concatenated test data.

x\_val = pd.DataFrame(X\_val)

x\_test = pd.DataFrame(X\_test)

df\_val\_lr = pd.concat([x\_val, knn\_model\_pred\_val,

dt\_model\_pred\_val], axis=1)

df\_test\_lr = pd.concat([x\_test, dt\_model\_pred\_test,

knn\_model\_pred\_test],axis=1)

# Logistic Regression Model

lr\_model = LogisticRegression()

lr\_model.fit(df\_val\_lr,y\_val)

lr\_model.score(df\_test\_lr,y\_test)

[**Powered By**](https://www.datacamp.com/datalab)

Using the .score() function computes the accuracy score by default and does not need actual predictions.

The performance result is 0.6383, which is 63.83% and is around 2% higher than the initial random forest; hence the blending model provides the best performance.

Our course [**Ensemble Methods in Python**](https://datacamp.com/courses/ensemble-methods-in-python)and [**Machine Learning with Tree-Based Models in R**](https://datacamp.com/courses/machine-learning-with-tree-based-models-in-r) could be a great next step to continue your machine learning journey by diving into the wonderful world of ensemble classifier methods, respectively in Python and R.

**Learn more with coding:**

URL: <https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/>

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

<https://github.com/campusx-official/voting-ensemle>

<https://github.com/campusx-official/bagging-ensemble>

try first:- <https://medium.com/@sumbatilinda/ensemble-learning-in-machine-learning-bagging-boosting-and-stacking-a00c6bae971f>

**Pay attention on below terms:**

Row sampling, column sampling With or without replacement

Bagging, Pasting, Random subspaces, Random Patches,

Bagging with classification and Regression

Voting ensemble with Classification and Regression

GridSearchCV to test with multiple different hyperparameter values

When to use ensemble learning