**TASK**

1. In Python, inheriting form the appropriate **scikit-learn** base classes,

define a new **HyperParamEnsembleClassifier** model that implements

an ensemble models using a single base estimator type, and different

hyperparameter combinations for each model in the ensemble.

• The model should work for multiclass classification problems and

accept numeric inputs.

• The class should implement training the model by training each of

its member models (implemented in the **fit** method) and making

predictions (implemented through both the **predict** and

**predict\_proba** methods) using voting.

• Parameters passed to the model constructor should include the

base estimator to use, a Python dictionary defining the

hyperparamter ranges to select form (following the style used by

scikit-learn for hyper-parameter tuning), and the number of

estimators to include

you can

use existing scikit-learn implementations for performing tasks like base

estimators, cross validation, grid searches, and measuring performance.

Similarly, **all scikit-learn models are built on top of numpy and it provides**

**many useful utility functions.**

**Explanations**

In **Scikit-Learn**, an **estimator** refers to any object that can learn from data (i.e., a model). Estimators include classifiers (for classification tasks), regressors (for regression tasks), and transformers (for preprocessing data).

### ****What is a Base Estimator?****

A **base estimator** is simply a machine learning model like:

* **Decision Tree (**DecisionTreeClassifier**)**
* **Naïve Bayes (**GaussianNB**)**
* **k-NN (**KNeighborsClassifier**)**
* **Support Vector Machine (**SVC**)**
* **Logistic Regression (**LogisticRegression**)**

In your assignment, the **base estimator** refers to the fundamental model that your **HyperParamEnsembleClassifier** will use to build multiple models with different hyperparameter settings.

### ****What are**** BaseEstimator ****and**** ClassifierMixin ****in Scikit-Learn?****

These two classes **help you create custom machine learning models** that work like Scikit-Learn's built-in models.

📌 **Why use them?**

* BaseEstimator → **Makes your model compatible with Scikit-Learn tools** (e.g., GridSearchCV and Pipeline).
* ClassifierMixin → **Gives your model classification-specific methods**, like score().

### ****🔹 Step-by-Step: How to Implement a Custom Classifier****

#### ****1️⃣ Start by Importing the Base Classes****

python

CopiaModifica

from sklearn.base import BaseEstimator, ClassifierMixinimport numpy as np

#### ****2️⃣ Define Your Custom Class****

📌 **Your model must inherit from** BaseEstimator **and** ClassifierMixin**.**

python

CopiaModifica

class MyClassifier(BaseEstimator, ClassifierMixin):

def \_\_init\_\_(self, some\_param=1):

"""Initialize the model with parameters"""

self.some\_param = some\_param

* \_\_init\_\_() → This sets the initial **hyperparameters** of the model.
* some\_param is just an example—your real model will have actual hyperparameters.

#### ****3️⃣ Implement the**** fit() ****Method****

📌 **This method "trains" the model (even if it's simple).**

python

CopiaModifica

def fit(self, X, y):

"""Train the model"""

self.classes\_ = np.unique(y) # Stores unique class labels

return self # Return the trained model

* X → The input data (features).
* y → The target labels (classification categories).
* np.unique(y) stores the different class labels (e.g., ['cat', 'dog', 'rabbit']).

#### ****4️⃣ Implement the**** predict() ****Method****

📌 **This method makes predictions after training.**

python

CopiaModifica

def predict(self, X):

"""Predict the class of each sample in X"""

random\_predictions = np.random.choice(self.classes\_, size=len(X))

return random\_predictions

* Here, we just **randomly assign** a class (this is just an example).
* In a real classifier, you would **use trained logic** instead.

#### ****5️⃣ Implement the**** score() ****Method (from**** ClassifierMixin****)****

When you inherit from ClassifierMixin, your class **automatically gets the** score() **method**:

class HyperParamClassifier(BaseEstimator, ClassifierMixin):

By default, ClassifierMixin.score() works like this:

python

CopiaModifica

def score(self, X, y):

"""Compute accuracy score (default for classifiers)."""

return accuracy\_score(y, self.predict(X))

* This **compares predictions to actual values** and returns **accuracy**.

### ****🔹 Final Custom Classifier Code****

python

CopiaModifica

from sklearn.base import BaseEstimator, ClassifierMixinimport numpy as np

class MyClassifier(BaseEstimator, ClassifierMixin):

def \_\_init\_\_(self, some\_param=1):

"""Initialize the model with parameters"""

self.some\_param = some\_param

def fit(self, X, y):

"""Train the model"""

self.classes\_ = np.unique(y) # Store unique class labels

return self # Return trained model

def predict(self, X):

"""Predict the class of each sample in X"""

random\_predictions = np.random.choice(self.classes\_, size=len(X))

return random\_predictions

def score(self, X, y):

"""Compute accuracy of the model"""

predictions = self.predict(X) # Get predictions

return np.mean(predictions == y) # Compare predictions to true labels

### ****🔹 How This Relates to Your Assignment****

In your HyperParamEnsembleClassifier, you need to:

* **Inherit from** BaseEstimator **and** ClassifierMixin.
* **Define** fit() to train multiple models (with different hyperparameters).
* **Define** predict() to make final predictions using voting.
* **Use** score() to evaluate model accuracy.

### ****What Does "Voting" Mean Here?****

In your assignment, **"voting"** refers to how your ensemble model will **combine predictions** from multiple KNN models to make a **final decision**. Since you will train **multiple KNN classifiers with different hyperparameters**, each one will give a different prediction for the same input. The goal is to **combine** these predictions in a smart way.

### ****🔹 Types of Voting in Ensemble Learning****

There are **two main ways** to perform voting in your HyperParamEnsembleClassifier:

#### ****1️⃣ Hard Voting (Majority Vote) – Used in**** predict()

* Each KNN model predicts a **class label** (e.g., **Class A or Class B**).
* The **most common (majority) prediction** among all models is chosen.
* If **5 models predict "A"** and **3 models predict "B"**, the final result is **A**.

📌 **Example:**

| **Model 1** | **Model 2** | **Model 3** | **Model 4** | **Model 5** | **Final Prediction** |
| --- | --- | --- | --- | --- | --- |
| A | A | B | A | B | **A** (Majority) |

📌 **Resource:** <https://scikit-learn.org/stable/modules/ensemble.html#voting-classifier>

#### ****2️⃣ Soft Voting (Probability Averaging) – Used in**** predict\_proba()

* Instead of choosing the most frequent class, each model provides **probabilities** for each class.
* These probabilities are **averaged across all models**.
* The class with the highest **average probability** is selected.

📌 **Example:**

| **Model 1 (A: 0.7, B: 0.3)** | **Model 2 (A: 0.6, B: 0.4)** | **Model 3 (A: 0.8, B: 0.2)** | **Final Probability (A, B)** |
| --- | --- | --- | --- |
| **A (70%)** | **A (60%)** | **A (80%)** | **A: (0.7+0.6+0.8)/3 = 0.7** (Final: A) |

📌 **Resource:** https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

### ****🔹 What You Need to Implement****

* **In** predict() → Use **hard voting** (majority vote).
* **In** predict\_proba() → Use **soft voting** (probability averaging).

The assignment requires you to:

**Choose a Base Classifier**

* 1. This can be any standard Scikit-Learn classifier like **Decision Tree, k-NN, Naïve Bayes, Logistic Regression, or SVM**.
  2. The **base estimator** will be the fundamental model that your ensemble will use multiple times with different hyperparameter settings.

**Implement the Ensemble with Different Hyperparameters**

* 1. Instead of tuning one model, **train multiple instances** of the chosen base model using different hyperparameter settings.
  2. Each model in the ensemble will have a different set of hyperparameters (sampled from a given hyperparameter grid).

**Define Key Functions**

* 1. \_\_init\_\_(): Set up the base classifier, hyperparameter grid, and number of estimators.
  2. fit(X, y): Train multiple models with different hyperparameters.
  3. predict(X): Use majority voting to return class predictions.
  4. predict\_proba(X): Average probability scores across models (if the classifier supports probabilities).
  5. score(X, y): Use classification accuracy as an evaluation metric (inherited from ClassifierMixin).

**Design an Evaluation Experiment**

* 1. Benchmark your ensemble against **at least 4 standard classifiers**.
  2. Use **at least 5 datasets** for testing.
  3. Choose an appropriate performance metric (e.g., accuracy, F1-score, AUC).

**Link to study:**

### ****Scikit-Learn Documentation****

**Important info**

https://scikit-learn.org/stable/developers/develop.html

**Custom Estimators (BaseEstimator & ClassifierMixin)**

* + https://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator
  + Explains how to properly implement BaseEstimator and ClassifierMixin to create custom models.

**Voting Classifier (For Combining Predictions)**

* + https://scikit-learn.org/stable/modules/ensemble.html#voting-classifier
  + Since your classifier needs to combine multiple base models, this will help with the voting mechanism.

**Grid Search & Hyperparameter Tuning**

* + https://scikit-learn.org/stable/modules/grid\_search.html
  + Your classifier will train multiple models with different hyperparameter settings, so understanding how GridSearchCV and ParameterSampler work will be useful.

### ****2. Python Basics for Class Implementation****

* **Python Classes and Inheritance**
  + [https://docs.python.org/3/tutorial/classes.html](https://docs.python.org/3/tutorial/classes.html" \t "_new)
  + Since you need to define a class with methods (\_\_init\_\_, fit, predict, etc.), reviewing Python class structures is useful.

### ****3. NumPy for Matrix Operations (Useful for Aggregation in Voting)****

* **NumPy Arrays and Operations**
  + https://numpy.org/doc/stable/user/quickstart.html
  + You'll need NumPy for handling matrices when aggregating model predictions.

### ****4. Scikit-Learn Base Estimators (To Choose a Base Model)****

* **List of All Scikit-Learn Classifiers**
  + https://scikit-learn.org/stable/supervised\_learning.html
  + Helps you decide which base classifier (Decision Tree, SVM, k-NN, etc.) to use in your ensemble.

### ****5. Handling Model Evaluation & Metrics****

* **Classification Metrics (Accuracy, F1-Score, etc.)**
  + https://scikit-learn.org/stable/modules/model\_evaluation.html
  + Since your assignment requires benchmarking your ensemble against other models, understanding classification metrics is key.

**Solution**

**Used datasets:**

from sklearn.datasets import load\_iris, load\_breast\_cancer, load\_digits, load\_wine

# Example: Load the Iris dataset

data = load\_iris()

X, y = data.data, data.target

from sklearn.datasets import fetch\_openml

# Load the Soybean dataset

data = fetch\_openml(name="soybean\_small", version=1, as\_frame=False)

X, y = data.data, data.target

**Used baseEstimator KNN**

[1.6. Nearest Neighbors — scikit-learn 1.6.1 documentation](https://scikit-learn.org/stable/modules/neighbors.html" \l "nearest-neighbors-classification)

[KNeighborsClassifier — scikit-learn 1.6.1 documentation](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html)

· **Neighbors-based classification** doesn't learn rules, it just **remembers and looks around** when needed.

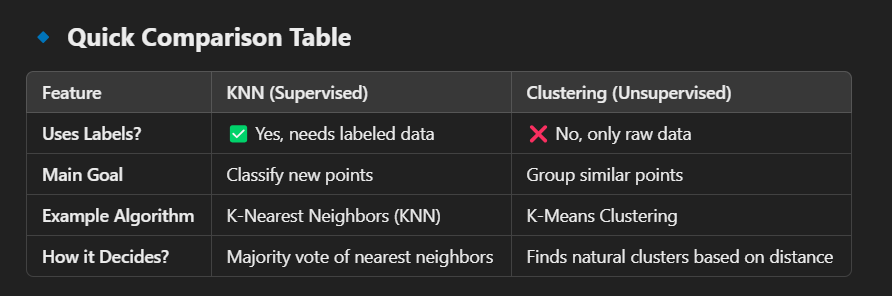
· **KNeighborsClassifier (KNN)** → Look at **k** nearest points and choose the most common class.

· **Closer points can be more important** (weighted voting).

PS. KNN != Clustering

· **KNN** helps with **classification** when we already have **labeled data**.

· **Clustering** finds **groups in raw data** when labels **aren’t available**



**Gameplan of solution:**

✅ **Use KNN as the base model**  
✅ **Train multiple KNNs with different values of** k  
✅ **Store all models and their predictions**  
✅ **Use majority voting for final classification**  
✅ **Use probability averaging for** predict\_proba()

Before writing \_\_init\_\_(), you should **properly document your class** by filling in the docstring sections (Parameters, Attributes, Notes, etc.).

### ****What to Do in** \_\_init\_\_()**

In the constructor (\_\_init\_\_), you should:

1. **Loop** from 0 to n\_estimators.
2. **Store** base\_estimator → This will be KNeighborsClassifier from Scikit-Learn.
3. **Store** param\_grid → This is a dictionary defining different values for n\_neighbors (e.g., {‘n\_neighbors’: [3, 5, 7, 10]}).
4. **Store** n\_estimators → This tells how many models to train (e.g., 4 models if {‘n\_neighbors’: [3, 5, 7, 10]}).
5. **Create an empty list** to store multiple trained KNN models.

### ****🔹 How It Works****

* You **do not** write KNN from scratch.
* Instead, you **call** KNeighborsClassifier **multiple times** with different n\_neighbors values.
* Store each trained model in a **list**.

### ****🔹 Key Takeaways****

✅ self.models is an **instance attribute**, meaning each instance of HyperParamClassifier has its own list.  
✅ **It gets initialized in** \_\_init\_\_(), and then fit() will **fill it with trained models**.  
✅ You **don’t need to declare it before calling** \_\_init\_\_—Scikit-Learn h

Unlike languages like **Java or C++**, Python **does not require declaring variables** before using them.  
Instead, **you can assign an attribute to an instance at any time**, and Python will create it automatically.

Example:

class Example:

def \_\_init\_\_(self):

self.value = 42 # No need to predefine "value"

def set\_extra\_attribute(self):

self.extra = "New Attribute" # This dynamically adds "extra" to the object

obj = Example()print(obj.value) # ✅ Works (defined in \_\_init\_\_)

obj.set\_extra\_attribute()print(obj.extra) # ✅ Works (created dynamically in method)

### ****How Does This Relate to Scikit-Learn?****

Scikit-Learn **follows Python’s dynamic nature**:

* You don’t have to **predefine attributes** before \_\_init\_\_().
* BaseEstimator **does not force predefined attributes**—it just provides get\_params() and set\_params(), which work with any dynamically created attributes.

### ****Why Does**** \_\_init\_\_() ****Have**** self****?****

✅ self **refers to the instance of the class** that is being created.

When you initialize an object from a class, Python **automatically passes** that object as the **first parameter** to every method inside the class.

* That’s why we **must include** self **as the first argument in** \_\_init\_\_().

def \_\_init\_\_(self, param\_grid, n\_estimators ):

self **is required because Python automatically passes the instance to class methods.**  
✅ **You initialize your class by passing** param\_grid **and** n\_estimators**.**  
✅ **After initialization, your model stores these values inside the instance**

#### ****Initialize**** HyperParamClassifier

my\_model = HyperParamClassifier(param\_grid=param\_grid, n\_estimators=5)

**Now, inside fit(), you will:**

### ****What is the**** fit() ****Function? (In Simple Terms)****

✅ **The** fit() **function is where the machine learning model "learns" from the training data (**X**,** y**).**  
✅ **It prepares the model so that it can later make predictions on new data.**

### ****🔹 What Does**** fit() ****Do?****

**Validates** X **and** y

* 1. Makes sure X is a **2D feature matrix** ((n\_samples, n\_features)).
  2. Makes sure y is a **1D target array** ((n\_samples,)).
  3. Ensures X and y have the **same number of rows**.

**Learns from the Data**

* 1. For models like **Decision Trees, Logistic Regression, or Neural Networks**, fit() **computes and stores patterns** (e.g., splits, weights, coefficients).
  2. For **instance-based models like KNN**, fit() **just stores** X **and** y, since predictions happen at query time.

**Stores Necessary Information**

* 1. **Some models store** X **and** y (like KNN, which needs them for distance calculations).
  2. **Some models don’t store** X **and** y, but instead store computed values (like regression coefficients in LinearRegression).
  3. Stores the **classes\_** (unique class labels) for classification tasks.

**Returns** self **(the trained model)**

* 1. This all for **chaining**:

### ****🔹 Example:**** fit() ****in a Simple Custom Model****

import numpy as np

class SimpleModel:

def \_\_init\_\_(self):

self.trained\_data = None # Placeholder to store X

self.labels = None # Placeholder to store y

def fit(self, X, y):

"""Learn from the data (just stores it for now)."""

# ✅ Validate input

if X.shape[0] != y.shape[0]:

raise ValueError("X and y must have the same number of samples.")

self.trained\_data = X # Stores the training data

self.labels = y # Stores the target labels

return self # ✅ Returns itself to allow chaining

def predict(self, X\_new):

"""Make predictions (dummy implementation)."""

return np.random.choice(self.labels, size=len(X\_new)) # Random predictions

# Example Usage

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

predictions = model.predict(X\_test)

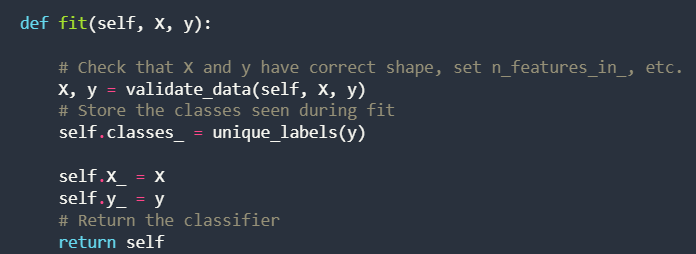
### ****🔹 How**** fit() ****Differs for Different Models****

| **Model Type** | **What** fit() **Does** |
| --- | --- |
| **Linear Regression** | Computes weights (w and b) to minimize error. |
| **Decision Tree** | Finds best splits and stores the tree structure. |
| **Neural Network** | Adjusts weights using backpropagation. |
| **KNN** | Just **stores** X **and** y—doesn’t "train" anything. |

**📌**

1. **Loop through** param\_grid **to get different hyperparameters.**
2. **Assign them to KNN models using** .set\_params()**.**
3. **Train and store the models.**

**I need to validate x,y then set the parameters in each knn, then fit the data in each of them (** For **instance-based models like KNN**, fit() **just stores** X **and** y, since predictions happen at query time.) and append them back to them models KNN

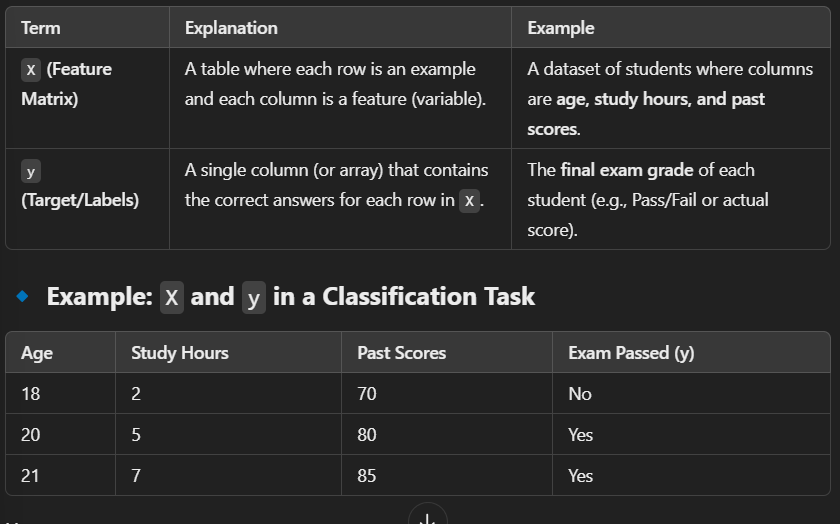
def fit(self, X, y):**......**  # Check that X and y have correct shape, set n\_features\_in\_, etc.**...**  X, y = validate\_data(self, X, y)**...**  # Store the classes seen during fit**...**  self.classes\_ = unique\_labels(y)**......**  self.X\_ = X**...**  self.y\_ = y**...**  # Return the classifier**...**  return self

### ****What Are**** X ****and**** y****, and Why Do We Validate Them?****

✅ X **(features) and** y **(labels/targets) are the training data used to train a machine learning model.**  
✅ **Validation ensures they have the correct format, preventing errors during training.**

### ****🔹 Breaking Down**** X ****and**** y ****in Simple Terms****

📌 Think of X **as the input data** (things the model uses to make predictions).  
📌 Think of y **as the correct answers** (what the model should learn to predict).



Here:

* X = [[18, 2, 70], [20, 5, 80], [21, 7, 85]]
* y = ["No", "Yes", "Yes"]

### ****🔹 Why Do We Validate**** X ****and**** y ****in**** fit()****?****

✅ **To prevent errors before training starts.**  
✅ **To make sure the model gets the correct input format.**  
✅ **To ensure that** X **and** y **match in sample size (**X.shape[0] == y.shape[0]**).**

### ****🔹 What Happens If We Don’t Validate?****

❌ If X is not a matrix → The model might crash.  
❌ If y is not a 1D array → The model won’t know what to predict.  
❌ If X and y have different numbers of rows → The model won’t work.

## ****🔹 What Happens in**** fit()****? (Explained Like You're 5)****

✅ fit() **is where the model "learns".**  
✅ It **looks at** X **(the data) and** y **(the answers)** to understand patterns.  
✅ **After** fit()**, the model is ready to make predictions.**

### ****🔹 How Does**** fit() ****Work?****

When you call:

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

This happens:

1. The model **checks** X\_train and y\_train to make sure they’re valid.
2. It **memorizes patterns** in X\_train that match y\_train.
3. **After learning, it’s ready to predict** new X\_test values.

### ****🔹 Key Rules for**** fit()

✅ **Must validate** X **and** y → Ensure correct input format.  
✅ **Must not store** X **and** y → Models **learn** from them, but don’t keep them.  
✅ **Must return** self → So that you can chain commands like:

python

CopiaModifica

y\_pred = MyModel().fit(X\_train, y\_train).predict(X\_test)

How to unpack dictionary \*\*

def greet(name, age):

print(f"Hello, my name is {name} and I am {age} years old.")

person = {"name": "Alice", "age": 25}

greet(\*\*person) # ✅ Equivalent to greet(name="Alice", age=25)

Predict:

### ****Explanation of Your**** predict() ****Code, Line by Line****

Let's break it down step by step.

python

CopiaModifica

from scipy.stats import modeimport numpy as np

🔹 **Imports required libraries**

* mode (from scipy.stats) → Finds the most common value for majority voting.
* numpy (np) → Handles array operations efficiently.

python

CopiaModifica

def predict(self, X):

🔹 **Defines the** predict() **method**

* self → Refers to the instance of HyperParamClassifier.
* X → The new input data (test samples) for which we want to predict class labels.

### ****1️⃣ Ensure the Model is Trained****

python

CopiaModifica

if not self.models:

raise ValueError("The model has not been trained. Call fit() first.")

🔹 **Checks if** fit() **has been called**

* If self.models is empty, it means no models have been trained.
* **Raises an error** if fit() was not called before predict().

### ****2️⃣ Validate**** X

python

CopiaModifica

if not isinstance(X, (np.ndarray, list)):

raise ValueError("X should be a NumPy array or a list of numerical values.")

🔹 **Ensures** X **is a valid input format**

* It must be either a NumPy array (np.ndarray) or a Python list (list).
* **Raises an error** if X is not in the correct format.

python

CopiaModifica

X = np.array(X) if isinstance(X, list) else X

🔹 **Converts** X **to a NumPy array if it's a list**

* This ensures **consistent data processing**.

python

CopiaModifica

if X.ndim != 2:

raise ValueError(f"X must be a 2D array (samples, features). Found {X.ndim}D instead.")

🔹 **Checks that** X **is a 2D array**

* ML models expect input in the form **(n\_samples, n\_features)**.
* If X is not 2D, it **raises an error**.

### ****3️⃣ Get Predictions from All Models****

python

CopiaModifica

predictions = np.array([model.predict(X) for model in self.models])

🔹 **Each trained model in** self.models **makes predictions on** X

* Calls .predict(X) on every individual model in the ensemble.
* Stores results in a NumPy array with shape **(n\_estimators, n\_samples)**.

**Example Output of** predictions

csharp

CopiaModifica

predictions = [

[0, 1, 1], # Model 1 predictions

[1, 1, 0], # Model 2 predictions

[0, 1, 0] # Model 3 predictions

]

Shape: **(n\_estimators, n\_samples)** → Each row is a model's predictions.

### ****4️⃣ Perform Majority Voting****

python

CopiaModifica

final\_predictions = mode(predictions, axis=0).mode[0]

🔹 **Finds the most common class per sample**

* axis=0 → Takes the mode **column-wise**, meaning it selects the most common prediction for each sample across models.
* .mode[0] → Extracts the most frequent class label per sample.

**Example Majority Vote**

mathematica

CopiaModifica

Predictions from 3 models:

Model 1 → [0, 1, 1]

Model 2 → [1, 1, 0]

Model 3 → [0, 1, 0]

Majority vote per sample:

Sample 1 → (0, 1, 0) → Majority = 0

Sample 2 → (1, 1, 1) → Majority = 1

Sample 3 → (1, 0, 0) → Majority = 0

Final Output: [0, 1, 0]

This ensures that **each test sample gets a class label based on the majority vote.**

### ****5️⃣ Return the Final Predicted Labels****

python

CopiaModifica

return final\_predictions

🔹 **Returns the array of majority-voted class labels**

* Output shape: **(n\_samples,)**
* This is the final classification result for all input samples.

## ****✅ Steps to Implement**** predict\_proba()

1️⃣ **Ensure the model is trained** (self.models must not be empty).  
2️⃣ **Validate** X (ensure correct format, convert to NumPy).  
3️⃣ **Get class probability predictions from each trained model** (model.predict\_proba(X)).  
4️⃣ **Average the probabilities across all models** to compute final probabilities.  
5️⃣ **Return probability distributions for each test sample**.

## ****Example Calculation****

Imagine **3 models predict probabilities for 2 classes** (n\_samples=1):

css

CopiaModifica

Model 1 → [0.8, 0.2]

Model 2 → [0.6, 0.4]

Model 3 → [0.9, 0.1]

Averaging these probabilities:

css

CopiaModifica

Final Probabilities → [(0.8 + 0.6 + 0.9) / 3, (0.2 + 0.4 + 0.1) / 3]

→ [0.77, 0.23]

So, predict\_proba() returns:

python

CopiaModifica

[[0.77, 0.23]] # 77% probability for class 0, 23% for class 1

## ****📌 Final Takeaway****

🚀 **Your** predict\_proba() **now correctly computes soft voting probabilities.**  
🚀 **It follows your assignment's guidelines—no pre-built ensembles!**  
🚀 **Now your ensemble supports both majority voting (**predict()**) and probability-based voting (**predict\_proba()**).**

Would you like help with **testing and evaluating** your ensemble now? 🚀

### ****Difference Between Majority Voting (****predict()****) and Soft Voting (****predict\_proba()****)****

Both methods combine predictions from multiple models in an ensemble, but **they work differently**.

## ****1️⃣ Majority Voting (****predict()****) – Hard Voting****

🔹 **Each model makes a prediction (**predict()**).**  
🔹 The ensemble **selects the most frequent class** (majority vote).  
🔹 **Final output:** The most common class label.

### ****🔹 Example: 3 Models Predict for 5 Samples****

| **Model** | **Sample 1** | **Sample 2** | **Sample 3** | **Sample 4** | **Sample 5** |
| --- | --- | --- | --- | --- | --- |
| Model 1 | 0 | 1 | 1 | 0 | 1 |
| Model 2 | 1 | 1 | 0 | 0 | 1 |
| Model 3 | 0 | 1 | 0 | 1 | 1 |
| **Majority Vote (Final Output)** | 0 | 1 | 0 | 0 | 1 |

🔹 The **most common class per sample** is selected.

## ****2️⃣ Soft Voting (****predict\_proba()****) – Weighted Probability Voting****

🔹 **Each model predicts class probabilities (**predict\_proba()**).**  
🔹 The ensemble **takes the average of all predicted probabilities** for each class.  
🔹 **Final output:** A probability distribution over all classes.

### ****🔹 Example: 3 Models Predict Probabilities****

#### ****Predicted Probabilities from 3 Models****

| **Sample** | **Model** | **Class 0 Prob** | **Class 1 Prob** |
| --- | --- | --- | --- |
| 1 | Model 1 | 0.8 | 0.2 |
| 1 | Model 2 | 0.6 | 0.4 |
| 1 | Model 3 | 0.9 | 0.1 |
| **Final Prob (Mean)** | - | **0.77** | **0.23** |

🔹 The final probability is the **mean of all model probabilities**:

P(class 0)=0.8+0.6+0.93=0.77P(\text{class 0}) = \frac{0.8 + 0.6 + 0.9}{3} = 0.77P(class 0)=30.8+0.6+0.9​=0.77 P(class 1)=0.2+0.4+0.13=0.23P(\text{class 1}) = \frac{0.2 + 0.4 + 0.1}{3} = 0.23P(class 1)=30.2+0.4+0.1​=0.23

## ****📌 Key Differences****

| **Feature** | **Majority Voting (predict())** | **Soft Voting (predict\_proba())** |
| --- | --- | --- |
| **Based On** | Class labels (predict()) | Probability distributions (predict\_proba()) |
| **Aggregation** | Picks most common class (mode) | Averages class probabilities |
| **Output** | Hard class labels (0, 1, 2, ...) | Soft probabilities ([0.77, 0.23]) |
| **Better for** | General classification | Probabilistic decisions (confidence scores) |

**Task 2-3**

**Benchmark measure : accuracy**

**Algorithm to go against**

**Decision tree**

**Naive bayes**

**Random Forest**

**Single KNN**

**Datasets:**

**breast\_cancer, digits, wine**

**Iris, forest cover type**

**All numerical ( didnt want to convert categorical lol)**

### ****Is Your Code Actually Testing the Models?****

✅ Your current code **trains models** and **evaluates them using cross-validation**.  
❌ However, **it does NOT test models on a separate test set** after training.

Since your assignment asks for **performance evaluation**, you should ensure your code **trains, validates, AND tests** the models.

### ****📌 What You Need to Fix****

1️⃣ **Split Data into Train & Test** – Right now, you use the whole dataset for training.  
2️⃣ **Test on Unseen Data** – After training, evaluate on a separate test set.  
3️⃣ **Ensure Cross-Validation is Used** – If you’re already using cross\_val\_score(), that’s good for training evaluation, but you still need to test the final model separately.  
4️⃣ **Select a Performance Metric** – Choose accuracy, F1-score, or AUC based on the problem.

**Maybe ASSIGNMENT 2 IS TOTALLY WRONG**

**For now, lets downsample forest cover type, -> DONE**

**Split datasets -> DONE**

**Finish checking rankings ->DONE**