EXPERIMENT 7

BAGGING

In [2]:
```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load data
data = load_iris()
X = data.data
y = data.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Bagging implementation
class Bagging:
    def __init__(self, base_model, n_models=10):
        self.base_model = base_model
        self.n_models = n_models
        self.models = []

    def fit(self, X, y):
        for i in range(self.n_models):
            # Bootstrap sampling
            indices = np.random.choice(len(X), len(X), replace=True)
            X_sample, y_sample = X[indices], y[indices]
            model = self.base_model()
            model.fit(X_sample, y_sample)
            self.models.append(model)
            print(f"Model {i+1} trained with accuracy: {accuracy_score(y_sample, model.predict(X_sample))}")

    def predict(self, X):
        # Aggregate predictions from each model
        predictions = np.zeros((X.shape[0], len(self.models)))
        for i, model in enumerate(self.models):
            predictions[:, i] = model.predict(X)
        # Majority vote
        return np.round(np.mean(predictions, axis=1))

# Initialize and train the bagging model
bagging_model = Bagging(base_model=DecisionTreeClassifier, n_models=10)
bagging_model.fit(X_train, y_train)

# Make predictions and evaluate
y_pred = bagging_model.predict(X_test)
print(f"Bagging Model Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Model 1 trained with accuracy: 1.0
Model 2 trained with accuracy: 1.0
Model 3 trained with accuracy: 1.0
Model 4 trained with accuracy: 1.0
Model 5 trained with accuracy: 1.0
Model 6 trained with accuracy: 1.0
Model 7 trained with accuracy: 1.0
Model 8 trained with accuracy: 1.0
Model 9 trained with accuracy: 1.0
Model 10 trained with accuracy: 1.0
Bagging Model Accuracy: 1.0
```

BOOSTING

In [5]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.utils import shuffle

class AdaBoost:
    def __init__(self, base_model, n_models=50):
        self.base_model = base_model
        self.n_models = n_models
        self.models = []
        self.alphas = []
```

```
    def fit(self, X, y):
        n_samples = X.shape[0]
        weights = np.ones(n_samples) / n_samples
        for i in range(self.n_models):
            model = self.base_model()
            model.fit(X, y, sample_weight=weights)
            y_pred = model.predict(X)
            # Calculate error
            error = np.sum(weights * (y_pred != y)) / np.sum(weights)

            # Handle zero error
            if error == 0:
                alpha = 1.0
            else:
                alpha = 0.5 * np.log((1 - error) / error)

            # Update weights
            weights *= np.exp(alpha * (y_pred != y))
            weights /= np.sum(weights)

            # Check for NaN in weights
            if np.any(np.isnan(weights)):
                raise ValueError("Weights contain NaN values")

            self.models.append(model)
            self.alphas.append(alpha)
            print(f"Model {i+1}: Error = {error}, Alpha = {alpha}")

    def predict(self, X):
        predictions = np.zeros(X.shape[0])
        for model, alpha in zip(self.models, self.alphas):
            predictions += alpha * model.predict(X)
        return np.sign(predictions)

# Initialize and train the AdaBoost model
adaboost_model = AdaBoost(base_model=DecisionTreeClassifier, n_models=50)
adaboost_model.fit(X_train, y_train)

# Make predictions and evaluate
y_pred = adaboost_model.predict(X_test)
print(f"AdaBoost Model Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Model 1: Error = 0.0, Alpha = 1.0
Model 2: Error = 0.0, Alpha = 1.0
Model 3: Error = 0.0, Alpha = 1.0
Model 4: Error = 0.0, Alpha = 1.0
Model 5: Error = 0.0, Alpha = 1.0
Model 6: Error = 0.0, Alpha = 1.0
Model 7: Error = 0.0, Alpha = 1.0
Model 8: Error = 0.0, Alpha = 1.0
Model 9: Error = 0.0, Alpha = 1.0
Model 10: Error = 0.0, Alpha = 1.0
Model 11: Error = 0.0, Alpha = 1.0
Model 12: Error = 0.0, Alpha = 1.0
Model 13: Error = 0.0, Alpha = 1.0
Model 14: Error = 0.0, Alpha = 1.0
Model 15: Error = 0.0, Alpha = 1.0
Model 16: Error = 0.0, Alpha = 1.0
Model 17: Error = 0.0, Alpha = 1.0
Model 18: Error = 0.0, Alpha = 1.0
Model 19: Error = 0.0, Alpha = 1.0
Model 20: Error = 0.0, Alpha = 1.0
Model 21: Error = 0.0, Alpha = 1.0
Model 22: Error = 0.0, Alpha = 1.0
Model 23: Error = 0.0, Alpha = 1.0
Model 24: Error = 0.0, Alpha = 1.0
Model 25: Error = 0.0, Alpha = 1.0
Model 26: Error = 0.0, Alpha = 1.0
Model 27: Error = 0.0, Alpha = 1.0
Model 28: Error = 0.0, Alpha = 1.0
Model 29: Error = 0.0, Alpha = 1.0
Model 30: Error = 0.0, Alpha = 1.0
Model 31: Error = 0.0, Alpha = 1.0
Model 32: Error = 0.0, Alpha = 1.0
Model 33: Error = 0.0, Alpha = 1.0
Model 34: Error = 0.0, Alpha = 1.0
Model 35: Error = 0.0, Alpha = 1.0
Model 36: Error = 0.0, Alpha = 1.0
Model 37: Error = 0.0, Alpha = 1.0
Model 38: Error = 0.0, Alpha = 1.0
Model 39: Error = 0.0, Alpha = 1.0
Model 40: Error = 0.0, Alpha = 1.0
Model 41: Error = 0.0, Alpha = 1.0
Model 42: Error = 0.0, Alpha = 1.0
Model 43: Error = 0.0, Alpha = 1.0
Model 44: Error = 0.0, Alpha = 1.0
Model 45: Error = 0.0, Alpha = 1.0
Model 46: Error = 0.0, Alpha = 1.0
Model 47: Error = 0.0, Alpha = 1.0
Model 48: Error = 0.0, Alpha = 1.0
Model 49: Error = 0.0, Alpha = 1.0
Model 50: Error = 0.0, Alpha = 1.0
AdaBoost Model Accuracy: 0.7111111111111111
```

STACKING

In [4]:

```python
from sklearn.linear_model import LogisticRegression

class Stacking:
    def __init__(self, base_models, meta_model):
        self.base_models = base_models
        self.meta_model = meta_model
        self.meta_X_train = None

    def fit(self, X, y):
        meta_X = np.zeros((X.shape[0], len(self.base_models)))
        for i, model in enumerate(self.base_models):
            model.fit(X, y)
            meta_X[:, i] = model.predict(X)
            print(f"Base Model {i+1} predictions: {meta_X[:, i]}")
        self.meta_model.fit(meta_X, y)

    def predict(self, X):
        meta_X = np.zeros((X.shape[0], len(self.base_models)))
        for i, model in enumerate(self.base_models):
            meta_X[:, i] = model.predict(X)
        return self.meta_model.predict(meta_X)

# Initialize base models and meta-model
base_models = [DecisionTreeClassifier(), DecisionTreeClassifier(max_depth=3)]
meta_model = LogisticRegression()

# Initialize and train the stacking model
```

```
stacking_model = Stacking(base_models=base_models, meta_model=meta_model)
stacking_model.fit(X_train, y_train)

# Make predictions and evaluate
y_pred = stacking_model.predict(X_test)
print(f"Stacking Model Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Base Model 1 predictions: [1. 2. 2. 1. 2. 1. 2. 1. 0. 2. 1. 0. 0. 0. 1. 2. 0. 0. 0. 1. 0. 1. 2. 0.
 1. 2. 0. 2. 2. 1. 1. 2. 1. 0. 1. 2. 0. 0. 1. 1. 0. 2. 0. 0. 1. 1. 2. 1.
 2. 2. 1. 0. 0. 2. 2. 0. 0. 0. 1. 2. 0. 2. 2. 0. 1. 1. 2. 1. 2. 0. 2. 1.
 2. 1. 1. 1. 0. 1. 1. 0. 1. 2. 2. 0. 1. 2. 2. 0. 2. 0. 1. 2. 2. 1. 2. 1.
 1. 2. 2. 0. 1. 2. 0. 1. 2.]
Base Model 2 predictions: [1. 1. 2. 1. 2. 1. 2. 1. 0. 2. 1. 0. 0. 0. 1. 2. 0. 0. 0. 1. 0. 1. 2. 0.
 1. 2. 0. 2. 2. 1. 1. 2. 1. 0. 1. 2. 0. 0. 1. 1. 0. 2. 0. 0. 2. 1. 2. 1.
 1. 2. 1. 0. 0. 1. 2. 0. 0. 0. 1. 2. 0. 2. 2. 0. 1. 1. 2. 1. 2. 0. 2. 1.
 2. 1. 1. 1. 0. 1. 1. 0. 1. 2. 2. 0. 1. 2. 1. 0. 2. 0. 1. 2. 2. 1. 2. 1.
 1. 2. 2. 0. 1. 2. 0. 1. 2.]
Stacking Model Accuracy: 1.0
```

In [ ]: