😊 Ensemble learning

Ensemble learning in machine learning is like combining the wisdom of a group rather than relying on a single decision. Instead of using just one model to make predictions, ensemble learning combines multiple models to improve accuracy, stability, and robustness. It's similar to how a group of experts can provide better solutions than just one expert.

Example in Layman's Terms:

Imagine you're trying to guess how many jelly beans are in a jar. If you ask one person, their guess might be off. But if you ask a large group of people and take the **average** of their guesses, the result will likely be more accurate. Ensemble learning works the same way—combining the results of different models to get better predictions.

Types of Ensemble Learning:

1. **Bagging (Bootstrap Aggregating)**:
    - **How it works**: Multiple models (like decision trees) are trained on different random samples of the data, and their predictions are averaged.
    - **Example**: Random Forest is a popular bagging algorithm. It creates several decision trees and takes a "vote" from each tree to make the final prediction.
2. **Boosting**:
    - **How it works**: Models are trained one after another, and each new model tries to correct the mistakes made by the previous model. The final prediction is a weighted sum of all models.
    - **Example**: AdaBoost and Gradient Boosting are boosting algorithms. They focus more on the errors made by earlier models to improve accuracy over time.
3. **Stacking**:
    - **How it works**: Different models (like decision trees, support vector machines, etc.) are trained independently, and their outputs are combined by another model (called a meta-model) to make the final prediction.
    - **Example**: You might use logistic regression, a decision tree, and a neural network for the same problem. Then, another model (meta-learner) takes their predictions and makes the final decision.

Use Cases of Ensemble Learning:

1. **Finance**:
    - Used to predict stock prices or detect fraudulent transactions. Combining multiple models reduces the chances of errors in predictions.
2. **Healthcare**:

- In medical diagnosis, ensemble learning can help improve the accuracy of predicting diseases based on patient data (like detecting cancer from medical scans).

3. **Marketing**:
    - For customer segmentation and predicting customer behavior, ensemble models help classify customers into different groups based on behavior or preferences.
4. **Image Recognition**:
    - In tasks like face recognition or object detection, ensemble learning can improve the accuracy and precision of the results.

Benefits of Ensemble Learning:

1. **Improved Accuracy**:
    - By combining multiple models, ensemble methods usually perform better than individual models.
2. **Reduced Overfitting**:
    - Ensemble learning reduces the risk of overfitting, where a model performs well on training data but poorly on new, unseen data. Bagging techniques like Random Forest help mitigate this.
3. **Better Generalization**:
    - Ensemble models generalize better across different datasets because they aggregate the strengths of multiple models.
4. **Flexibility**:
    - Different models can be combined, allowing for flexibility. For example, you can combine a neural network, a decision tree, and a logistic regression model in an ensemble.

Disadvantages of Ensemble Learning:

1. **Increased Complexity**:
    - Combining multiple models can be computationally expensive and harder to interpret. It's no longer a single simple model but a combination of many.
2. **Longer Training Time**:
    - Since multiple models are being trained and combined, ensemble methods can take significantly longer to train compared to a single model.
3. **Less Interpretability**:
    - If you need to explain why a model made a certain decision, ensemble models can be more challenging to interpret compared to simpler models like decision trees or linear regression.

4. **Resource Intensive**:
    - Requires more computational power and memory, especially with large datasets or complex models.

Types of Ensemble Learning (Recap):

1. **Bagging** (e.g., Random Forest)
2. **Boosting** (e.g., AdaBoost, Gradient Boosting)
3. **Stacking** (using multiple models with a meta-model)
4. **Voting and Averaging** (simple majority or average of model predictions)

Summary:

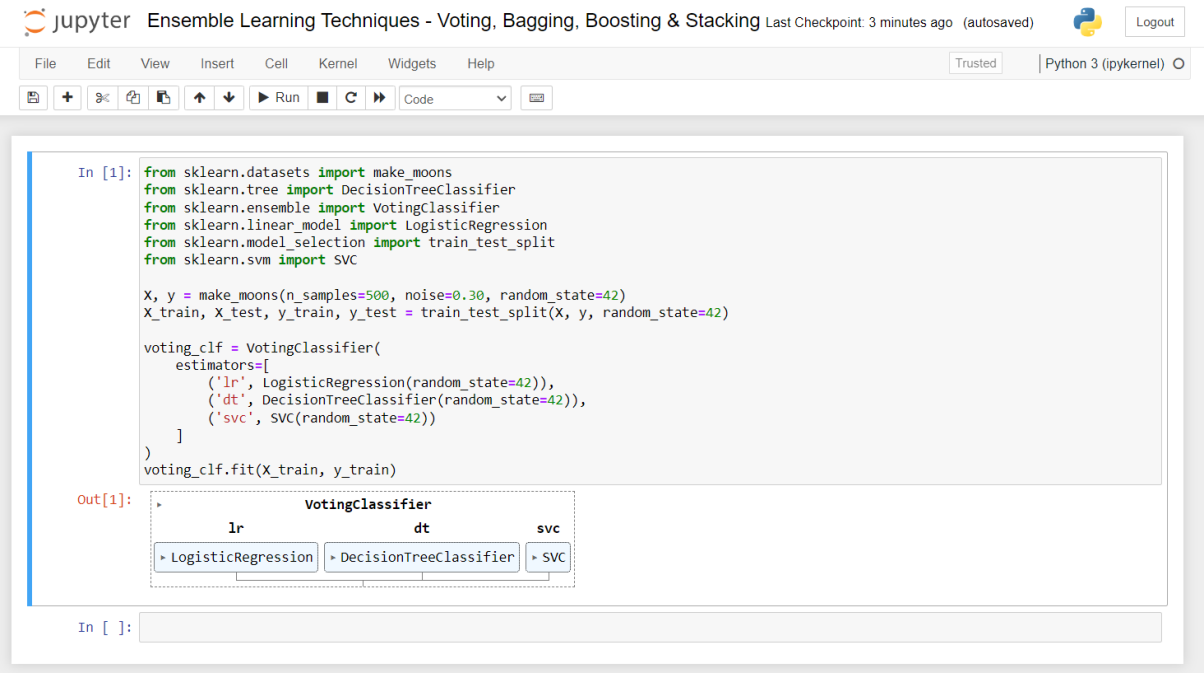
Ensemble learning improves model performance by combining the predictions of multiple models. It's widely used across industries for tasks like fraud detection, medical diagnosis, and stock market prediction. While it improves accuracy and reduces overfitting, it also comes with increased complexity and longer training times.

😄 To begin with the Lab:

1. **Generating a Synthetic Dataset**:
    - You are creating a synthetic dataset using the make_moons() function. This dataset has 500 samples with some **noise** added, simulating a non-linear pattern (like two interlocking moons), which is often used to test classification algorithms.
2. **Splitting the Data**:
    - You split the dataset into **training** and **test** sets. The training set is used to train the model, and the test set is used later to evaluate its performance. By default, 75% of the data is used for training, and 25% is used for testing.
3. **Creating a Voting Classifier**:
    - You are using a **VotingClassifier**, which combines predictions from multiple models (also called estimators) to improve accuracy. In this case, you are combining three different classifiers:
    - **Logistic Regression** (lr)
    - **Decision Tree Classifier** (dt)
    - **Support Vector Classifier (SVC)** (svc)
4. **Fitting the Voting Classifier**:
    - The voting classifier is trained on the **training data** (X_train, y_train). Each model will make predictions, and the **VotingClassifier** will combine these predictions. It can either use **hard voting** (majority vote) or **soft voting** (average probabilities).

    In short, this code combines three different machine learning models (logistic regression, decision tree, and support vector machine) into a **voting classifier**. This classifier is trained on a dataset of two interlocking moon shapes, and the goal is to use the combined strengths of these models to make better predictions.

5. In this step, you are fitting the **VotingClassifier** to the training data, which means you are training the combined model using the specified individual estimators (Logistic Regression, Decision Tree, and SVC). After fitting the model, if you want to evaluate the performance of each individual estimator within the voting classifier, you can check their **overall accuracy** on the training data or test data. This helps you understand how well each model is contributing to the ensemble's predictions.



6. By evaluating the individual models within the **VotingClassifier**, you can determine how well each model performs on the given dataset. Here are the accuracies you mentioned:
    - **Logistic Regression**: 86.4% accuracy
    - **Decision Tree**: 85.6% accuracy
    - **Support Vector Classifier (SVC)**: 89.6% accuracy
7. This information helps you understand the strengths and weaknesses of each model:
    - The **Support Vector Classifier** has the highest accuracy (89.6%), indicating it performs best on this dataset.
    - The **Logistic Regression** model follows closely with 86.4%.
    - The **Decision Tree** model has slightly lower accuracy at 85.6%.

    Overall, these performance metrics allow you to assess which model might contribute more effectively to the ensemble and make informed decisions about model selection or adjustments.

```
In [2]: for name,clf in voting_clf.named_estimators_.items():
            print(f"{name}= {clf.score(X_test,y_test)}")
```

```
lr= 0.864
dt= 0.856
svc= 0.896
```

8. By evaluating the **VotingClassifier** on the test data (X_test, y_test), you found that its overall accuracy is **90.4%**. This demonstrates that the voting classifier effectively combines the strengths of the individual models, leading to improved performance compared to each model alone.

```
In [3]: voting_clf.score(X_test,y_test)
```

```
Out[3]: 0.904
```

9. Below we have set up a Voting Classifier using Logistic Regression, Decision Tree, and Support Vector Classifier (SVC) **with soft voting**, which utilizes predicted probabilities for final decisions. We enabled probability estimates for the SVC model to allow it to contribute to the soft voting. After that, we trained the Voting Classifier on the training data. Finally, we evaluated its accuracy on the test data to see how well the ensemble performed on unseen samples.

```
In [4]: voting_clf = VotingClassifier(
            estimators=[
                ('lr', LogisticRegression(random_state=42)),
                ('dt', DecisionTreeClassifier(random_state=42)),
                ('svc', SVC(random_state=42))
            ],
            voting='soft'
        )
```

```
In [5]: voting_clf.named_estimators['svc'].probability=True
        voting_clf.fit(X_train, y_train)
        voting_clf.score(X_test,y_test)
```

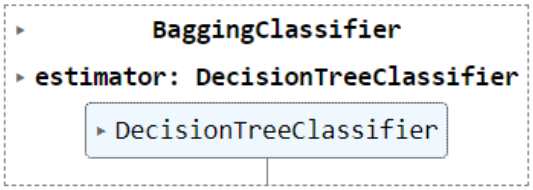
```
Out[5]: 0.912
```

10. Here you created a **Bagging Classifier** that uses a **Decision Tree** as the base model. You specified that the ensemble will consist of **500 decision trees**, with each tree trained on a random sample of **100 data points** from the training set. The n_jobs=-1 parameter allows the model to utilize all available processor cores for faster training. Finally, you trained the bagging classifier on the training data.

Bagging

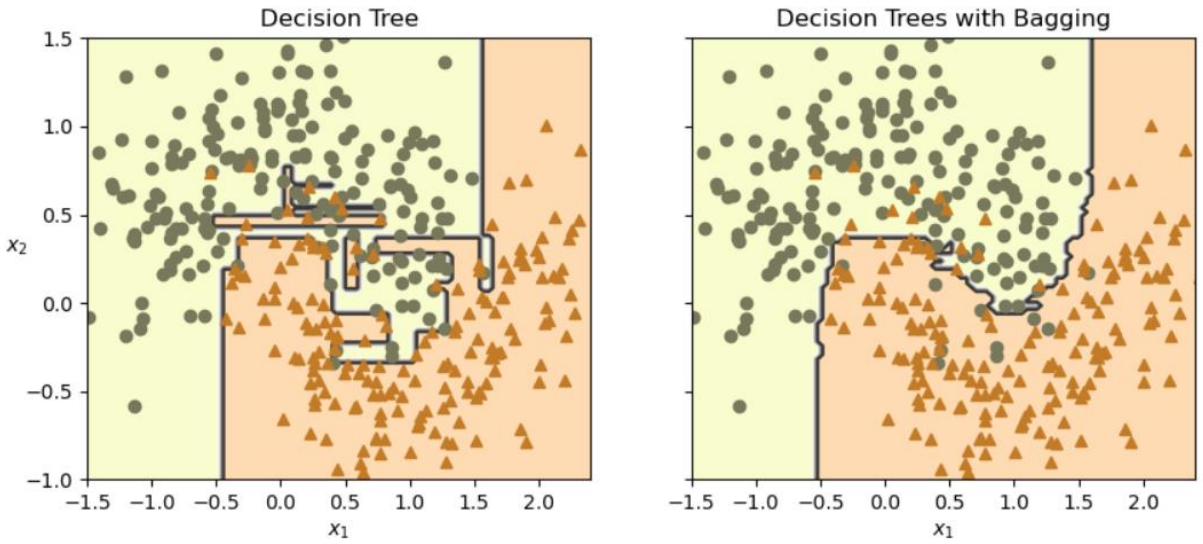
In [6]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

Out[6]:



11. Then you imported the necessary libraries for plotting and numerical operations. Then, you defined a function called `plot_decision_boundary` that visualizes the decision boundaries of a classifier on a given dataset. This function creates a grid of points, predicts the class for each point using the classifier, and then plots the decision regions along with the training data points.
12. Next, you trained a **Decision Tree Classifier** on the training data. You created a figure with two subplots to compare the decision boundaries of the trained decision tree and the bagging classifier. The first subplot shows the decision boundary of the standard decision tree, while the second subplot displays the decision boundary of the bagging classifier. Finally, you displayed the plots to visualize how each model classifies the data.

In [7]:
```python
import matplotlib.pyplot as plt
import numpy as np

def plot_decision_boundary(clf, X, y, alpha=1.0):
    axes=[-1.5, 2.4, -1, 1.5]
    x1, x2 = np.meshgrid(np.linspace(axes[0], axes[1], 100),
                         np.linspace(axes[2], axes[3], 100))
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)

    plt.contourf(x1, x2, y_pred, alpha=0.3 * alpha, cmap='Wistia')
    plt.contour(x1, x2, y_pred, cmap="Greys", alpha=0.8 * alpha)
    colors = ["#78785c", "#c47b27"]
    markers = ("o", "^")
    for idx in (0, 1):
        plt.plot(X[:, 0][y == idx], X[:, 1][y == idx],
                 color=colors[idx], marker=markers[idx], linestyle="none")
    plt.axis(axes)
    plt.xlabel(r"$x_1$")
    plt.ylabel(r"$x_2$", rotation=0)

tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(axes[0])
plot_decision_boundary(tree_clf, X_train, y_train)
plt.title("Decision Tree")
plt.sca(axes[1])
plot_decision_boundary(bag_clf, X_train, y_train)
plt.title("Decision Trees with Bagging")
plt.ylabel("")
plt.show()
```



13. You created a **Bagging Classifier** using a **Decision Tree** as the base model, specifying **500 estimators** to be trained. You enabled out-of-bag (OOB) scoring by setting oob_score=True, which allows you to evaluate the model's performance using samples that were not included in the training of each estimator. The n_jobs=-1 parameter allows the model to use all available processor cores for faster training. After training the

bagging classifier on the training data, you accessed the OOB score to assess its performance.

```
In [8]: bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                                   oob_score=True, n_jobs=-1, random_state=42)

        bag_clf.fit(X_train, y_train)
        bag_clf.oob_score_
```

```
Out[8]: 0.896
```

14. You used the trained **Bagging Classifier** to make predictions on the test data (X_test). After obtaining the predicted labels, you calculated the accuracy of these predictions by comparing them to the actual labels in the test set (y_test). The result gives you the model's accuracy score, indicating how well the bagging classifier performed on the unseen data.

```
In [9]: from sklearn.metrics import accuracy_score

        y_pred = bag_clf.predict(X_test)
        accuracy_score(y_test, y_pred)
```

```
Out[9]: 0.92
```

15. When you use the bagging technique with multiple decision trees, it becomes a **Random Forest**.
    - Random Forest is like a group of friends (the decision trees) who are each trying to make a decision (like guessing the outcome of a game or a class label). Instead of relying on just one friend's opinion, you ask several friends for their guesses. Each friend has their own way of thinking (because they are individual decision trees), and they all see the situation from slightly different angles.

16. You created a **Random Forest Classifier** with **500 trees** and a limit of **16 leaf nodes** for each tree to control complexity. After training this model on the training data, you used it to make predictions on the test data.
17. Next, you set up a **Bagging Classifier** that uses a decision tree with the same constraints on the number of features and leaf nodes, also consisting of **500 trees**. After training the bagging classifier on the training data, you made predictions on the test data as well.
18. Finally, you checked if the predictions from the bagging classifier were the same as those from the Random Forest classifier, confirming whether they produced identical results.

Random Forest

In [10]:
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

In [11]:
```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

In [12]:
```
bag_clf.fit(X_train, y_train)
y_pred_bag = bag_clf.predict(X_test)
np.all(y_pred_bag == y_pred_rf)
```

Out[12]: True

19. You imported the Iris dataset using the load_iris function and stored it in a variable called iris. Then, you created a **Random Forest Classifier** with **500 trees** and trained it on the Iris data features and target labels. After training, you looped through the feature importances, which indicate how much each feature contributes to the model's predictions, and printed each feature's importance score rounded to two decimal places along with the feature name. This gives insight into which features are most influential in classifying the different species of iris flowers.

Feature Importance

In [13]:
```
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris.data, iris.target)
for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

```
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

20. The next type of ensemble learning we are going to explore is boosting. In boosting, we train the models in a sequence. This means that, in this case, I will be using multiple machine learning algorithms, but each individual model will be trained sequentially.

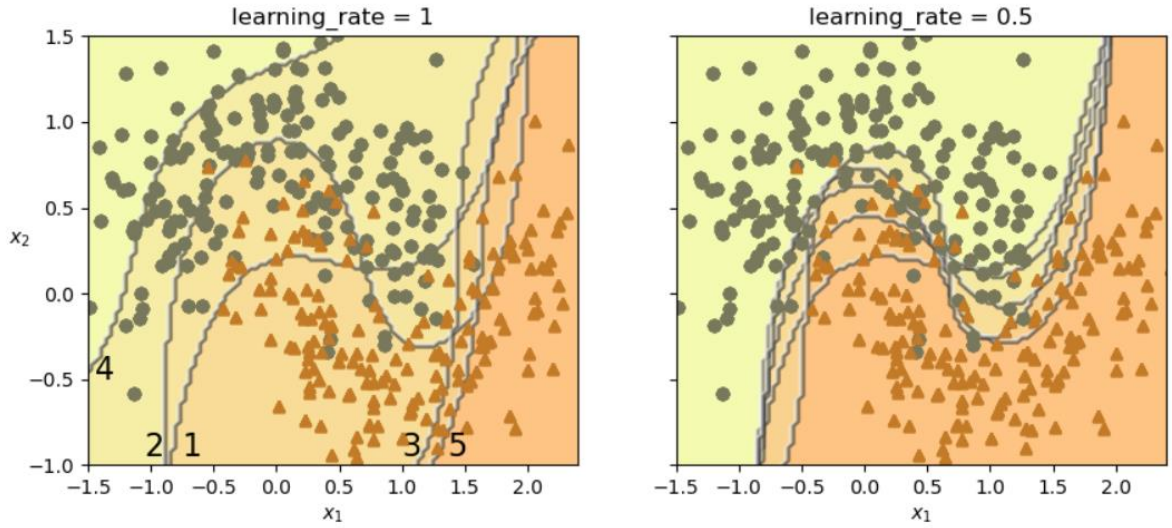
21. In the below code, you are implementing **AdaBoost** using Support Vector Classifiers (SVC) on a training dataset. You start by creating a plot with two subplots to visualize the decision boundaries of the classifiers for different learning rates. For each learning rate, you initialize sample weights equally among all training samples.

22. Then, you loop through five iterations, where in each iteration, you fit an SVC model to the training data using the current sample weights. After making predictions, you calculate the total error weight for misclassified samples and update the sample weights, giving more importance to those misclassified instances. This process helps the model focus on the harder-to-classify samples in subsequent iterations.
23. Finally, you plot the decision boundaries of the SVC for each iteration in the corresponding subplot, allowing you to visualize how the decision boundary evolves with each model trained. The overall result demonstrates how AdaBoost combines multiple weak classifiers to improve model performance.

Boosting

Adaboost

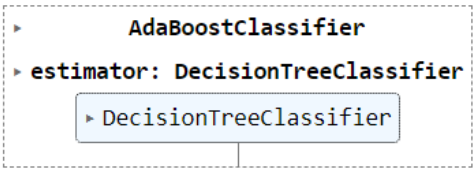
In [14]:
```python
#Adaboost
m = len(X_train)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
for subplot, learning_rate in ((0, 1), (1, 0.5)):
    sample_weights = np.ones(m) / m
    plt.sca(axes[subplot])
    for i in range(5):
        svm_clf = SVC(C=0.2, gamma=0.6, random_state=42)
        svm_clf.fit(X_train, y_train, sample_weight=sample_weights * m)
        y_pred = svm_clf.predict(X_train)

        error_weights = sample_weights[y_pred != y_train].sum()
        r = error_weights / sample_weights.sum()  # equation 7-1
        alpha = learning_rate * np.log((1 - r) / r)  # equation 7-2
        sample_weights[y_pred != y_train] *= np.exp(alpha)  # equation 7-3
        sample_weights /= sample_weights.sum()  # normalization step

        plot_decision_boundary(svm_clf, X_train, y_train, alpha=0.4)
        plt.title(f"learning_rate = {learning_rate}")
    if subplot == 0:
        plt.text(-0.75, -0.95, "1", fontsize=16)
        plt.text(-1.05, -0.95, "2", fontsize=16)
        plt.text(1.0, -0.95, "3", fontsize=16)
        plt.text(-1.45, -0.5, "4", fontsize=16)
        plt.text(1.36,  -0.95, "5", fontsize=16)
    else:
        plt.ylabel("")


plt.show()
```



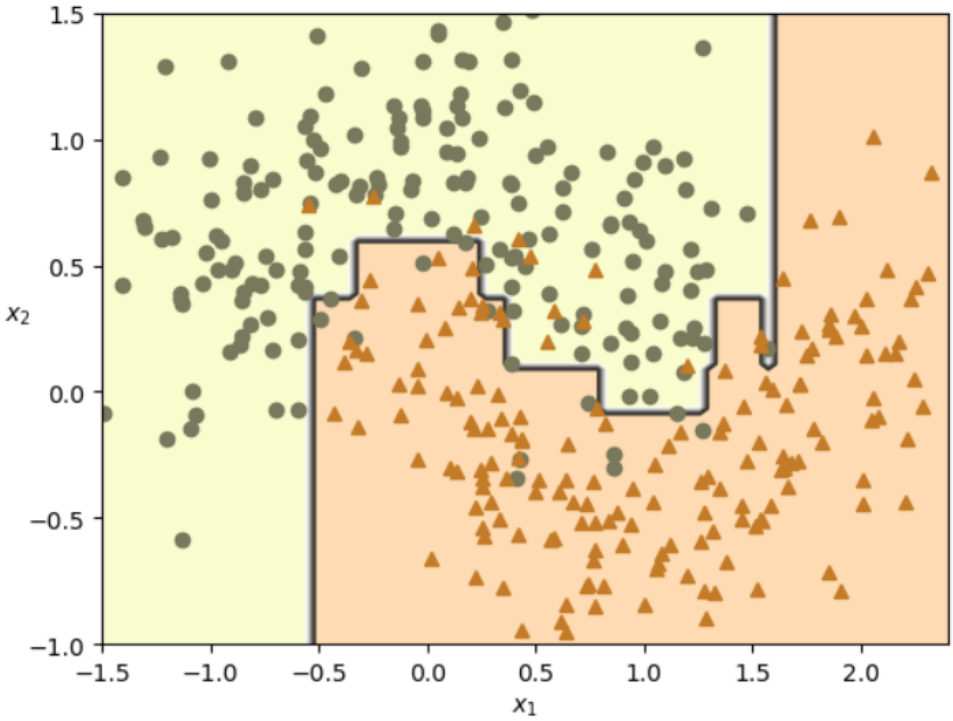
24. You are creating an **AdaBoost classifier** that uses a simple decision tree with a maximum depth of 1 as the base model. This classifier will consist of **30 estimators**, meaning it will build and combine 30 decision trees to improve accuracy. You set the learning rate to **0.5**, which controls how much each tree contributes to the final model.
25. After configuring the AdaBoost classifier, you fit it to the training data to learn from it. Finally, you visualize the decision boundary created by the AdaBoost classifier on the training data, allowing you to see how well the model distinguishes between different classes based on the features provided.

In [15]:
```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

Out[15]:



In [16]:
```
plot_decision_boundary(ada_clf, X_train, y_train)
```



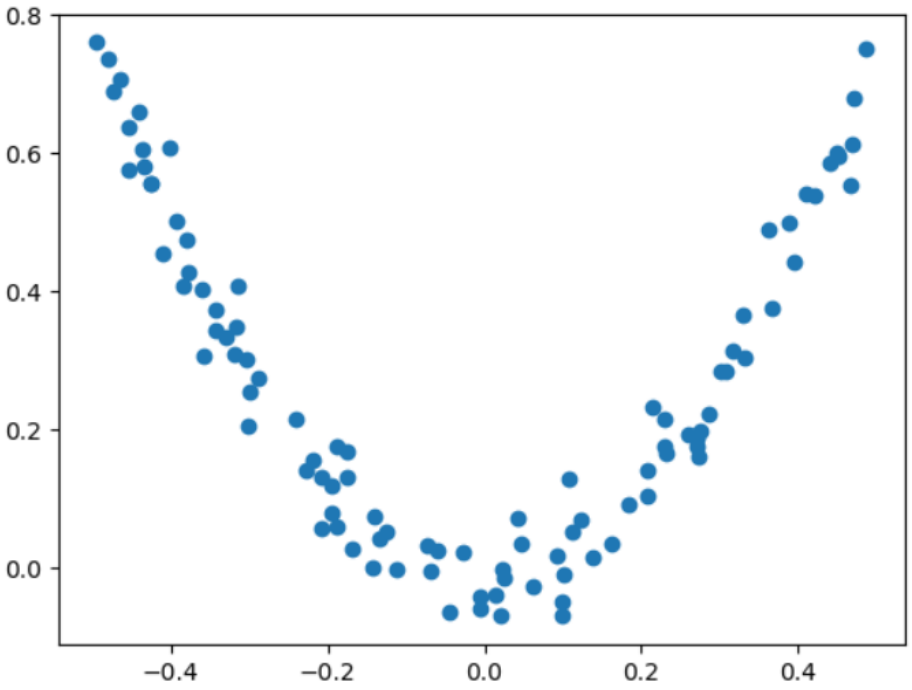
26. You are generating a dataset to use with a **Decision Tree Regressor**. First, you set a random seed for reproducibility and create an array of 100 random numbers between -0.5 and 0.5 to serve as the input features (X). The target values (y) are calculated using the formula y=3x2y = 3x^2y=3x2 plus some random Gaussian noise, creating a quadratic relationship with a bit of variability.
27. Next, you initialize a **Decision Tree Regressor** with a maximum depth of 2, which means the tree will have at most two levels, limiting its complexity. You then fit this decision tree model to the generated data, allowing it to learn the relationship between the input features and the target values. This setup is useful for understanding how a simple decision tree can model nonlinear relationships in data.
28. You are using the plt.scatter() function to create a scatter plot of the dataset. The plot displays the input features (X) on the horizontal axis and the corresponding target values (y) on the vertical axis. This visualization helps you see the distribution of the data points, illustrating how the target values vary with the input values.

In [17]:
```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)  # y = 3x² + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Out[17]:
```
DecisionTreeRegressor(max_depth=2, random_state=42)
```

In [18]:
```
plt.scatter(X,y)
```

Out[18]:
```
<matplotlib.collections.PathCollection at 0x203afc69bd0>
```



29. You are continuing the process of boosting by first calculating the new residuals (`y2`) after training the second decision tree regressor (`tree_reg2`). The residuals represent the remaining errors after the first model's predictions.
30. Next, you create another decision tree regressor (`tree_reg3`) with a maximum depth of 2 and fit it to the new residuals (`y3`), which are calculated by subtracting the predictions of the second tree from `y2`. This step allows the third tree to focus on correcting the errors from the second model.
31. Finally, you define a new array of input values (`X_new`) for which you want to make predictions. You use all three decision trees (`tree_reg1`, `tree_reg2`, and `tree_reg3`) to predict the values for `X_new` and sum their predictions. This combined prediction reflects the contributions of each tree, illustrating how boosting works by aggregating the outputs from multiple models to improve the overall prediction accuracy.

```
In [19]: y2 = y - tree_reg1.predict(X)
         tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
         tree_reg2.fit(X, y2)
```

```
Out[19]: DecisionTreeRegressor
         DecisionTreeRegressor(max_depth=2, random_state=43)
```

```
In [22]: y3 = y2 - tree_reg2.predict(X)
         tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
         tree_reg3.fit(X, y3)
```

```
Out[22]: DecisionTreeRegressor
         DecisionTreeRegressor(max_depth=2, random_state=44)
```

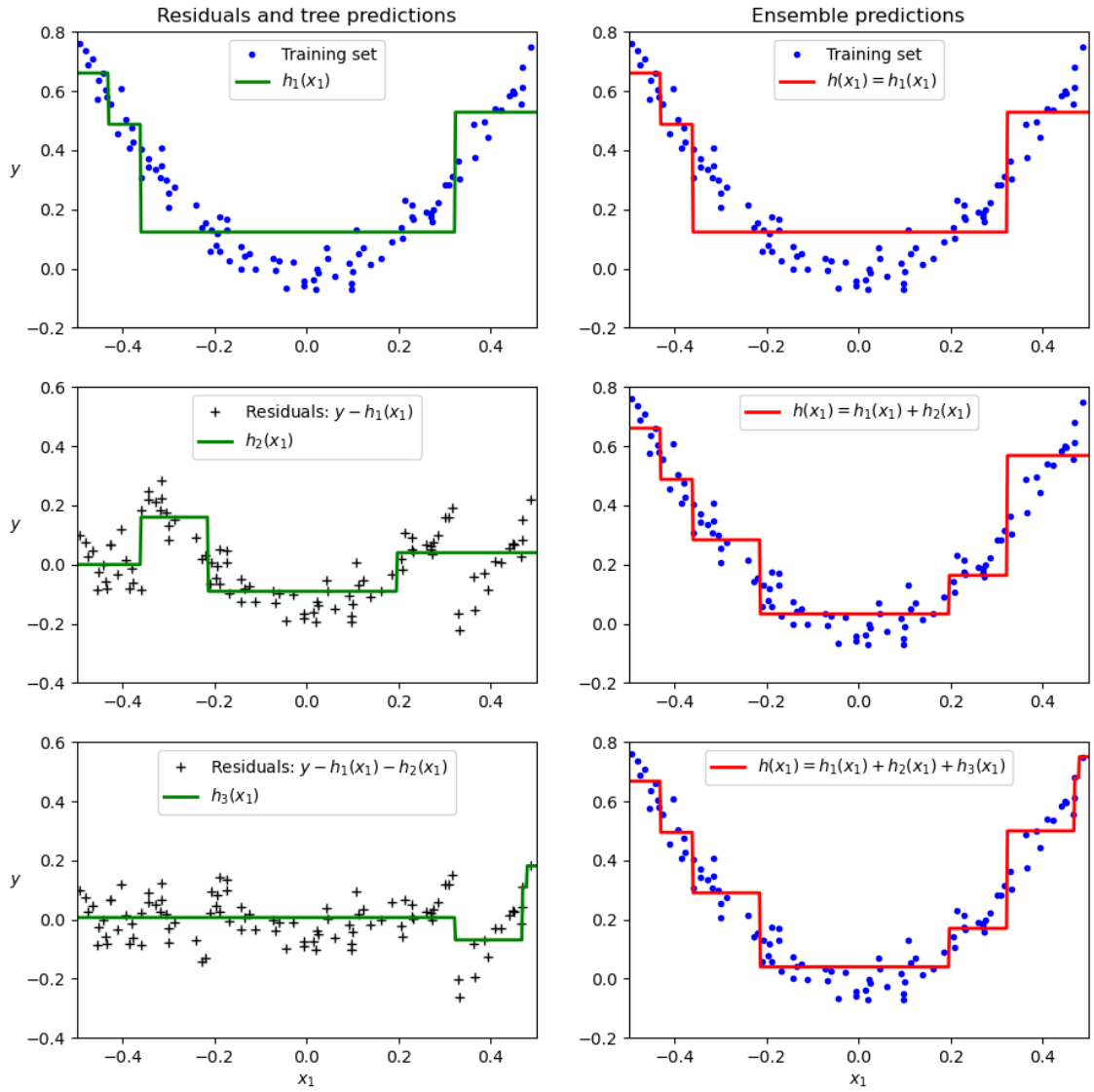
```
In [23]: X_new = np.array([[-0.4], [0.], [0.5]])
         sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

```
Out[23]: array([0.49484029, 0.04021166, 0.75026781])
```

32. In this code, you create a series of plots to visualize the predictions made by three decision tree regressors (tree_reg1, tree_reg2, and tree_reg3) in a boosting framework.
    - **Function Definition**: The plot_predictions function is defined to plot the predictions of the regressors against the training data. It takes several parameters, including the regressors, input data (X), target values (y), plotting axes, styles for the lines, and optional labels.
    - **Figure Setup**: A figure with a size of 11 by 11 is created to hold multiple subplots.
    - **Subplot 1**: The first subplot shows the predictions made by the first tree regressor (tree_reg1) on the training data, illustrating how it captures the general trend in the data.
    - **Subplot 2**: The second subplot demonstrates the ensemble prediction based solely on the first tree regressor.
    - **Subplot 3**: The third subplot visualizes the second tree regressor (tree_reg2) and its predictions on the residuals from the first tree's predictions, which helps illustrate how this model is focusing on correcting the errors from the first model.
    - **Subplot 4**: The fourth subplot shows the combined predictions of the first and second trees, illustrating the improvement in predictions by combining their outputs.
    - **Subplot 5**: The fifth subplot displays the predictions made by the third tree regressor (tree_reg3) on the residuals of the first two trees, again focusing on correcting the remaining errors.
    - **Subplot 6**: Finally, the sixth subplot shows the overall ensemble prediction, which includes contributions from all three trees, demonstrating how the boosting approach aggregates the predictions to improve accuracy.

- After setting up all the subplots, the final line displays the plots, allowing you to visually compare the predictions of each model and the overall ensemble performance.

In [24]:
```
def plot_predictions(regressors, X, y, axes, style,
                     label=None, data_style="b.", data_label=None):
    x1 = np.linspace(axes[0], axes[1], 500)
    y_pred = sum(regressor.predict(x1.reshape(-1, 1))
                 for regressor in regressors)
    plt.plot(X[:, 0], y, data_style, label=data_label)
    plt.plot(x1, y_pred, style, linewidth=2, label=label)
    if label or data_label:
        plt.legend(loc="upper center")
    plt.axis(axes)

plt.figure(figsize=(11, 11))

plt.subplot(3, 2, 1)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.2, 0.8], style="g-",
                 label="$h_1(x_1)$", data_label="Training set")
plt.ylabel("$y$  ", rotation=0)
plt.title("Residuals and tree predictions")

plt.subplot(3, 2, 2)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.2, 0.8], style="r-",
                 label="$h(x_1) = h_1(x_1)$", data_label="Training set")
plt.title("Ensemble predictions")

plt.subplot(3, 2, 3)
plot_predictions([tree_reg2], X, y2, axes=[-0.5, 0.5, -0.4, 0.6], style="g-",
                 label="$h_2(x_1)$", data_style="k+",
                 data_label="Residuals: $y - h_1(x_1)$")
plt.ylabel("$y$  ", rotation=0)

plt.subplot(3, 2, 4)
plot_predictions([tree_reg1, tree_reg2], X, y, axes=[-0.5, 0.5, -0.2, 0.8],
                 style="r-", label="$h(x_1) = h_1(x_1) + h_2(x_1)$")

plt.subplot(3, 2, 5)
plot_predictions([tree_reg3], X, y3, axes=[-0.5, 0.5, -0.4, 0.6], style="g-",
                 label="$h_3(x_1)$", data_style="k+",
                 data_label="Residuals: $y - h_1(x_1) - h_2(x_1)$")
plt.xlabel("$x_1$")
plt.ylabel("$y$  ", rotation=0)

plt.subplot(3, 2, 6)
plot_predictions([tree_reg1, tree_reg2, tree_reg3], X, y,
                 axes=[-0.5, 0.5, -0.2, 0.8], style="r-",
                 label="$h(x_1) = h_1(x_1) + h_2(x_1) + h_3(x_1)$")
plt.xlabel("$x_1$")

plt.show()
```



33. This code snippet sets up and trains two Gradient Boosting regression models using GradientBoostingRegressor from sklearn.ensemble.
34. First, it initializes a model (gbrt) with a maximum tree depth of 2, 3 estimators, and a learning rate of 1.0. It then fits this model to the dataset defined by X (features) and y (target values).
35. Next, a second model (gbrt_best) is created with a smaller learning rate of 0.05 and increased to 500 estimators, allowing it to potentially learn more subtle patterns in the data. The parameter n_iter_no_change=10 is used to stop training if the validation score doesn't improve for 10 iterations. After fitting this model to the data, the number of estimators used is retrieved.
36. Finally, two plots are created side by side to visualize the predictions of both models over the input feature x_1, showing how the change in learning rate and the number of estimators affect the ensemble predictions.

```
In [25]: from sklearn.ensemble import GradientBoostingRegressor

         gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                          learning_rate=1.0, random_state=42)
         gbrt.fit(X, y)
```

```
Out[25]: GradientBoostingRegressor
         GradientBoostingRegressor(learning_rate=1.0, max_depth=2, n_estimators=3,
                                   random_state=42)
```

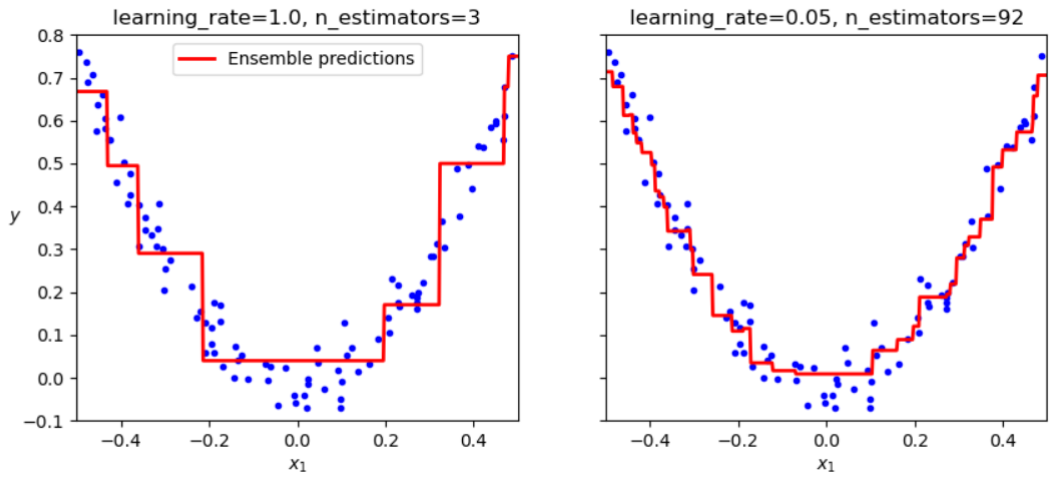
```
In [26]: gbrt_best = GradientBoostingRegressor(
             max_depth=2, learning_rate=0.05, n_estimators=500,
             n_iter_no_change=10, random_state=42)
         gbrt_best.fit(X, y)
```

```
Out[26]: GradientBoostingRegressor
         GradientBoostingRegressor(learning_rate=0.05, max_depth=2, n_estimators=500,
                                   n_iter_no_change=10, random_state=42)
```

```
In [27]: gbrt_best.n_estimators_
```

```
Out[27]: 92
```

```
In [28]: fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)

         plt.sca(axes[0])
         plot_predictions([gbrt], X, y, axes=[-0.5, 0.5, -0.1, 0.8], style="r-",
                          label="Ensemble predictions")
         plt.title(f"learning_rate={gbrt.learning_rate}, "
                   f"n_estimators={gbrt.n_estimators_}")
         plt.xlabel("$x_1$")
         plt.ylabel("$y$", rotation=0)

         plt.sca(axes[1])
         plot_predictions([gbrt_best], X, y, axes=[-0.5, 0.5, -0.1, 0.8], style="r-")
         plt.title(f"learning_rate={gbrt_best.learning_rate}, "
                   f"n_estimators={gbrt_best.n_estimators_}")
         plt.xlabel("$x_1$")

         plt.show()
```



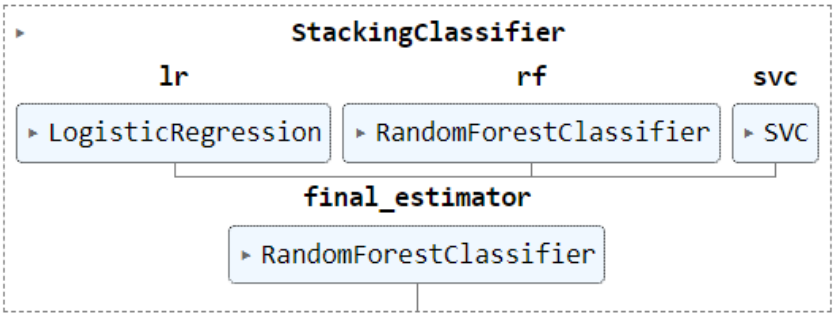
37. The code demonstrates the use of the `StackingClassifier` from `sklearn.ensemble` to create an ensemble model that combines three base estimators: Logistic Regression, Random Forest, and Support Vector Classifier, with a final estimator of Random

Forest; it employs five-fold cross-validation during training and is fitted using the training data (`X_train`, `y_train`), and the model achieves an accuracy score of approximately 0.928 on the test data (`X_test`, `y_test`), showcasing the effectiveness of stacking for improving predictive performance by leveraging the strengths of multiple machine learning algorithms.

Stacking

In [29]:
```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5  # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

Out[29]:



In [30]:
```
stacking_clf.score(X_test, y_test)
```

Out[30]:
```
0.928
```