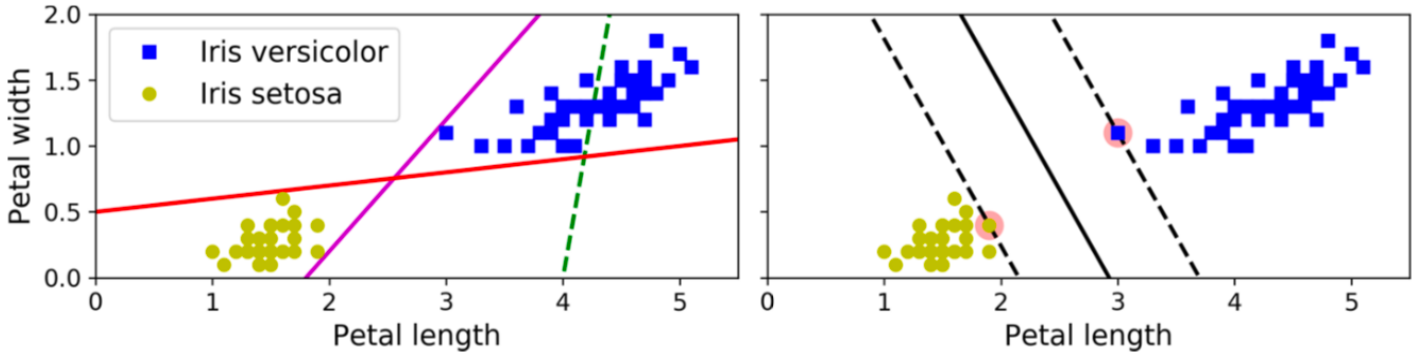
Data-intensive space engineering

Lecture 4

Carlos Sanmiguel Vila

Support Vector Machines

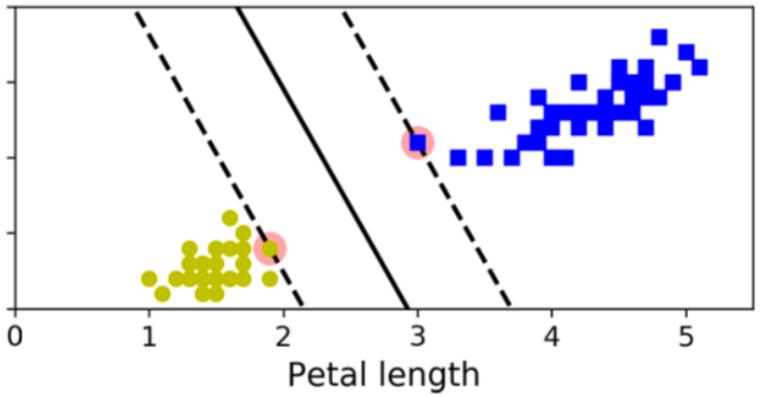
- A support vector machine (SVM) is a machine learning model, capable of performing linear or nonlinear classification, regression, and even novelty detection.
- SVMs shine with small to medium sized nonlinear datasets (i.e., 100/1000 instances), especially for classification tasks. However, they don't scale very well to very large datasets.



- An SVM classifier looks not only to separate the two classes but also to stay as far away from the closest training instances as possible.

Support Vector Machines

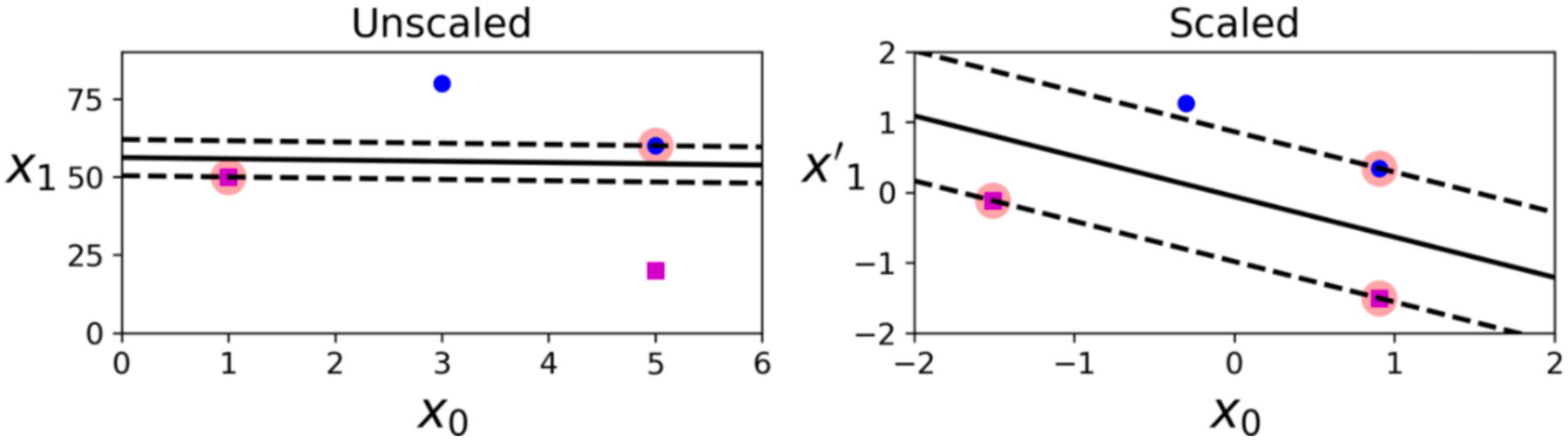
- We look for a large margin classification. An SVM classifier tries to fit the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.



- Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure).

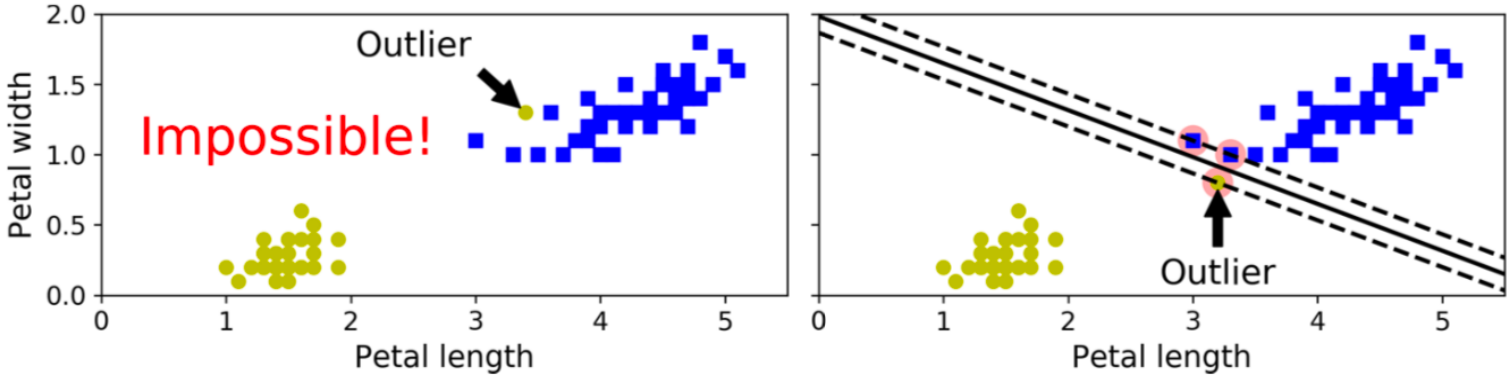
Challenges of using SVMs

- SVMs are very sensitive to the feature scales.



Challenges of using SVMs

- **Hard margin classification** requires that all training instances are correctly classified
- It only works if the dataset is linearly separable
- Sensitive to outliers.



How to solve this problem? -> **Soft margin classification**

• We use a more flexible model to find a good balance between a large margin and limiting margin violations (i.e., instances that end up in the middle of the street or even on the wrong side).

SVMs in Sklearn

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

scaler = StandardScaler()
svm_clf1 = LinearSVC(C=1, loss="hinge", random_state=42)
svm_clf2 = LinearSVC(C=100, loss="hinge", random_state=42)

scaled_svm_clf1 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ])
scaled_svm_clf2 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ])

scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)
```

C (Regularization Parameter): Controls the trade-off between maximizing the margin and minimizing classification errors. A small CCC leads to a wider margin with more misclassifications (potentially underfitting), while a large CCC tries to classify all training examples correctly (potentially overfitting).

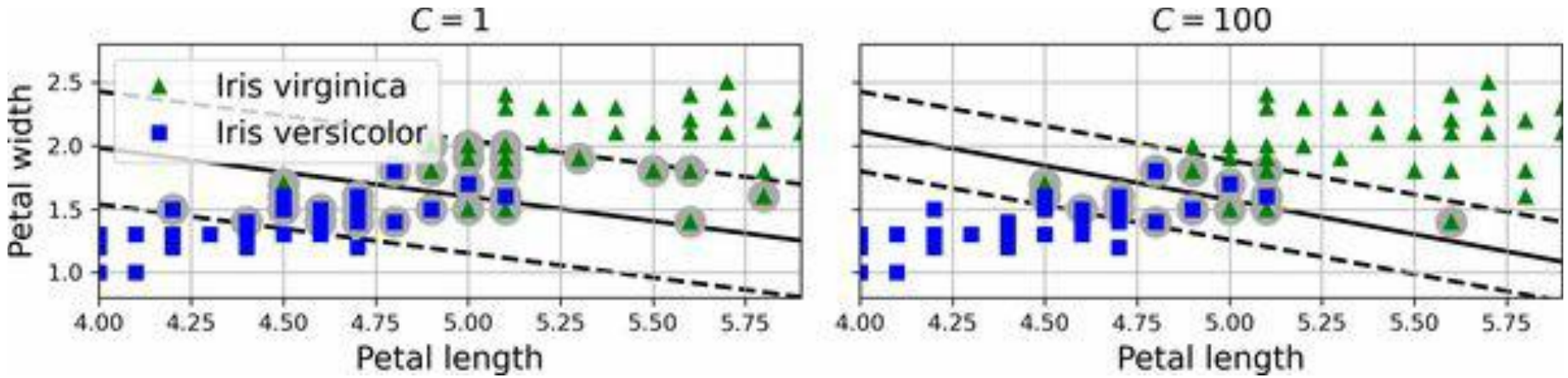
Hinge Loss is a loss function used primarily in **Support Vector Machines (SVM)** for classification tasks. It's designed to **maximize the margin** between different classes by penalizing misclassified points and those that are too close to the decision boundary.

The hinge loss ensures that the SVM focuses on correctly classifying data points and maximizing the margin between the classes.

This is an example from the reference book. To see a course implementation, check AulaGlobal Notebook

SVMs in Sklearn

- The strength of the regularization is inversely proportional to C
- If your SVM model is overfitting, you can try regularizing it by decreasing C
- Reducing C makes the street larger, but it also leads to more margin violations



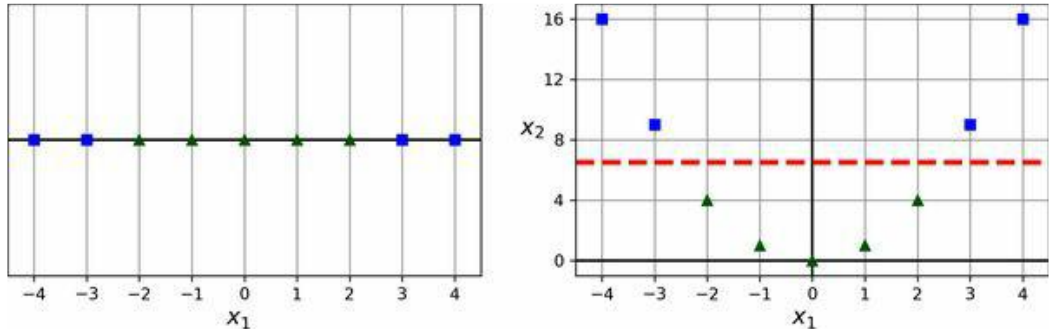
Hinge Loss

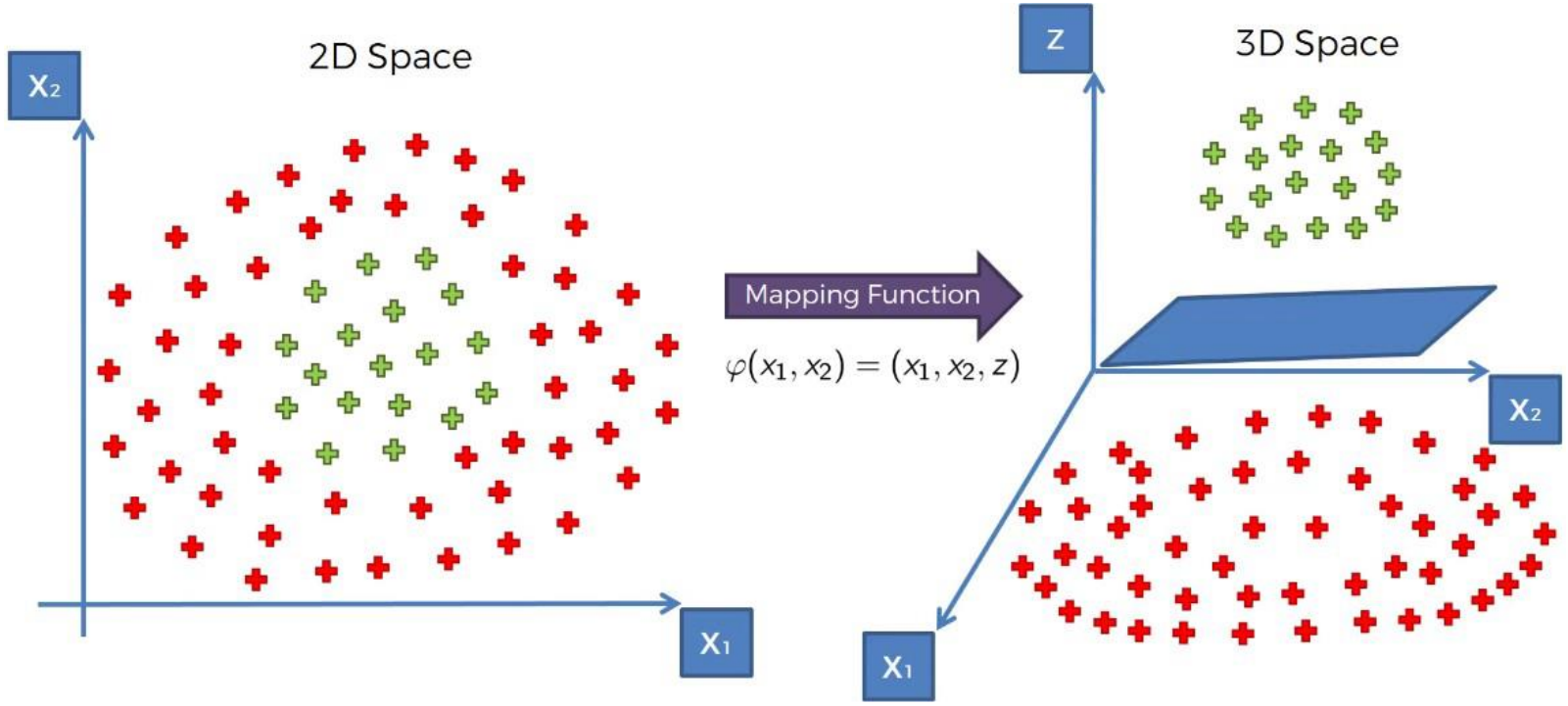
$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

Where:

- $y \in \{-1, 1\}$ is the true label of the data point (binary classification).
- $f(x)$ is the predicted value (distance from the decision boundary).
- The loss is zero if $y \cdot f(x) \geq 1$ (correct classification with sufficient margin), and positive otherwise.

Nonlinear SVM Classification





Nonlinear SVM Classification

- In practice, the SVM algorithm is implemented using a kernel, which transforms an input data space into the required form.
- SVM uses a technique called the kernel trick. Here, the kernel transforms a low-dimensional input space into a higher-dimensional space.
- In other words, you can say that it converts non-separable problem to separable problems by adding more dimension to it.
- It is most useful in non-linear separation problems. The kernel trick helps you to build a more accurate classifier.
- A kernel function takes two inputs (data points) and returns the inner product in the new, higher-dimensional space. You don’t need to calculate the transformation to the higher-dimensional space explicitly; the kernel function does this indirectly by calculating the distance between two points in that space.

Nonlinear SVM Classification

Here are the common types of kernels:

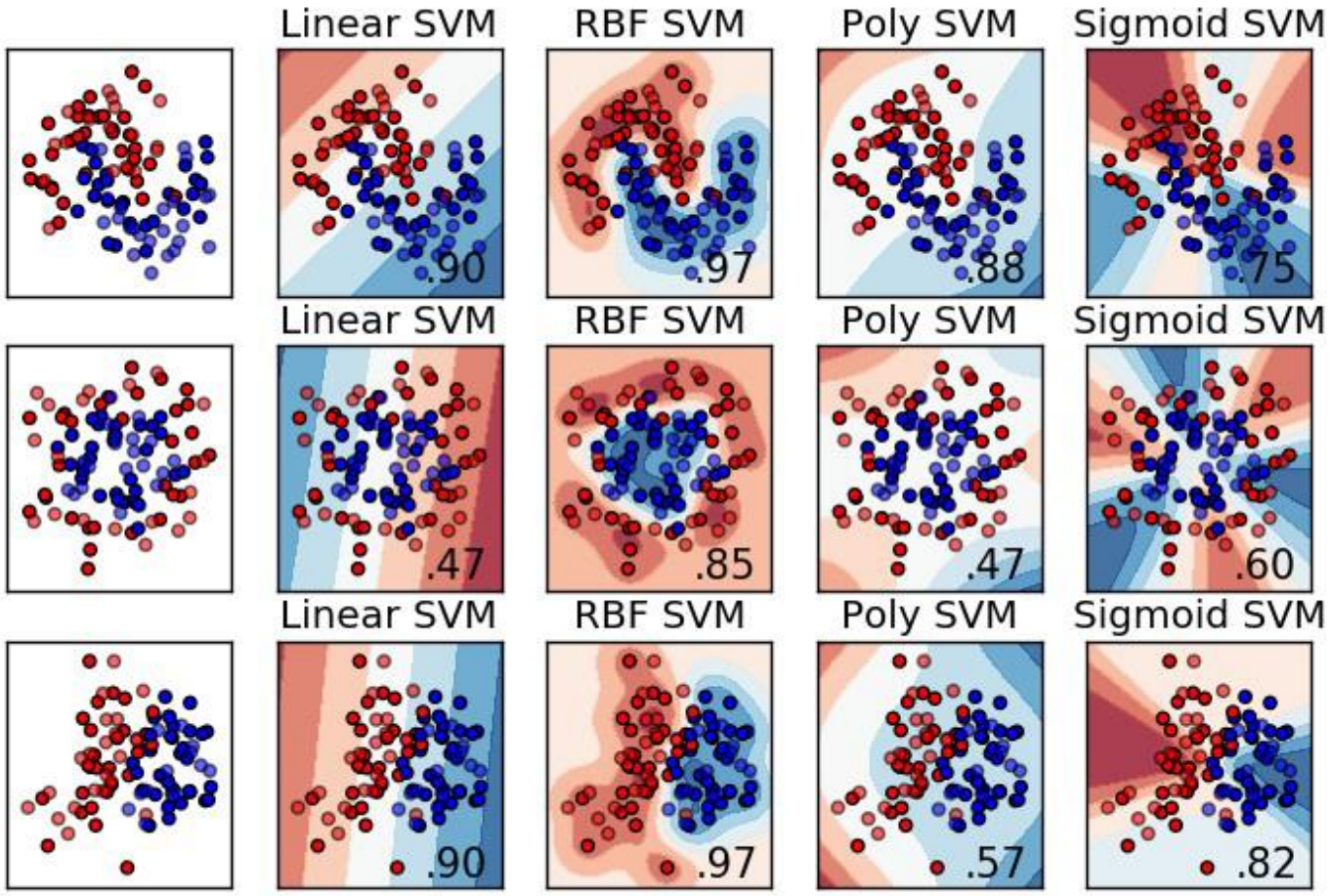
1. Linear Kernel:
Equivalent to no transformation. This is used when the data is linearly separable.
Inner product: $K(x_i, x_j) = x_i \cdot x_j$

2. Polynomial Kernel:
Maps the original features into a polynomial feature space.
Inner product: $K(x_i, x_j) = (x_i \cdot x_j + 1)^d$, where d is the degree of the polynomial.

3. Radial Basis Function (RBF) Kernel:
Maps the data into an infinite-dimensional space and is widely used for non-linear problems.
Inner product: $K(x_i, x_j) = e^{-\gamma}||x_i - x_j||^2$, where γ is a parameter that controls the influence of a training example.

4. Sigmoid Kernel:
Similar to the activation function in neural networks, it can be useful for certain types of binary classification problems.
Inner product: $K(x_i, x_j) = \tanh(\alpha(x_i \cdot x_j) + c)$

Types of kernels



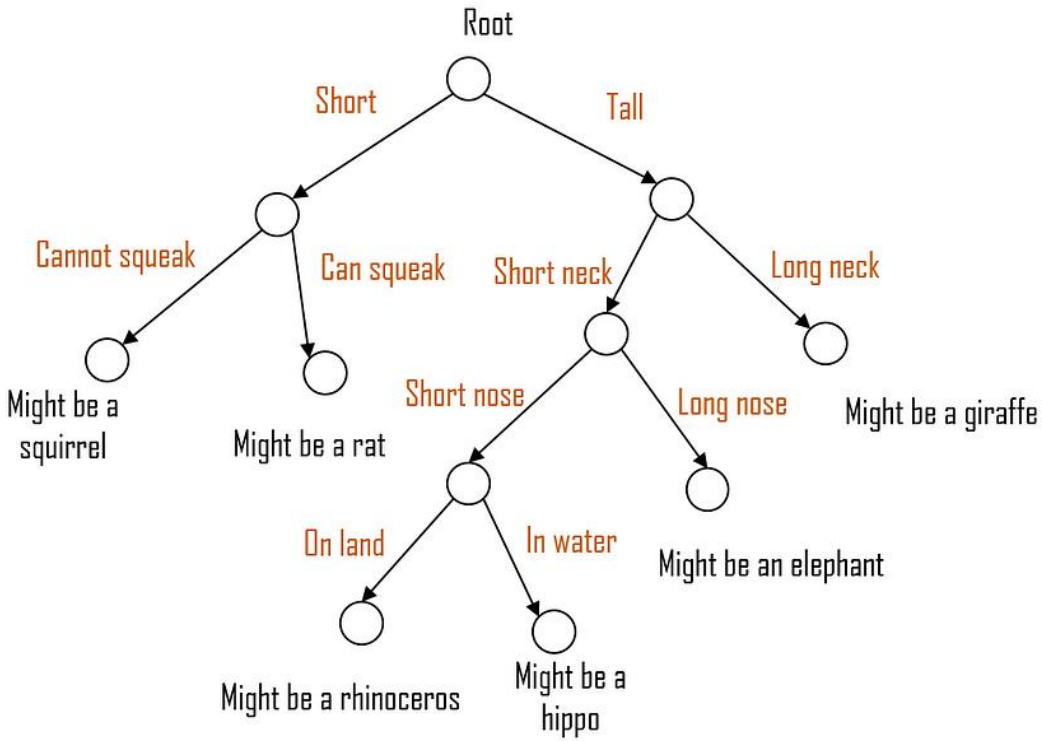
SVM vs. Logistic Regression

Algorithm	Pros	Cons
Logistic Regression	- Simple and interpretable - Works well for linear separability - Probabilistic outputs	- Limited to linear decision boundaries - Sensitive to outliers - Struggles with non-linear data
SVM (Linear Kernel)	- Effective for linearly separable data - Maximizes margin - Robust to outliers	- Linear decision boundary (same as logistic regression) - Not probabilistic by default
SVM (Non-Linear Kernel)	- Powerful for non-linearly separable data - Flexible with kernels - Robust to outliers	- Computationally expensive for large datasets - Requires hyperparameter tuning (e.g., C, gamma)

Decision Tree

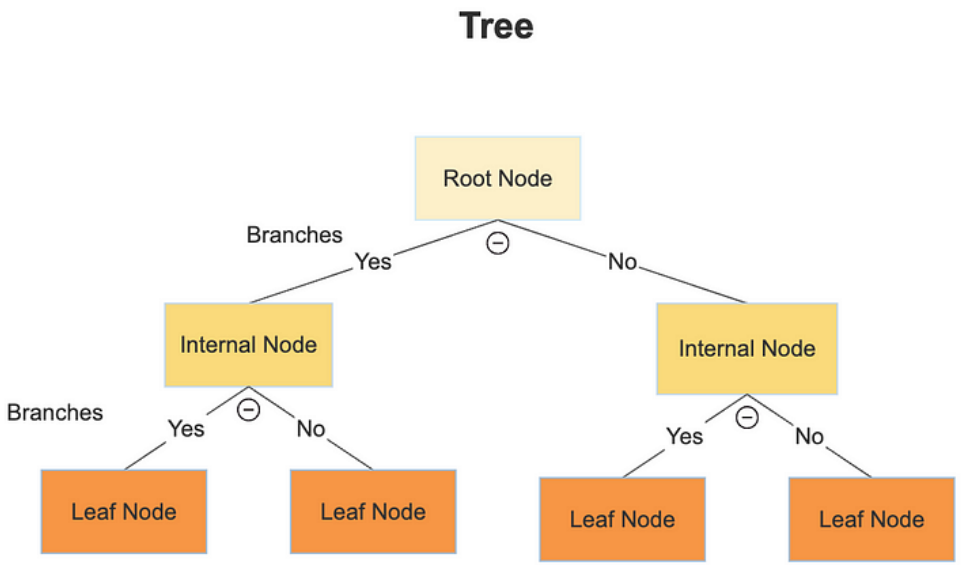
- A Decision Tree is a supervised machine learning algorithm for classification and regression tasks. It splits the data into subsets based on the feature values, forming a tree-like structure.

- Each internal node represents a "decision" based on a feature, each branch represents the outcome of the decision, and each leaf represents a class or a value for regression.



Structure of a Decision Tree

- **Root Node:** The topmost node representing the entire dataset, which is then divided into two or more homogeneous sets.
- **Decision Nodes:** These are sub-nodes that split further and represent decisions made on the features.
- **Leaf Nodes:** Also known as terminal nodes, these nodes represent the final output or decision, whether it's a class label or a value.
- **Branches:** These are the outcomes of the decision nodes, leading to the next decision node or a leaf node.



How Does a Decision Tree Work?

The process of building a decision tree involves selecting the best attribute to split the data at each node. This selection is based on criteria like Gini impurity, entropy, or variance reduction.

- **Choosing the Best Split:** At each node, the algorithm evaluates all possible splits and selects the one that best separates the data. For classification trees, criteria like Gini impurity or information gain (entropy) are used. For regression trees, the reduction in variance is a common criterion.
- **Splitting:** Once the best split is chosen, the data is divided accordingly, and the process is recursively repeated for each subset.
- **Stopping Criteria:** The tree continues to grow until a stopping criterion is met. This could be a maximum depth, a minimum number of samples in a node, or a threshold for the impurity measure.

Entropy

Entropy measures randomness or uncertainty in a dataset. In the context of decision trees, it quantifies the impurity or disorder in a set of examples.

•If a dataset contains examples from only one class, its entropy is zero, indicating complete purity.
•If the dataset is evenly split between classes, its entropy is at its maximum of 1, indicating maximum disorder.
The formula for entropy (D) for a classification problem is:

$$H(D) = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

p_i represents the probability of a data point belonging to class iii in the dataset D. Specifically:
•p_i is the proportion of data points in the dataset that belong to class i, calculated as:

$$p_i = \frac{|D_i|}{|D|}$$

Where:
• $|D_i|$ is the number of data points in class iii.
• $|D|$ is the total number of data points in the dataset.

Gini Impurity

Gini impurity is another measure of impurity used in decision trees. It represents the probability of incorrectly classifying a randomly chosen element if it were labeled according to the distribution of labels in the subset.

Gini impurity is zero when all elements belong to a single class (pure).
It reaches its maximum (0.5 for binary classification) when the elements are equally distributed among the classes.
The formula for Gini impurity (G) is:

$$G(D) = 1 - \sum_{i=1}^{C} p_i^2$$

Generally, gini impurity should be used in large datasets, and entropy should be used in smaller datasets since entropy has a log function, which makes calculation harder.

Information Gain

Information gain measures the reduction in entropy or impurity after a dataset is split based on an attribute. It quantifies the effectiveness of an attribute in classifying the training data.

•A high information gain indicates that the attribute has effectively split the data into pure subsets.

•It's calculated as the difference between the entropy of the original set and the weighted sum of the entropies of the subsets created by the split.

The formula for information gain (IG) is:

$$IG(D, A) = H(D) - \sum_{v \in V(A)} \frac{|D_v|}{|D|} H(D_v)$$

•$H(D)$: The entropy of the dataset D before the split.

•$H(D_v)$: The entropy of the subset of the dataset D_v after splitting on value v of attribute A.

•$|D_v|$: The number of samples in the subset corresponding to the value v of attribute A.

•$|D|$: The total number of samples in the dataset.

Pre-Pruning and Post-Pruning

Decision trees are powerful tools for classification and regression tasks due to their simplicity and interpretability. However, they are prone to overfitting, especially when they grow too deep and learn the noise in the training data. To combat this, pruning techniques—pre-pruning and post-pruning—are used to improve the model's generalization performance.

What is Pruning?

Pruning in decision trees refers to the process of reducing the size of the tree by removing sections that provide little power in classifying instances. The goal of pruning is to improve the model's performance on unseen data by reducing overfitting.

Pre-Pruning

Pre-pruning, also known as early stopping, involves halting the growth of the decision tree before it becomes too complex. This is done by setting certain conditions that must be met for a split to occur.

Methods of Pre-Pruning

1. **Maximum Depth**: Limit the maximum depth of the tree. Once the tree reaches this depth, no further splits are made.
2. **Minimum Samples for Split**: Specify the minimum number of samples required to split a node. If a node has fewer samples than this threshold, it is not split further.
3. **Minimum Samples per Leaf**: Define the minimum number of samples that a leaf node must have. Nodes that do not meet this criterion are not created.
4. **Maximum Number of Nodes**: Set a limit on the total number of nodes in the tree. Once this limit is reached, no additional nodes are created.

Pre-Pruning

Advantages of Pre-Pruning

- **Efficiency**: Trees are smaller and less complex, making them faster to construct and evaluate.
- **Reduced Overfitting**: By stopping the growth early, the model is less likely to learn noise from the training data.

Disadvantages of Pre-Pruning

- **Underfitting**: There is a risk of stopping the growth too early, resulting in a model that is too simple and does not capture the underlying patterns in the data.

Post-Pruning

Methods of Post-Pruning

1.Reduced Error Pruning: Remove nodes if the removal improves the performance of the tree on a validation dataset.
2.Cost Complexity Pruning (CCP): Introduce a cost complexity parameter that balances the tree's size and its performance. Nodes are pruned if the cost complexity criterion is improved.
3.Minimal Cost-Complexity Pruning: Calculate the cost complexity for each subtree and prune the tree iteratively to minimize this cost.

Post-Pruning

Advantages of Post-Pruning

•**Better Performance**: By allowing the tree to grow fully before pruning, the model can capture more complex patterns in the data before simplifying.
•**More Control**: Post-pruning allows for more refined adjustments, as the full tree is available for analysis.

Disadvantages of Post-Pruning

•**Computationally Intensive**: Growing a full tree and then pruning it can be more time-consuming and computationally expensive.
•**Complexity**: Requires more sophisticated techniques and validation datasets to determine which nodes to prune.

Decision Trees - Regression

In the context of regression, decision trees aim to predict a continuous target variable. One crucial aspect of decision tree regression is variance reduction, which is used to determine the best splits.

What is Variance?
Remember in statistics, variance measures the spread or dispersion of a set of values. It quantifies how much the values differ from the mean (average) of the dataset. In decision tree regression, variance is used to assess the homogeneity of the target variable within subsets of the data. Lower variance indicates that the values are closer to the mean and, therefore, more homogeneous.

Decision Trees - Regression

Variance Reduction in Decision Trees

Variance reduction is the criterion for selecting the best split at each node in a decision tree regressor. The goal is to minimize the variance within each of the resulting subsets after a split, leading to more accurate predictions.

1. **Before Split**: The variance of the target variable in the parent node is calculated.

2. **After Split**: The variance of the target variable in each of the child nodes is calculated.

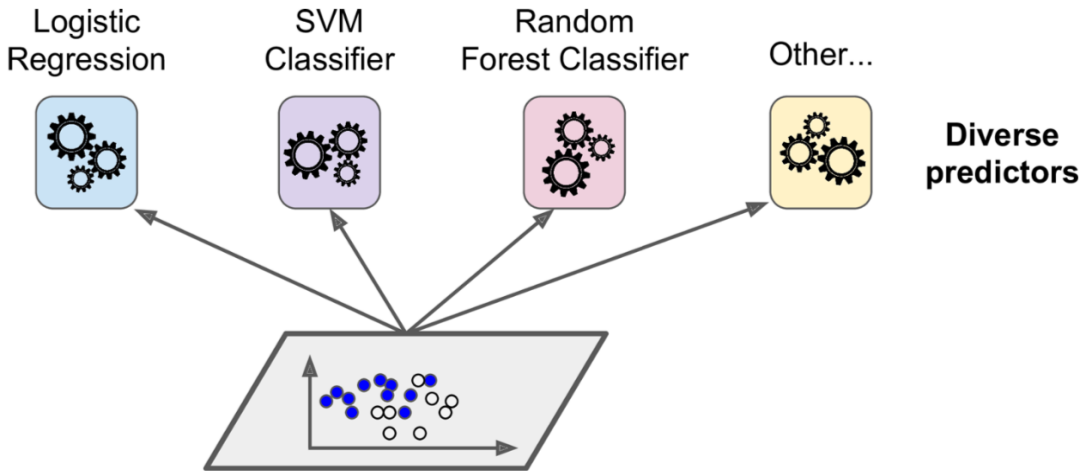
3. **Variance Reduction**: The reduction in variance is computed as the difference between the variance of the parent node and the weighted sum of the variances of the child nodes. The split that results in the maximum variance reduction is chosen.

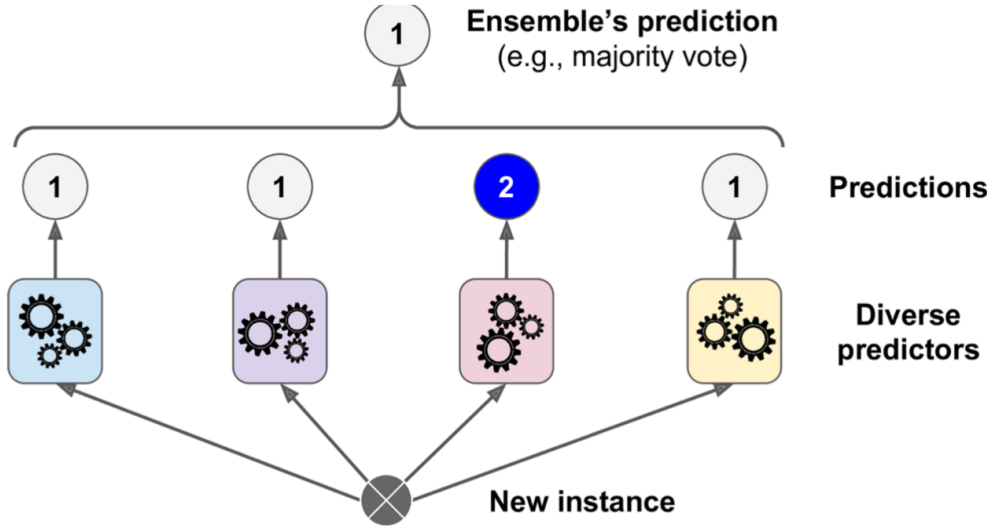
Decision Trees

Algorithm	Pros	Cons	When to Use
Decision Trees	- Easy to interpret and visualize. - No need for feature scaling. - Handles both categorical and continuous data. - Fast for training and prediction. - Can handle non-linear relationships. - Provides feature importance.	- Prone to overfitting without regularization (max depth, min samples split). - Instability: small changes in data can lead to very different trees. - Generally, lower predictive accuracy than ensemble methods.	- When interpretability and visualization are important. - Small to medium-sized datasets. - Can be a weak learner in ensemble methods like Random Forests.

Ensemble learning

•Ensemble learning is a machine learning technique where multiple models (often called **weak learners**) are trained and combined to solve the same problem.
•The idea is that a group of weak learners can come together to form a **strong learner** that achieves better accuracy and generalization.

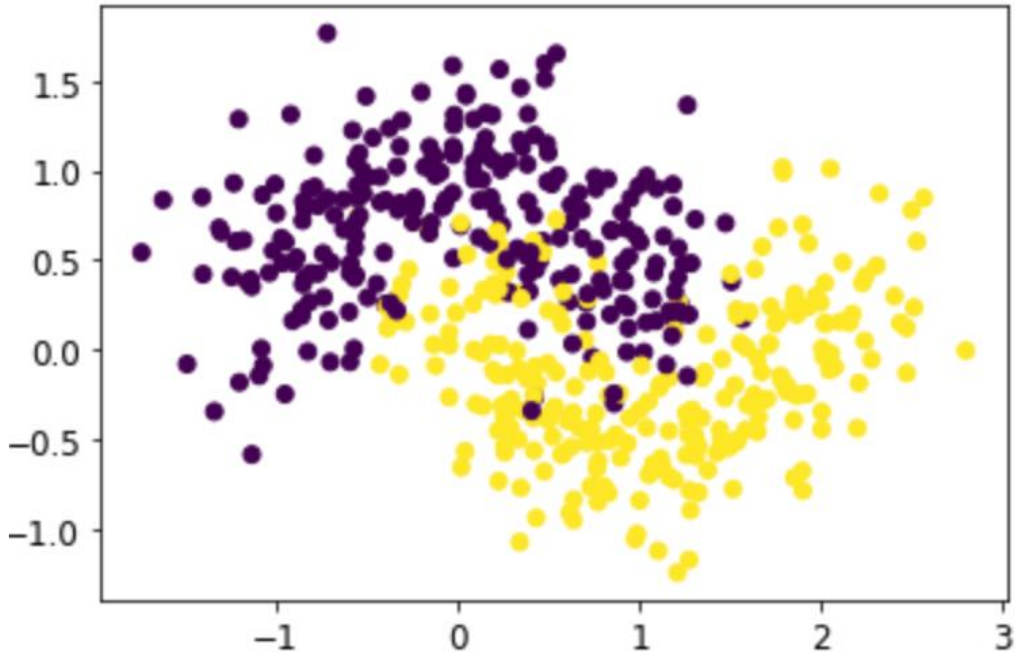




Ensemble learning

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```



Ensemble learning

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.912
```

Why Use Ensemble Learning?

- Increases accuracy by reducing bias and variance.
- Reduces overfitting by averaging multiple models.
- More robust to outliers and noise in the data.

Types of Ensemble Learning:

1.Bagging (Bootstrap Aggregating):

- Builds multiple models in parallel using different subsets of the training data (with replacement).
- Each model votes (for classification) or averages (for regression) to produce the final result.
- Example: **Random Forests**.

2.Boosting:

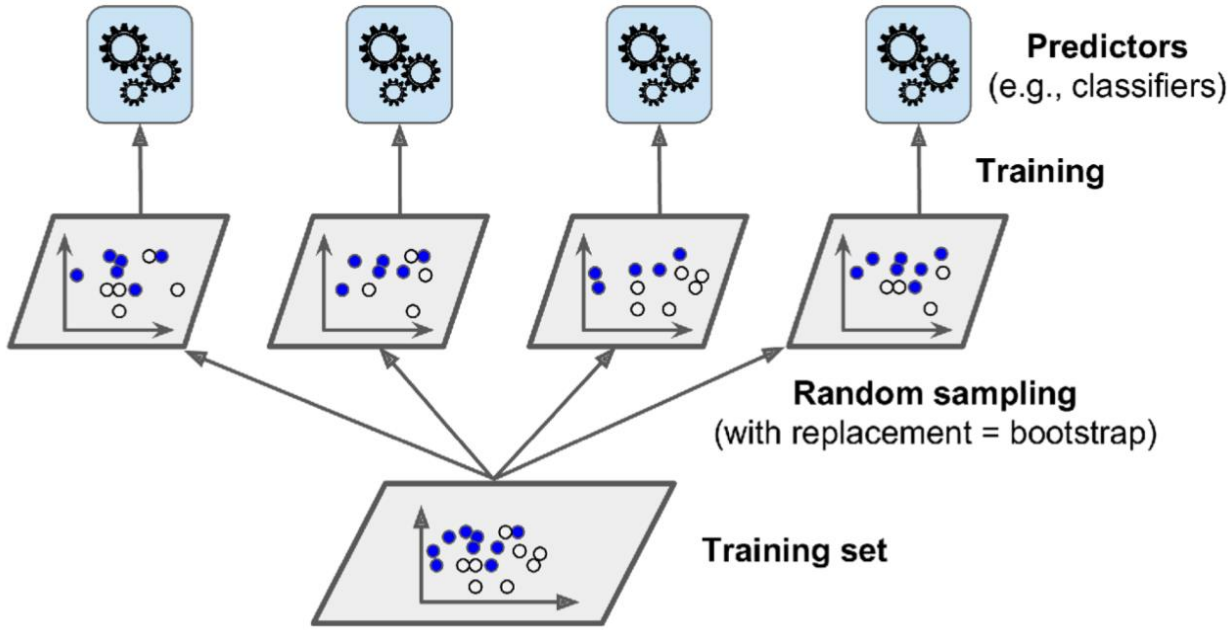
- Models are trained sequentially. Each model tries to correct the errors made by the previous one.
- Example: **AdaBoost**, **Gradient Boosting**.

3.Stacking:

- Different models are trained, and their outputs are combined using a meta-model that learns how to combine them best.

Bagging and pasting

- In the previous example, we used a diverse set of classifiers
- Another approach is to use the same learning algorithm and train them on different random subsets of the training set
  - Sampling with replacement: bagging
  - Sampling without replacement: pasting



Bagging

- Ensemble of 500 Decision Tree classifiers, each trained on 100 training instances

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

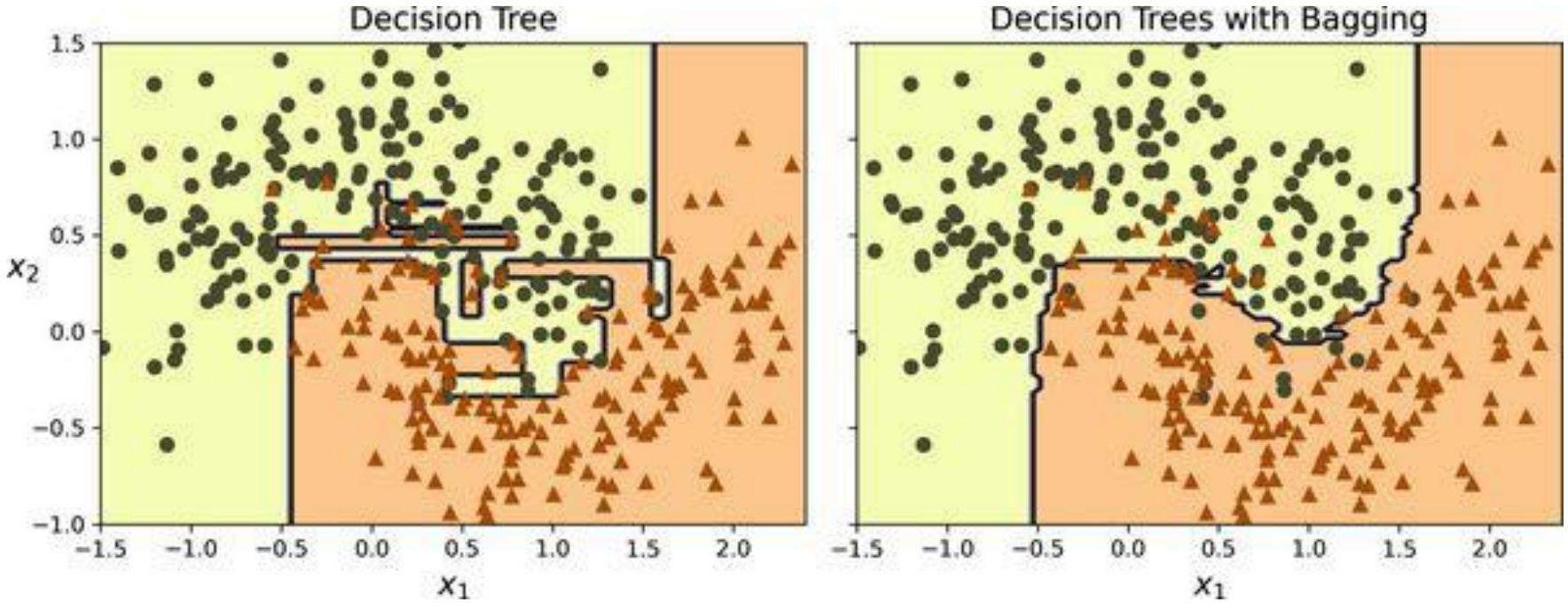
```
0.904
```

- Compare with a single Decision Tree classifier

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

```
0.856
```

Bagging


Decision Tree
Decision Trees with Bagging
x_1
x_2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2.0

Random Forests

Random Forest is an ensemble method that combines multiple **Decision Trees** to create a stronger model.
It uses **Bagging** and **Feature Randomization** to grow each tree from a random subset of the data and features.
Key Concepts:

- **Bootstrap Sampling**: Each Decision Tree is trained on a random sample of the data with replacement.
- **Random Feature Selection**: At each split in a tree, only a random subset of features is considered. This reduces correlation between trees.
- **Voting**: For classification, the final prediction is the majority vote of all trees. For regression, the output is the average of all trees.

Random Forests

Why Random Forests are Effective

- **Reduces Overfitting**: Because each tree is trained on a different subset of data and features, overfitting is minimized.
- **Feature Importance**: Random Forests provide insights into which features are most important by averaging feature importance across all trees.
- **Handles Missing Data**: Random Forests can handle missing data by averaging over trees that use different data splits.

Random Forests

Hyperparameters in Random Forests

- **n_estimators**: The number of trees in the forest. More trees improve performance, but require more computation.
- **max_depth**: Limits how deep each tree can grow. Limiting depth helps avoid overfitting.
- **max_features**: The number of features to consider when looking for the best split. Lower values reduce overfitting but may limit model capacity.
- **min_samples_split**: The minimum number of samples required to split an internal node.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node.

Random Forests

Advantages	Disadvantages
- Handles large datasets well.	- Can be computationally expensive with many trees.
- Robust to overfitting.	- Interpretability is reduced compared to a single Decision Tree.
- Provides feature importance measures.	- Requires tuning of hyperparameters for best performance.
- Works well with both categorical and continuous data.	- May not perform as well as Boosting in some cases, especially with complex data.