Machine Learning & Deep Learning

Reda Khoufache & Bilal FAYE

Master 2 MIAGE & BUT3 Informatique

Université Paris Descartes & UIT Villetaneuse

2023/2024

Outline

Introduction

Motivation

Course organization

Outline

Machine Learning

Definition

Machine learning is a research field that combines statistics, artificial intelligence, and computer science to extract knowledge from data. It is also known as predictive analytics or statistical learning. In recent years, machine learning has become widespread in everyday life, powering automatic recommendations, personalized online services, and recognition systems. Commercial websites like Facebook, Amazon, and Netflix heavily rely on machine learning algorithms throughout their platforms.

Machine learning: Why Machine learning?

- In the early days, "intelligent" applications relied on handcrafted rules (if-else decisions) for data processing and user interactions, e.g., using word blacklists for spam filters.
- However, this approach had limitations: domain-specific logic and deep human expertise were required, leading to inflexibility in handling new tasks.
- Detecting faces in images highlighted the challenges of handcoding, as computer perception differed from human perception, making rule design difficult.
- In contrast, machine learning enables algorithms to identify face characteristics without explicit rules. For example, providing a program with a large collection of face images allows it to learn what constitutes a face.
- This data-driven approach of machine learning overcomes the manual limitations of rule-based systems and opens up possibilities for tackling complex tasks across various domains.

Machine learning: Problems Machine Learning Can Solve

Machine learning is a powerful field that enables automated decision-making by learning from data. In this context, supervised learning involves providing algorithms with input-output pairs, while unsupervised learning deals with input data only. Let's explore some practical examples of both types of machine learning tasks:

- In supervised learning:
  - Identifying zip codes from handwritten digits on an envelope.
  - Determining whether a tumor is benign based on a medical image.
  - Detecting fraudulent activity in credit card transactions.
- In unsupervised learning:
  - Identifying topics in a set of blog posts.
  - Segmenting customers into groups with similar preferences.
  - Detecting abnormal access patterns to a website.

Machine Learning: Knowing Your Task and Knowing Your Data

Understanding your data and its relevance is essential in machine learning. Avoid randomly applying algorithms; comprehend your dataset before building a model. Tailor algorithms to your problem. Answer questions like:

- Can the data answer my questions?
- How can I frame questions as ML problems?
- Is there enough representative data?
- Are extracted features suitable for predictions?
- How will success be measured?
- How does the ML solution fit into the larger context?

Stay aware of assumptions while building models. Remember, ML is a part of problem-solving, and complex solutions might not solve the right problem.

Machine Learning: Why Python?

Python is a versatile language for data science, with libraries for various tasks like **data loading**, **visualization**, **stats**, and more. It enables quick iteration and interaction through tools like **Jupyter Notebook**. Python's flexibility extends to **GUIs**, **web services**, and integration into existing systems.

Machine Learning: Essential Libraries and Tools

- **scikit-learn**: Machine learning library for various tasks like classification and regression.
- **NumPy**: Core package for numerical computations using arrays and matrices.
- **SciPy**: Extends NumPy with advanced scientific computing features.
- **matplotlib**: Popular plotting library for creating visualizations.
- **pandas**: Data manipulation library for efficient data analysis.
- **mglearn**: Provides tools and datasets for learning and practicing machine learning concepts.

Machine Learning: Essential Libraries and Tools

```
# install packages
pip install numpy scipy matplotlib ipython scikit-learn pandas mglearn
# import some packages
import numpy as np
import sklearn
import scipy as sp
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
```

Machine Learning: A First Application: Classifying Iris Species

The application on the Iris dataset is a classic example in machine learning. The Iris dataset contains measurements of petal and sepal length and width for three species of iris. The goal is to develop a model that can automatically classify iris flowers based on these measurements, showcasing the use of machine learning for classification based on distinct botanical features.

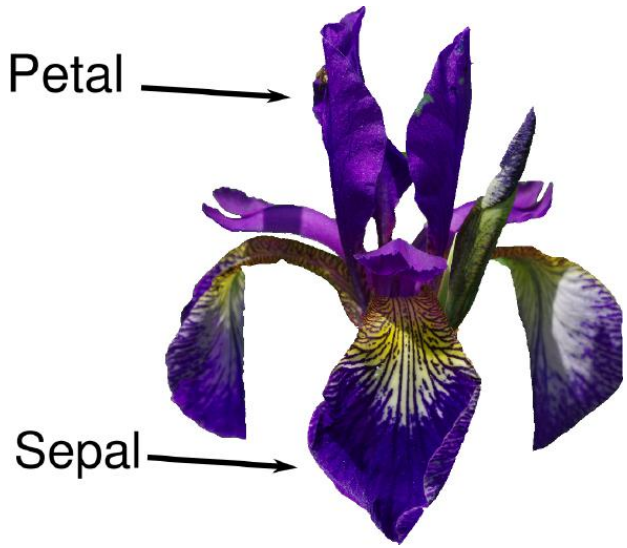

Figure: Parts of the iris flower

Machine Learning: A First Application: Classifying Iris Species

```
# Load the iris dataset
from sklearn.datasets import load_iris
iris_dataset = load_iris()

# Keys on iris dataset
print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))
```

Machine Learning: A First Application: Classifying Iris Species

Evaluating the model's performance requires separate data to test its generalization. We split our dataset into a training set (75%) and a test set (25%). In scikit-learn, data is represented by capital X and labels by lowercase y. By utilizing the train_test_split function, we can establish these sets and follow standard mathematical conventions for denoting input (X) and output (y) data.

Machine Learning: A First Application: Classifying Iris Species

```
# Split iris dataset to train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
iris_dataset['data'], iris_dataset['target'], random_state=0)

# Train shape
print("X_train shape: {}".format(X_train.shape))
print("y_train shape: {}".format(y_train.shape))

# Test shape
print("X_test shape: {}".format(X_test.shape))
print("y_test shape: {}".format(y_test.shape))
```

Machine Learning: A First Application: Classifying Iris Species

It is highly recommended to conduct an exploratory analysis of the dataset to gain insights before applying a machine learning algorithm. This approach helps in forming a comprehensive understanding. To illustrate, let's showcase a visualization of the dataset.

```
# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
grr = pd.plotting.scatter_matrix(iris_dataframe, c=y_train,
figsize=(15, 15), marker='o', hist_kwds={'bins': 20}, s=60,
alpha=.8, cmap=mglearn.cm3)
```

Machine Learning: A First Application: Classifying Iris Species

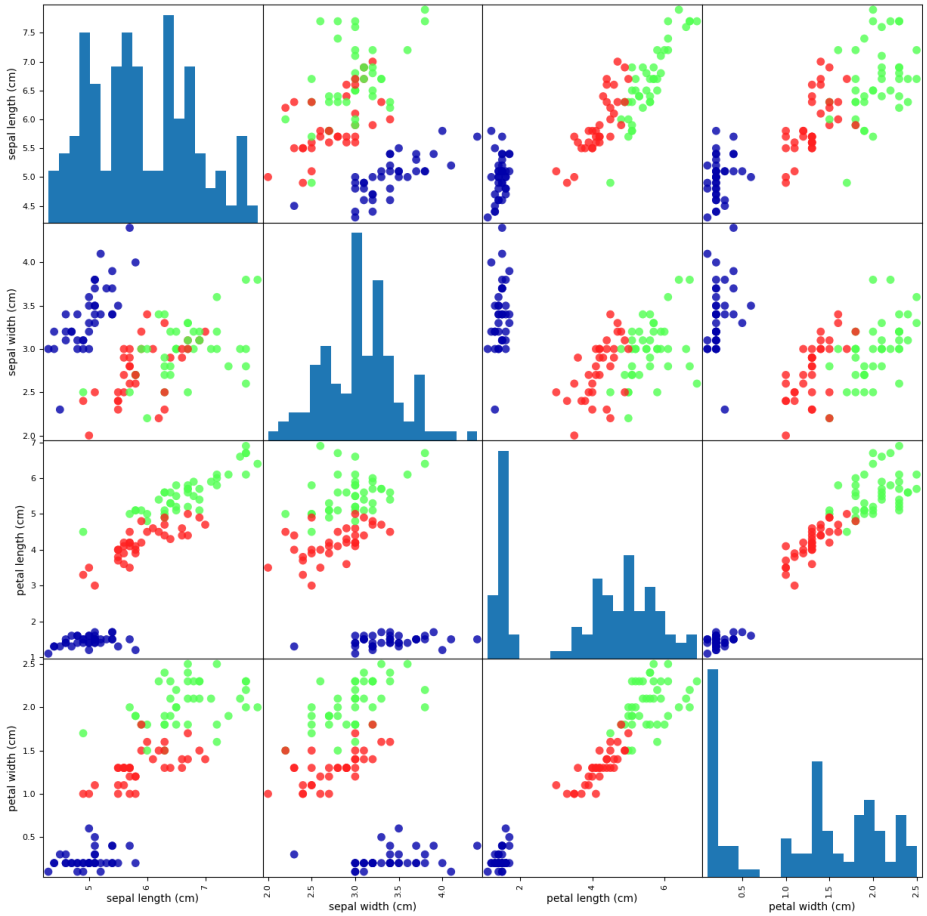

Figure: Pair plot of the Iris dataset, colored by class label

Machine Learning: A First Application: Classifying Iris Species

We will utilize the **k-Nearest Neighbors (KNN) algorithm** to classify the Iris dataset based on its features. KNN is a straightforward yet powerful classification technique. The algorithm's intricacies will be explained in the subsequent slide.

Machine Learning: A First Application: Classifying Iris Species

Algorithm k-Nearest Neighbors

procedure kNN($X, \mathcal{D}, k$)
    $\mathcal{N} \leftarrow$ empty list
    for $\mathbf{x} \in \mathcal{D}$ **do**
        Compute distance $d(\mathbf{x}, X)$
        Add $(\mathbf{x}, d(\mathbf{x}, X))$ to $\mathcal{N}$
    Sort $\mathcal{N}$ by distance
    $\mathcal{N}_k \leftarrow$ first k elements of $\mathcal{N}$
    $\hat{y} \leftarrow$ majority class in $\mathcal{N}_k$
    return $\hat{y}$

Machine Learning: A First Application: Classifying Iris Species

- X: Input instance for prediction
- $\mathcal{D}$: Training dataset with labeled instances
- k: Number of nearest neighbors to consider
- $\mathcal{N}$: List to store neighbors and distances
- $\mathbf{x}$: Instance from training dataset $\mathcal{D}$
- $d(\mathbf{x}, X)$: Distance between $\mathbf{x}$ and X
- $\mathcal{N}_k$: Subset of $\mathcal{N}$ with k nearest neighbors
- $\hat{y}$: Predicted class label for X based on $\mathcal{N}_k$

Machine Learning: A First Application: Classifying Iris Species

Build the k-nearest model:
All machine learning models in scikit-learn are implemented in their own classes, which are called Estimator classes. The k-nearest neighbors classification algorithm is implemented in the KNeighborsClassifier class in the neighbors module.

```
# Import the k-nearest model
from sklearn.neighbors import KNeighborsClassifier
# Create our k-nearest model
knn = KNeighborsClassifier(n_neighbors=1)
```

Machine Learning: A First Application: Classifying Iris Species

Train the k-nearest model on iris dataset :
To build the model on the training set, we call the fit method of the knn object, which takes as arguments the NumPy array X_train containing the training data and the NumPy array y_train of the corresponding training labels:

```
# Build the k-nearest model on the iris training dataset
knn.fit(X_train, y_train)
```

Machine Learning: A First Application: Classifying Iris Species

Making Predictions :
We can now make predictions using this model on new data for which we might not know the correct labels. Imagine we found an iris in the wild with a sepal length of 5 cm, a sepal width of 2.9 cm, a petal length of 1 cm, and a petal width of 0.2 cm.

```
# Put the data into a NumPy array
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))

# Make a prediction
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
iris_dataset['target_names'][prediction]))
```

Machine Learning: A First Application: Classifying Iris Species

Evaluating the Model:
This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species is for each iris in the test set.

Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species). We can measure how well the model works by computing the accuracy, which is the fraction of flowers for which the right species was predicted:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Machine Learning: A First Application: Classifying Iris Species

```
# Make prediction on test dataset
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))

# Compute accuracy
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))

# Combine prediction and accuracy
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Machine Learning: Summary and Outlook

In this chapter, we introduced machine learning applications. We focused on predicting iris species via flower measurements, a supervised task. Using X (features) and y (labels), we split data into train and test sets. We adopted the k-nearest neighbors (KNN) algorithm for classification, achieving 97% accuracy on the test set. This demonstrates the model's ability to predict new data with high confidence.

Machine Learning: Supervised Learning

Definition

Supervised learning is a category of machine learning where the algorithm learns from a labeled dataset, meaning it's provided with input data along with corresponding desired outputs. The goal of supervised learning is to build a model that can accurately map input data to the correct output based on the provided examples.

Machine Learning: Supervised Learning

In supervised machine learning, two primary types of problems stand out: classification and regression.

- **Classification:** Involves categorizing input data into predefined classes. The algorithm learns from labeled data and assigns new data points to the correct categories.

- **Regression:** Focuses on predicting continuous numeric values. Algorithms analyze relationships in training data to make accurate predictions for new data.

Machine Learning: Supervised Learning

In supervised machine learning, achieving the right balance between learning from data and predicting on new data is vital. Concepts like Generalization, Overfitting, and Underfitting play a significant role in this balance.

- **Generalization:**
  - Generalization involves applying learned patterns to new, unseen data.
  - A well-generalized model captures underlying trends without memorizing data.
  - It ensures reliable performance beyond the training set.
- **Overfitting:**
  - Overfitting occurs when a model fits noise and specifics of the training data.
  - Such models perform well on training but poorly on new data.
  - Balancing complexity prevents overfitting and promotes broader applicability.
- **Underfitting:**
  - Underfitting happens when a model is too simplistic to capture data patterns.
  - It leads to poor performance on both training and new data.
  - Addressing underfitting requires increasing model complexity or improving features.

Machine Learning: Supervised Learning

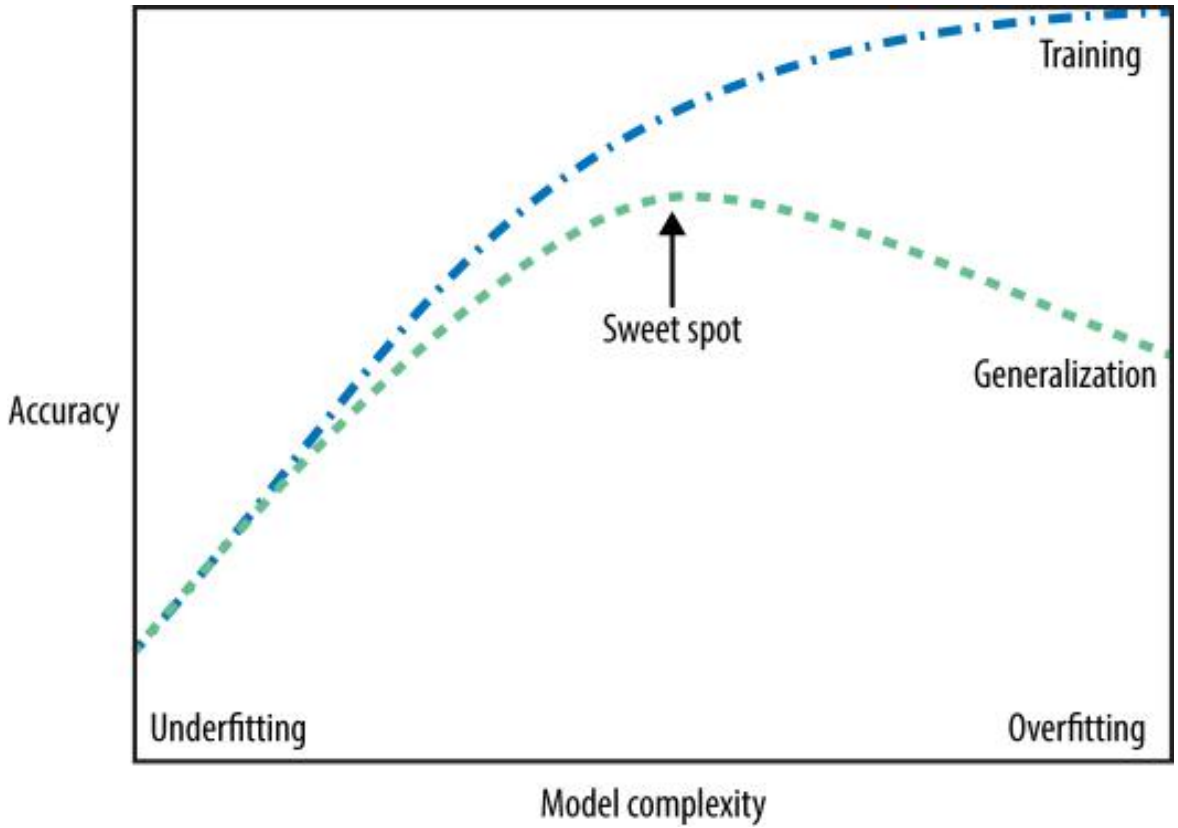

Figure: Trade-off of model complexity against training and test accuracy

Machine Learning: Supervised Learning

Supervised Machine Learning Algorithms:

- k-Nearest Neighbors
  - k-Neighbors classification
  - k-neighbors regression
- Linear Models
  - Linear models for regression
  - Linear models for classification
  - Linear models for multiclass classification
- Naive Bayes Classifiers
- Decision Trees
- Ensembles of Decision Trees
  - Random forests
  - Gradient boosted regression trees (gradient boosting machines)
- Kernelized Support Vector Machines
- Uncertainty Estimates from Classifiers

Supervised Learning: k-Nearest Neighbors

Definition

The k-NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset—its "nearest neighbors".

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

In its simplest version, the k-NN algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for. The prediction is then simply the known output for this training point.

```
import mglearn
# Plot one-nearest-neighbor model on the forge dataset
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

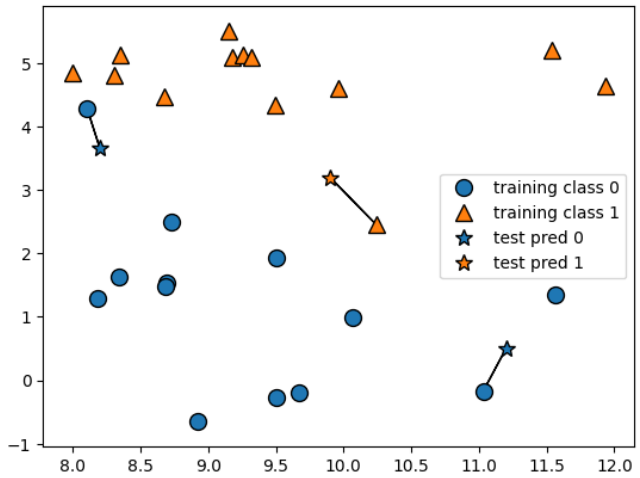

Figure: Predictions made by the one-nearest-neighbor model on the forge dataset

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

Instead of considering only the closest neighbor, we can also consider an arbitrary number, k, of neighbors. This is where the name of the k-nearest neighbors algorithm comes from. When considering more than one neighbor, we use voting to assign a label.

```
import mglearn
# Plot three-nearest-neighbor model on the forge dataset
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

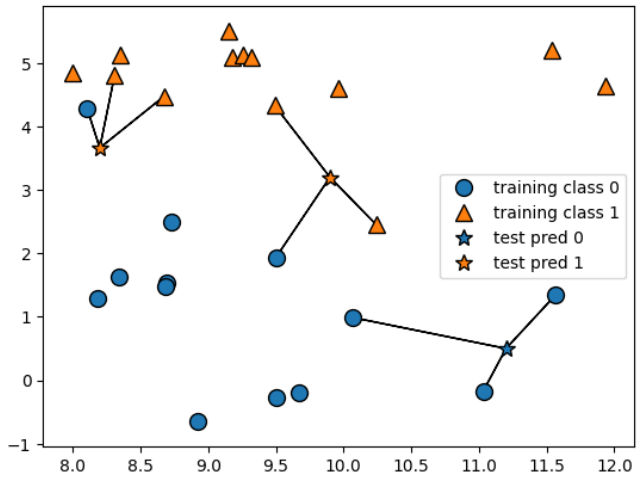

Figure: Predictions made by the three-nearest-neighbors model on the forge dataset

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

Now let's look at how we can apply the k-nearest neighbors algorithm using scikit-learn. First, we split our data into a training and a test set so we can evaluate generalization performance.

```
import mglearn
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
# Make prediction on test data
print("Test set predictions: {}".format(clf.predict(X_test)))
# Evaluate how well the model generalizes
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

Analyzing KNeighborsClassifier:
For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy-plane. We color the plane according to the class that would be assigned to a point in this region. This lets us view the decision boundary, which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

Analyzing KNeighborsClassifier:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))
for n_neighbors, ax in zip([1, 3, 9], axes):
        # the fit method returns the object self, so we can instantiate
        # and fit in one line
        clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
        mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax,
        alpha=.4)
        mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
        ax.set_title("{} neighbor(s)".format(n_neighbors))
        ax.set_xlabel("feature 0")
        ax.set_ylabel("feature 1")
        axes[0].legend(loc=3)
```

Supervised Learning: k-Nearest Neighbors: k-Neighbors classification

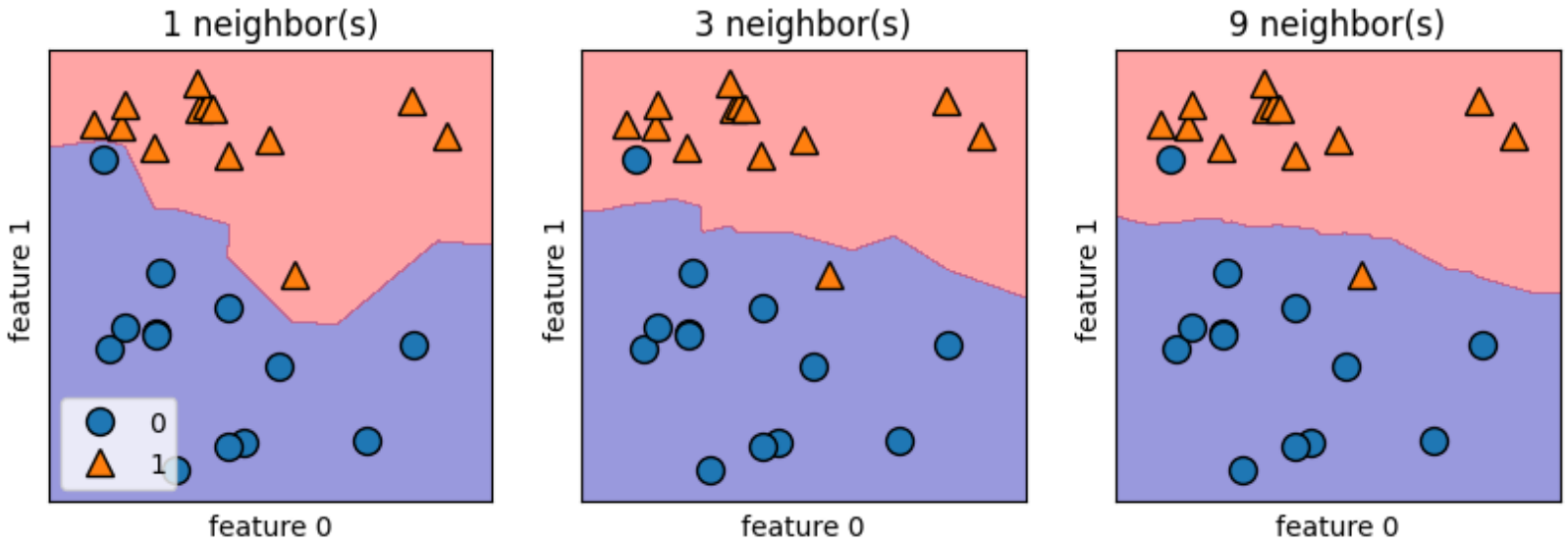

Figure: Decision boundaries created by the nearest neighbors model for different values of n_neighbors

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

There is also a regression variant of the k-nearest neighbors algorithm. Again, let's start by using the single nearest neighbor, this time using the wave dataset. We've added three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor.

```
import mglearn
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

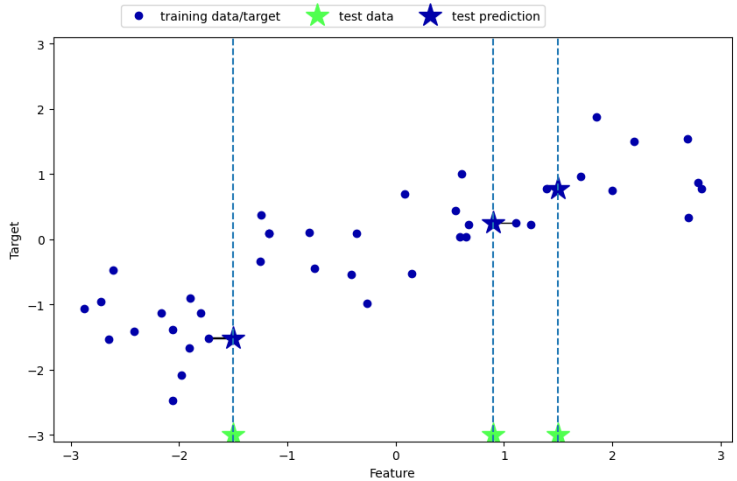

Figure: Predictions made by one-nearest-neighbor regression on the wave dataset

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

Again, we can use more than the single closest neighbor for regression. When using multiple nearest neighbors, the prediction is the average, or mean, of the relevant neighbors.

```
import mglearn
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

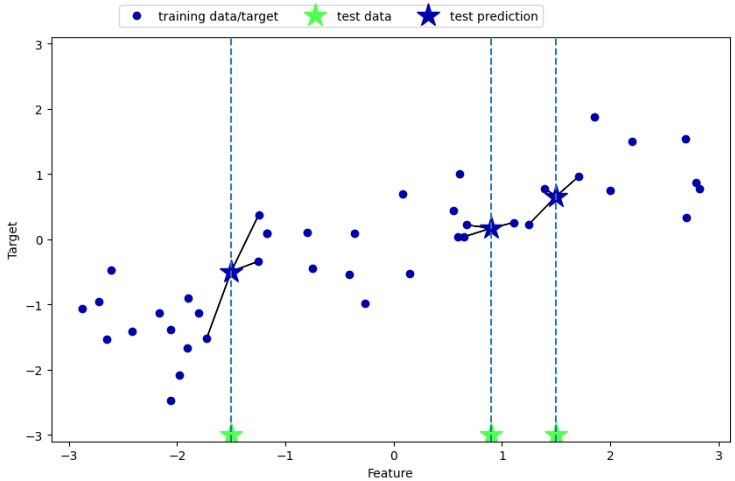

Figure: Predictions made by three-nearest-neighbors regression on the wave dataset

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

The k-nearest neighbors algorithm for regression is implemented in the KNeighbors Regressor class in scikit-learn . It's used similarly to KNeighborsClassifier :

```
import mglearn
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)
# fit the model using the training data and training targets
reg.fit(X_train, y_train)
# Make prediction on the test set
print("Test set predictions:\n{}".format(reg.predict(X_test)))
# Evaluate the model using R^2 score
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

The coefficient of determination (R-squared) is calculated using the formula:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Where:
n is the number of data points.
y_i signifies the actual observed value for the i th data point.
$\hat{y}_i$ is the predicted value for the i th data point by the model.
$\bar{y}$ stands for the mean of the observed values.

For our one-dimensional dataset, we can see what the predictions look like for all possible feature values. To do this, we create a test dataset consisting of many points on the line:

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
        # make predictions using 1, 3, or 9 neighbors
        reg = KNeighborsRegressor(n_neighbors=n_neighbors)
        reg.fit(X_train, y_train)
        ax.plot(line, reg.predict(line))
        ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
        ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)
        ax.set_title(
        "{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
        n_neighbors, reg.score(X_train, y_train),
        reg.score(X_test, y_test)))
        ax.set_xlabel("Feature")
        ax.set_ylabel("Target")
        axes[0].legend(["Model predictions", "Training data/target",
        "Test data/target"], loc="best")
```

Supervised Learning: k-Nearest Neighbors: k-neighbors regression

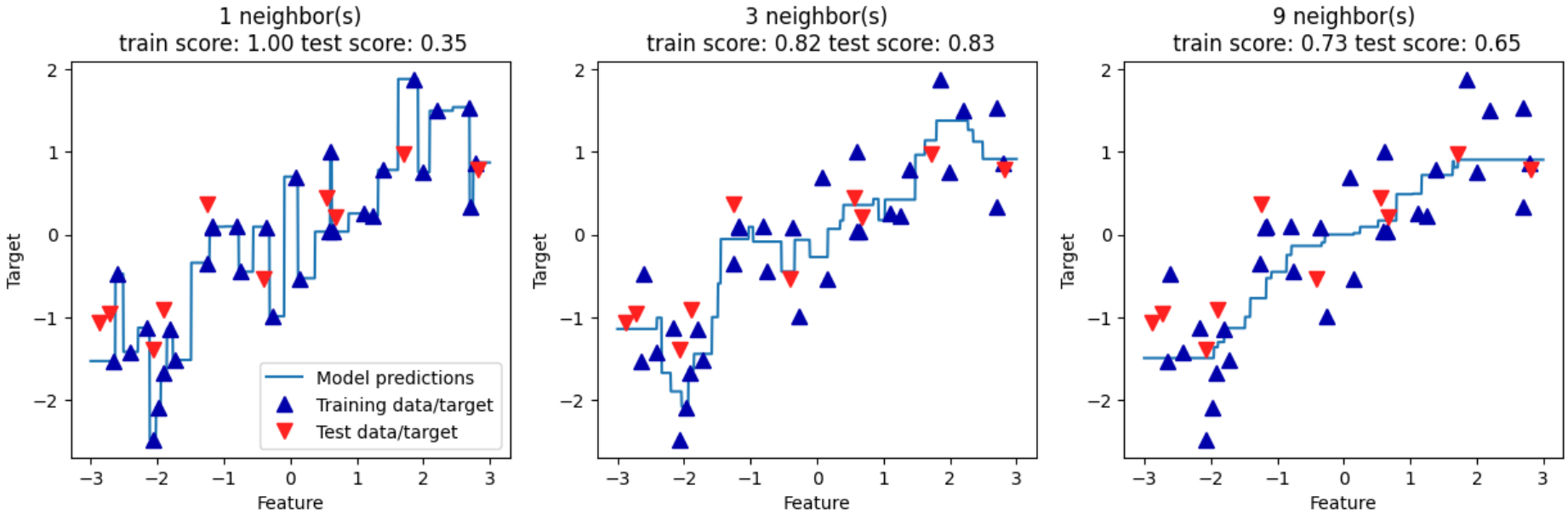

Figure: Comparing predictions made by nearest neighbors regression for different values of n_neighbors

Supervised Learning: k-Nearest Neighbors

Strengths, weaknesses, and parameters:

- **Strengths:**
  - The KNeighbors classifier is easy to understand and implement.
  - It often provides reasonable performance with minimal tuning.
  - It serves as a good baseline model before exploring more complex algorithms.
- **Weaknesses:**
  - Prediction can be slow, especially with large training datasets.
  - It struggles with high-dimensional datasets and sparse features.
  - Not often used in practice due to its limitations and efficiency issues.
- **Parameters:**
  - Important parameters include the number of neighbors and the distance metric.
  - A smaller number of neighbors, like three or five, is often effective.
  - The default Euclidean distance works well for many scenarios.

Supervised Learning: Linear Models

Definition

Linear models are a class of models that are widely used in practice and have been studied extensively in the last few decades, with roots going back over a hundred years. Linear models make a prediction using a linear function of the input features, which we will explain shortly.

Supervised Learning: Linear Models: Regression

For regression, the general prediction formula for a linear model looks as follows:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b$$

Here, $x[0]$ to $x[p]$ denotes the features (in this example, the number of features is p) of a single data point, w and b are parameters of the model that are learned, and $\hat{y}$ is the prediction the model makes.

For a dataset with a single feature, this is:

$$\hat{y} = w[0] * x[0] + b$$

Here, $w[0]$ is the slope and b is the y-axis offset. For more features, w contains the slopes along each feature axis. Alternatively, you can think of the predicted response as being a weighted sum of the input features, with weights (which can be negative) given by the entries of w.

Supervised Learning: Linear Models: Regression

Trying to learn the parameters $w[0]$ and b on our one-dimensional wave dataset might lead to the following line:

```
import mglearn
mglearn.plots.plot_linear_regression_wave()
```

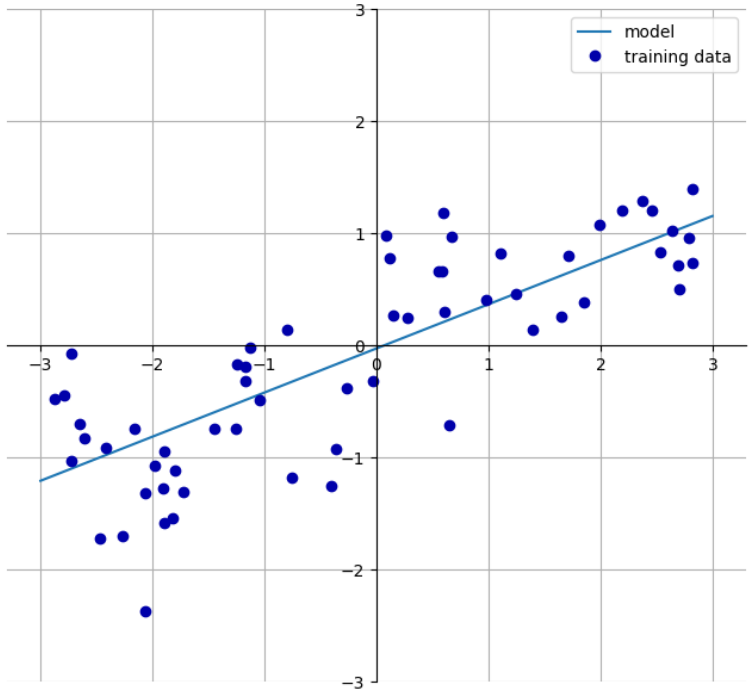

Figure: Predictions of a linear model on the wave dataset: $w[0]$: 0.393906 b : −0.031804

Supervised Learning: Linear Models: Regression

Linear regression, or ordinary least squares (OLS), is the simplest and most classic linear method for regression. Linear regression finds the parameters w and b that minimize the mean squared error between predictions and the true regression targets, y, on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values. Linear regression has no parameters, which is a benefit, but it also has no way to control model complexity.

Supervised Learning: Linear Models: Regression

```
import mglearn
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
lr = LinearRegression().fit(X_train, y_train)
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

The "slope" parameters (w), also called weights or coefficients, are stored in the *coef_attribute*, while the offset or intercept (b) is stored in the *intercept_attribute*.

Exercise:
Implement Ridge Regression and Lasso Regression on Boston Housing dataset as alternative methods to enhance the coefficient of determination (R^2) by tuning the regularization parameter (α).

```
import mglearn
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
# Load Boston Housing dataset
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# Complete the code
...
```

Supervised Learning: Linear Models: Classification

Linear models are also extensively used for classification. Let's look at binary classification first. In this case, a prediction is made using the following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function is smaller than zero, we predict the class –1; if it is larger than zero, we predict the class +1. This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients (w) and the intercept (b).

Supervised Learning: Linear Models: Classification

The two most common linear classification algorithms are logistic regression, and linear support vector machines (linear SVMs). Despite its name, LogisticRegression is a classification algorithm and not a regression algorithm, and it should not be confused with LinearRegression.

Supervised Learning: Linear Models: Classification

We can apply the LogisticRegression and LinearSVC models to the forge dataset, and visualize the decision boundary as found by the linear models:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
X, y = mglearn.datasets.make_forge()
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
    axes[0].legend()
```

Supervised Learning: Linear Models: Classification

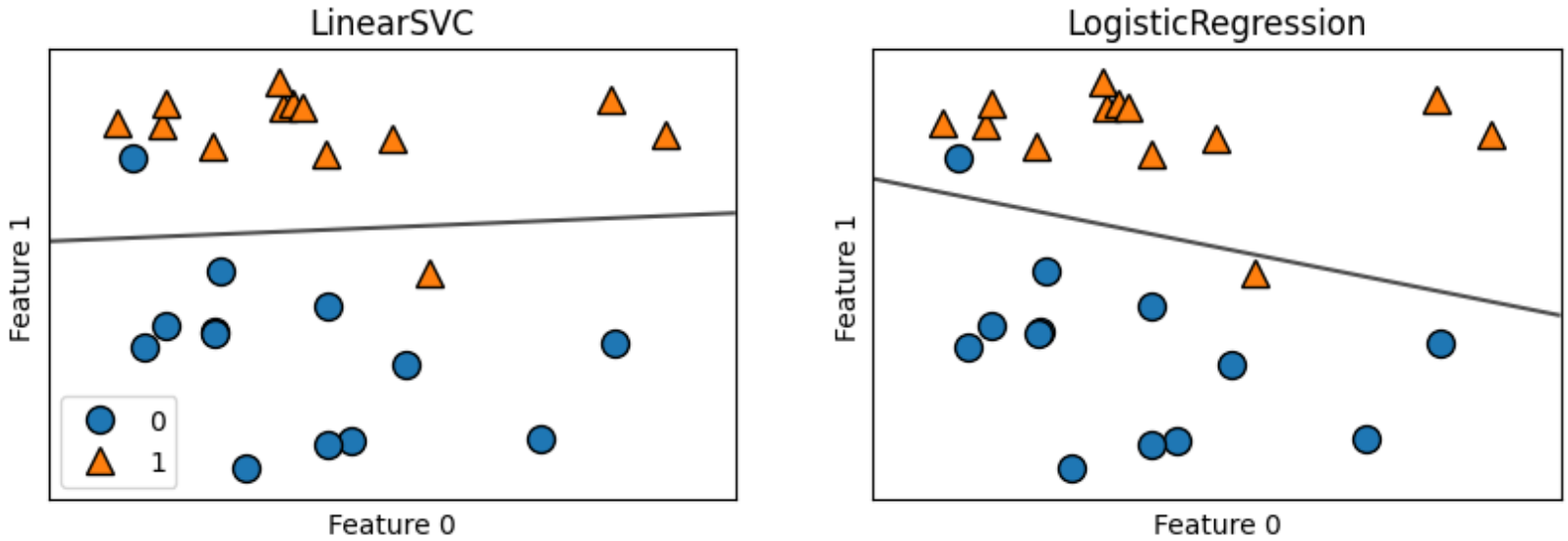

Figure: Decision boundaries of a linear SVM and logistic regression on the forge dataset with the default parameters

The two models come up with similar decision boundaries. Note that both misclassify two of the points. By default, both models apply an L2 regularization, in the same way that Ridge does for regression.

For LogisticRegression and LinearSVC the trade-off parameter that determines the strength of the regularization is called C , and higher values of C correspond to less regularization. In other words, when you use a high value for the parameter C , LogisticRegression and LinearSVC try to fit the training set as best as possible, while with low values of the parameter C , the models put more emphasis on finding a coefficient vector (w) that is close to zero.

Supervised Learning: Linear Models: Classification

The objective of LinearSVC is to find the optimal $\mathbf{w}$ and b that minimize the hinge loss, subject to a regularization term:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$$

where $\|\mathbf{w}\|^2$ is the L2 norm of the weight vector $\mathbf{w}$, C is the regularization parameter, $\mathbf{x}_i$ is the feature vector of the i-th sample, and y_i is its label.

Supervised Learning: Linear Models: Classification

```
import mglearn
mglearn.plots.plot_linear_svc_regularization()
```

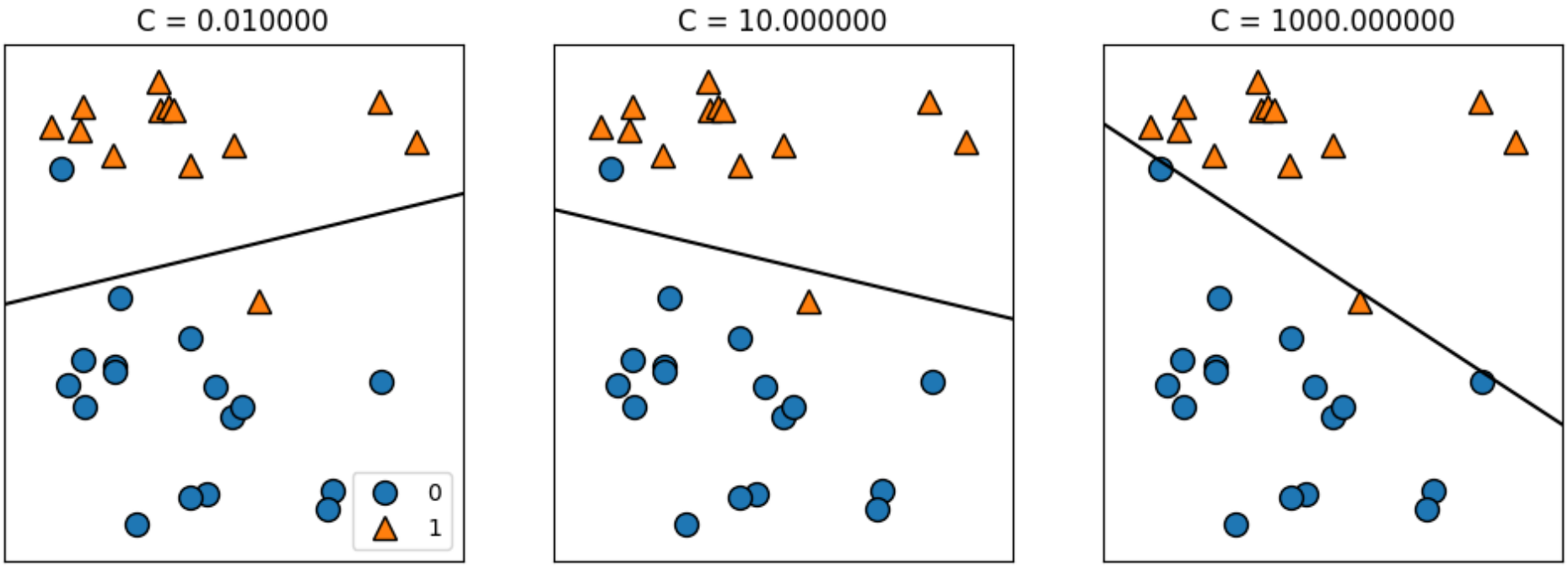

Figure: Decision boundaries of a linear SVM on the forge dataset for different values of C

Exercise:
Explore the impact of different values of the regularization parameter C on Logistic Regression performance in Breast Cancer dataset. Specifically, you will use three different values of: $C = 0.01, C = 1, C = 100$.

```
# Example with default regularization
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

Supervised Learning: Linear Models: multiclass classification

Many linear classification models are for binary classification only, and don't extend naturally to the multiclass case (with the exception of logistic regression). A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the one-vs.-rest approach. In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes. To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class "wins", and this class label is returned as the prediction.

Supervised Learning: Linear Models: multiclass classification

Let's apply the one-vs.-rest method to a simple three-class classification dataset. We use a two-dimensional dataset, where each class is given by data sampled from a Gaussian distribution:

```
import mglearn
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

Supervised Learning: Linear Models: multiclass classification

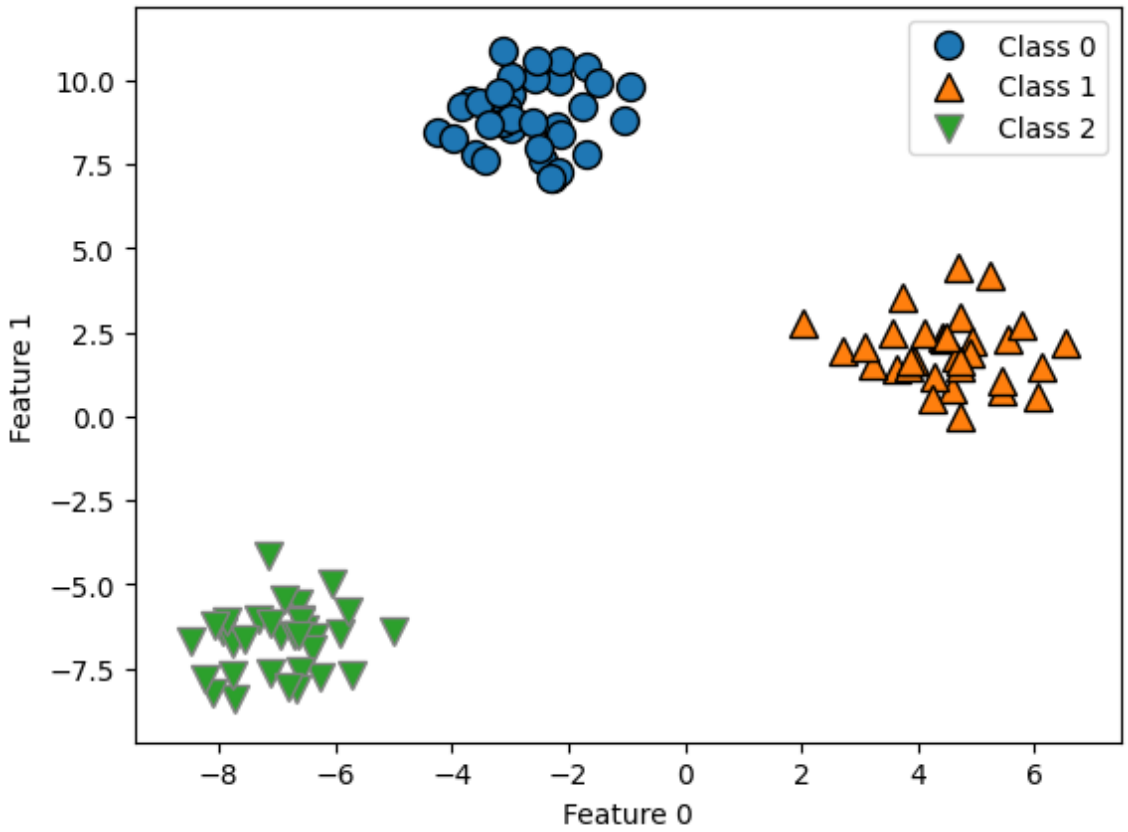

Figure: Two-dimensional toy dataset containing three classes

Supervised Learning: Linear Models: multiclass classification

Now, we train a LinearSVC classifier on the dataset:

```
from sklearn.svm import LinearSVC
import matplotlib.pyplot as plt
import mglearn
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
```

Supervised Learning: Linear Models: multiclass classification

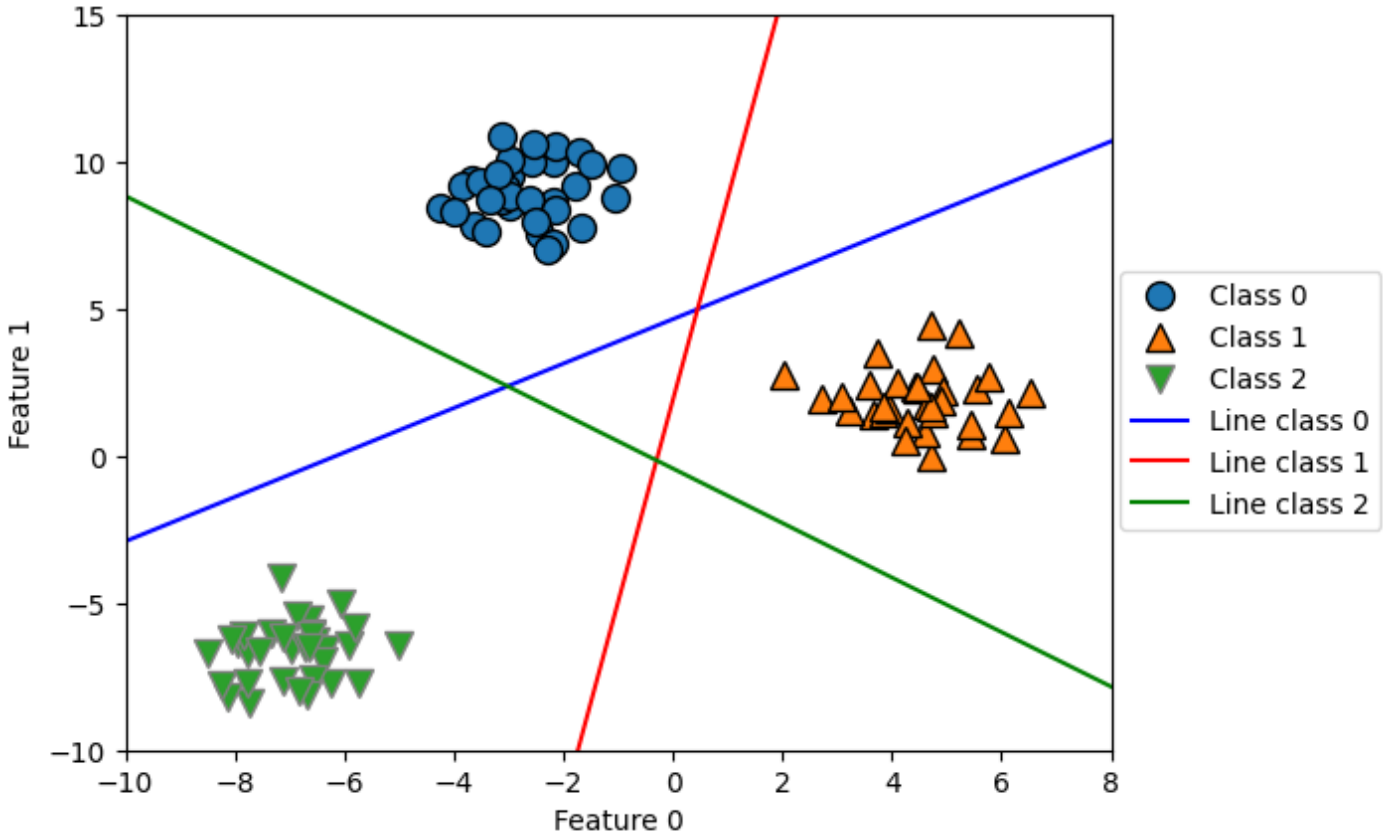

Figure: Decision boundaries learned by the three one-vs.-rest classifiers

Supervised Learning: Linear Models: multiclass classification

The following example shows the predictions for all regions of the 2D space:

```
from sklearn.svm import LinearSVC
import matplotlib.pyplot as plt
import mglearn
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
'Line class 2'], loc=(1.01, 0.3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Supervised Learning: Linear Models: multiclass classification

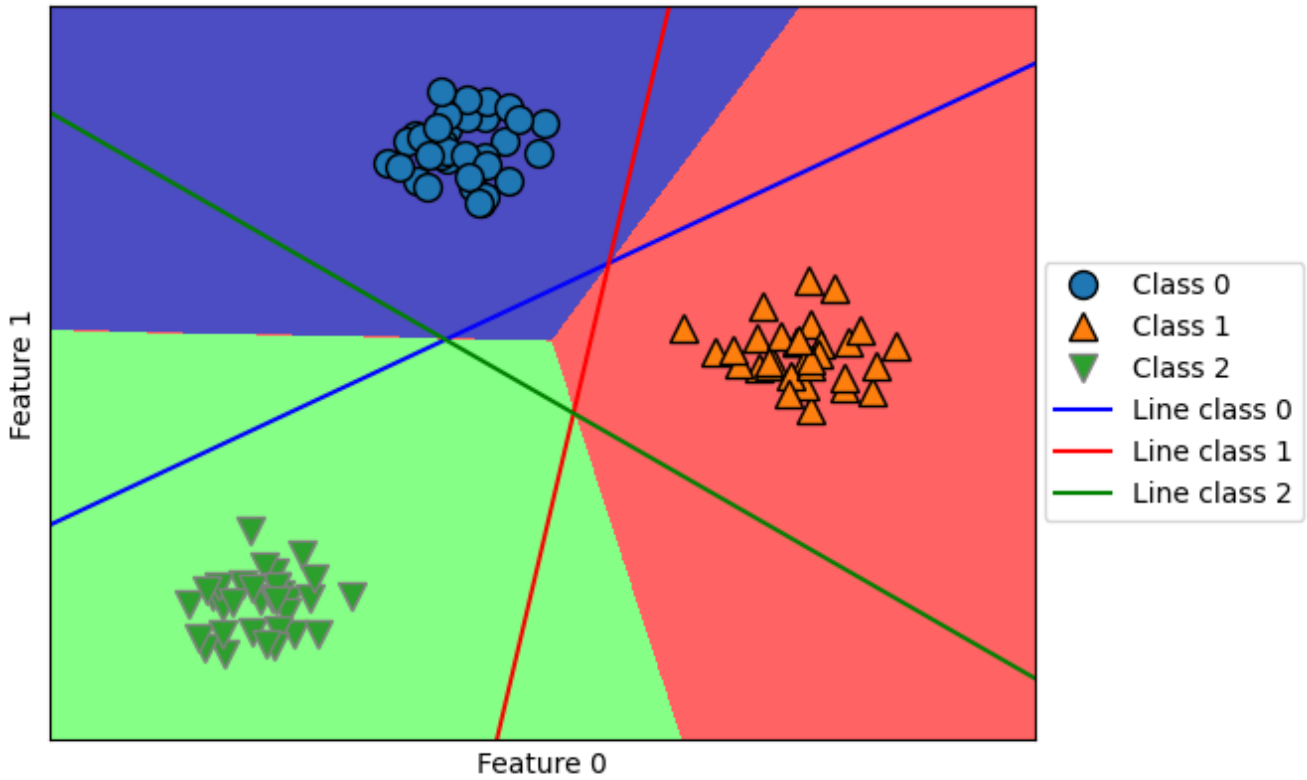

Figure: Multiclass decision boundaries derived from the three one-vs.-rest classifiers

Supervised Learning: Linear Models

Strengths, weaknesses, and parameters:

Strengths:

- Efficient training and prediction.
- Suitable for large datasets and sparse data.
- Scalable options like `solver='sag'`.
- Clear interpretation of predictions.

Weaknesses:

- Interpretation challenges with correlated features.

Parameters:

- Main parameter: Regularization (α in regression, C in LinearSVC and LogisticRegression).
- Larger α or smaller C: Simpler models.
- Tuning α and C important on a logarithmic scale.
- Choose between L1 (sparse important features) and L2 regularization (default).
- L1 aids model interpretability by focusing on key features.

Supervised Learning: Naive Bayes Classifiers

Introduction

Naive Bayes is a probabilistic classification algorithm based on Bayes' Theorem. It assumes that features are conditionally independent given the class label. This simplifying assumption allows for efficient training and prediction. The algorithm is widely used for text classification, spam detection, and more.

Supervised Learning: Naive Bayes Classifiers: Training

Given a dataset with n instances and m features, and C distinct classes $C_1, C_2, \ldots, C_c$, we want to calculate the probability of each class $P(C_i)$ and the conditional probability $P(x_j|C_i)$ for each feature x_j given class C_i.

1. **Calculate Class Priors:**
   - Calculate the total number of instances n and the count of instances n_i belonging to each class C_i.
   - Calculate the prior probability $P(C_i) = \frac{n_i}{n}$
2. **Calculate Conditional Probabilities (with Laplace Smoothing):**
   - For each feature x_j, calculate the number of instances n_{ij} where feature x_j occurs in class C_i.
   - Calculate the conditional probability $P(x_j|C_i) = \frac{n_{ij}+\alpha}{n_i+\alpha \cdot \text{Number of unique feature-values}}$
   - To avoid numerical issues, calculate the logarithm of probabilities: $\log(P(C_i))$ and $\log(P(x_j|C_i))$

Supervised Learning: Naive Bayes Classifiers: Inference

Given a new instance with feature values $x_1, x_2, ..., x_m$, we want to predict its class label C_i.

1. **Calculate Log Posterior Probabilities:**
   - Calculate the log posterior probability for each class C_i using Bayes' Theorem:

$$\log(\mathbb{P}(C_i|x_1, x_2, ..., x_m) = \log(\mathbb{P}(C_i)) + \sum_{j=1}^{m} \mathbb{P}(x_j|C_i)$$

2. **Prediction:**
   - Choose the class C_i that maximizes the log posterior probability:

$$C_{predicted} = argmax_{C_i}(\log(\mathbb{P}(C_i|x_1, x_2, ..., x_m)))$$

Supervised Learning: Naive Bayes Classifiers Implementation

There are three kinds of naive Bayes classifiers implemented in scikit-learn : GaussianNB , BernoulliNB , and MultinomialNB . GaussianNB can be applied to any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). BernoulliNB and MultinomialNB are mostly used in text data classification.

Supervised Learning: Naive Bayes Classifiers

Strengths, weaknesses, and parameters:

- **Strengths:**
  - **Efficiency:** Naive Bayes models are fast to train and predict, making them suitable for large datasets.
  - **Simplicity:** The training procedure is straightforward and easy to understand.
  - **Baseline Model:** Naive Bayes serves as a good baseline model, especially for quick initial assessments.
  - **High-Dimensional Data:** These models perform well on high-dimensional sparse data, such as text data.
- **Weaknesses:**
  - **Independence Assumption:** The "naive" assumption of feature independence may not hold in some cases, leading to potential inaccuracies.
  - **Limited Expressiveness:** Due to the independence assumption, complex relationships between features are not captured.
  - **Accuracy Trade-off:** While often accurate, Naive Bayes might not match the performance of more complex models in certain scenarios.
- **Parameters:**
  - **Alpha:** The smoothing parameter ($0 \leq \alpha \leq 1$) controls model complexity. Higher alpha leads to smoother statistics and simpler models. Tuning alpha can enhance accuracy.

Supervised Learning: Decision Trees

Definition

Decision trees are common models for classification and regression tasks. They use a series of if/else questions to make decisions effectively, similar to a process of elimination. These questions help in distinguishing between different classes or making predictions.

```
import mglearn
mglearn.plots.plot_animal_tree()
```

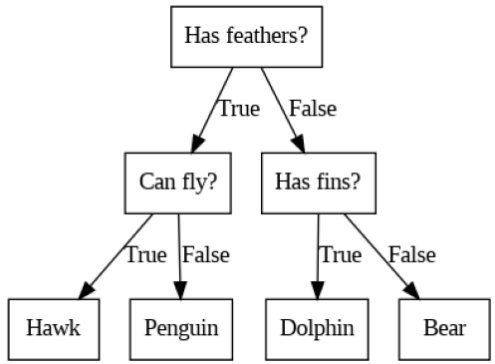

Figure: A decision tree to distinguish among several animals

Supervised Learning: Decision Trees

Building a decision tree involves finding the optimal sequence of if/else questions to efficiently arrive at the correct answer. These questions, known as tests, are used to distinguish between classes or make predictions. In real-world datasets, unlike the animal example, questions often pertain to continuous features such as "Is feature i larger than value a?" We'll walk through this process using the two_moons dataset, consisting of two half-moon shapes with 75 data points in each class.

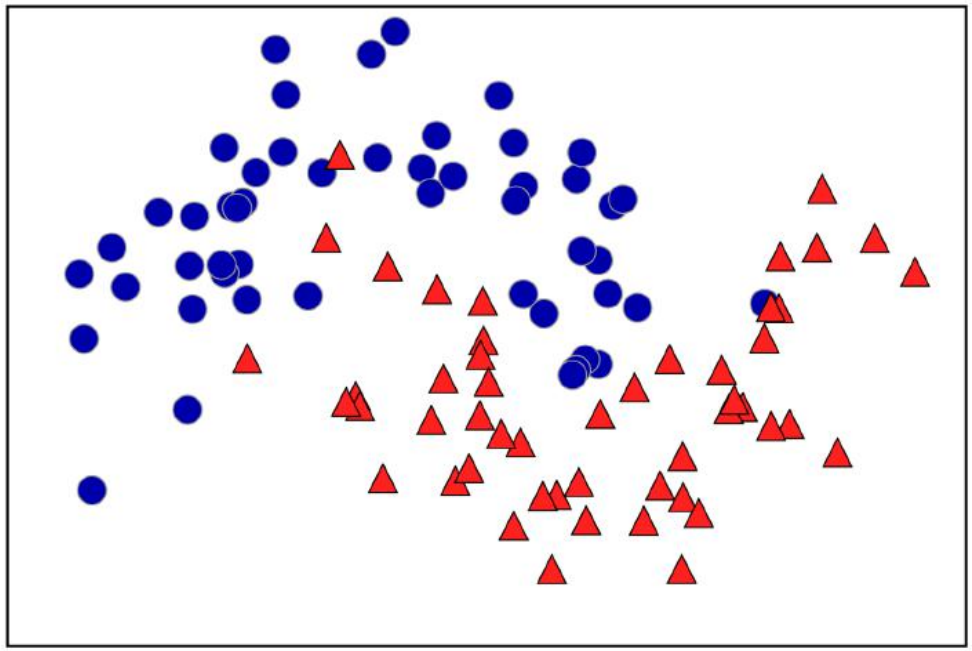

Figure: Two-moons dataset on which the decision tree will be built

Supervised Learning: Decision Trees

The decision tree-building process involves finding the most informative test at each step to separate data points effectively. It begins with an initial test that partitions the dataset based on a specific feature and threshold. The top node represents the entire dataset, and depending on the test outcome, points are assigned to either the left or right child nodes. The goal is to refine these partitions to minimize misclassifications.

The algorithm continues by searching for the most informative tests within each child node, iteratively improving the separation of data points. This process repeats until a stopping criterion is met or until the tree reaches a predefined depth. The result is a hierarchical structure of if/else questions, forming a decision tree that can make predictions based on input features.

Supervised Learning: Decision Trees

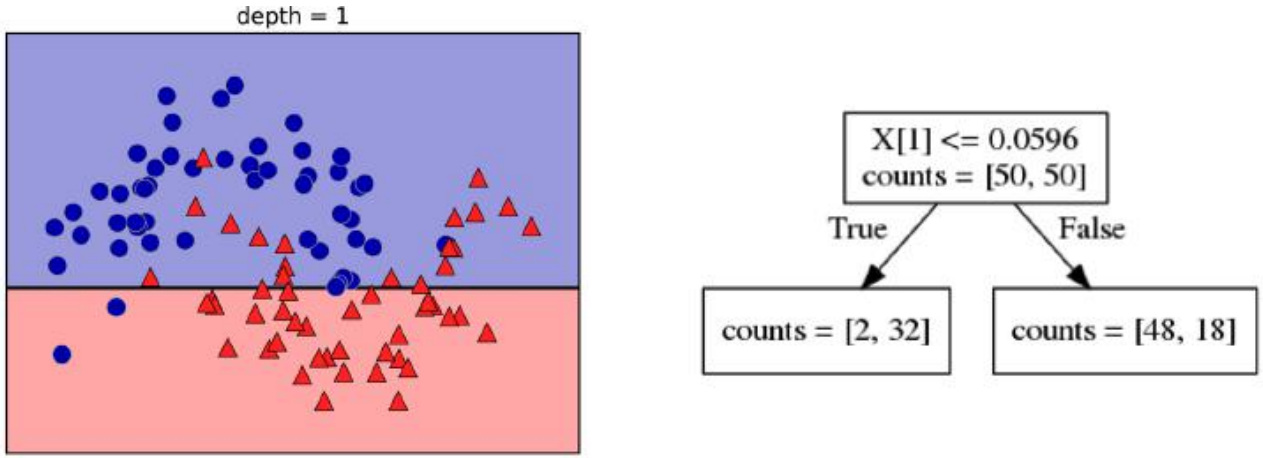

Figure: Decision boundary of tree with depth 1 (left) and corresponding tree (right)

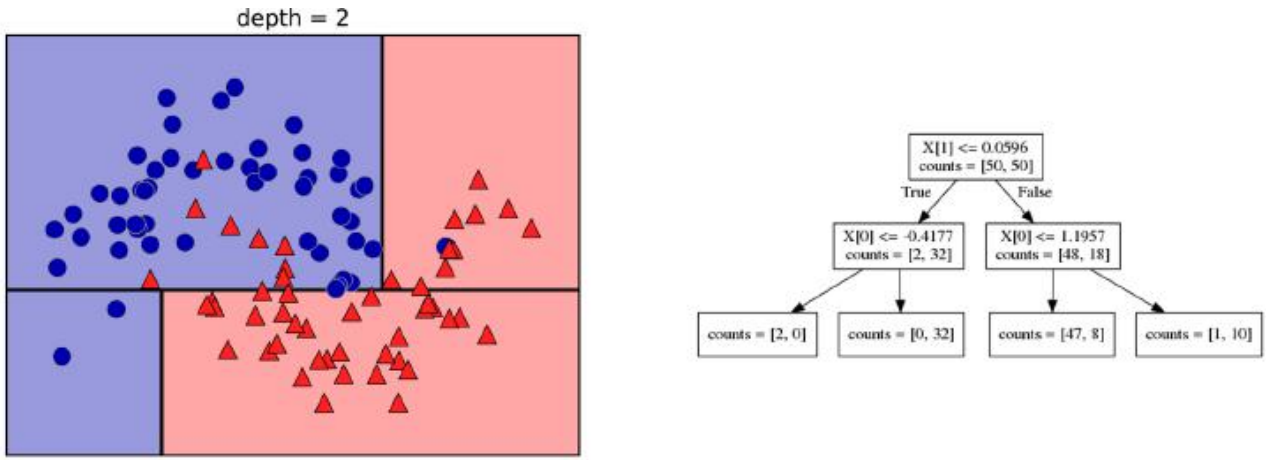

Figure: Decision boundary of tree with depth 2 (left) and corresponding decision tree (right)

Supervised Learning: Decision Trees

The decision tree construction process is recursive, forming a binary tree structure with each node representing a test. These tests split the data along specific axes, resulting in a hierarchical partition. Since each test focuses on a single feature, the partition boundaries are always axis-parallel.
The recursive partitioning continues until each leaf node in the decision tree contains only one unique target value, making it pure. This means that all data points within a pure leaf belong to the same class or have the same regression value.

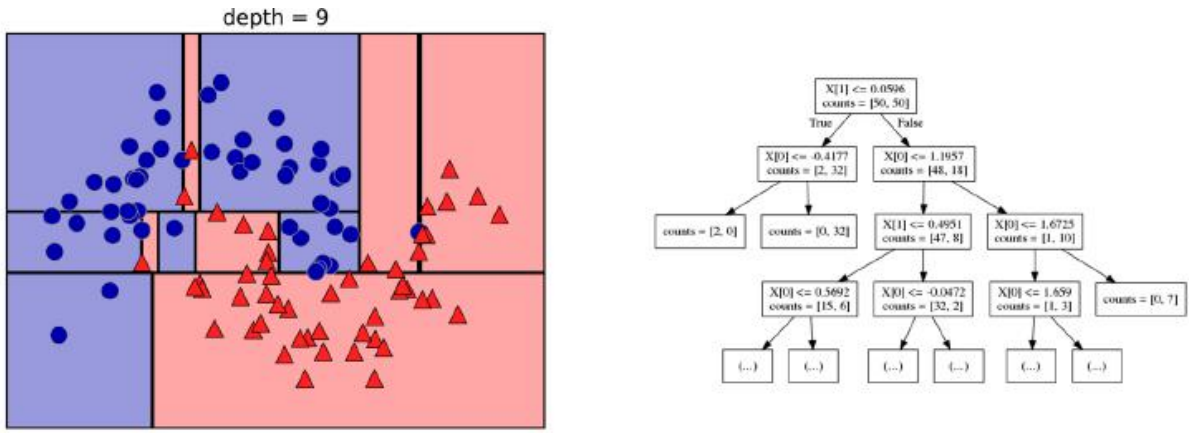

Figure: Decision boundary of tree with depth 9 (left) and part of the corresponding tree (right); the full tree is quite large and hard to visualize

Supervised Learning: Decision Trees

A prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target (or the single target in the case of pure leaves) in that region. The region can be found by traversing the tree from the root and going left or right, depending on whether the test is fulfilled or not.

It is also possible to use trees for regression tasks, using exactly the same technique. To make a prediction, we traverse the tree based on the tests in each node and find the leaf the new data point falls into. The output for this data point is the mean target of the training points in this leaf.

Controlling complexity of decision trees:
Building decision trees by expanding them until all leaves are pure, meaning they perfectly match the training data, often results in overly complex and highly overfit models. Overfitting is evident when decision boundaries incorrectly classify data points and focus too much on outliers, deviating from the expected boundary shapes.

To address overfitting, two common strategies are employed: pre-pruning and post-pruning. Pre-pruning involves setting constraints before constructing the tree, such as limiting the tree's maximum depth, the number of leaves, or the minimum number of data points required to split a node. Post-pruning, on the other hand, builds the tree first and then removes or combines nodes that provide little information. These techniques help create more generalizable decision trees.

Model with pure leaves

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Model limiting the depth of the tree decreases overfitting

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Supervised Learning: Decision Trees

We can visualize the tree using the export_graphviz function from the tree module. This writes a file in the .dot file format, which is a text file format for storing graphs. We set an option to color the nodes to reflect the majority class in each node and pass the class and features names so the tree can be properly labeled:

```
from sklearn.tree import export_graphviz
import graphviz

export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
feature_names=cancer.feature_names, impurity=True, filled=True)
with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Supervised Learning: Decision Trees

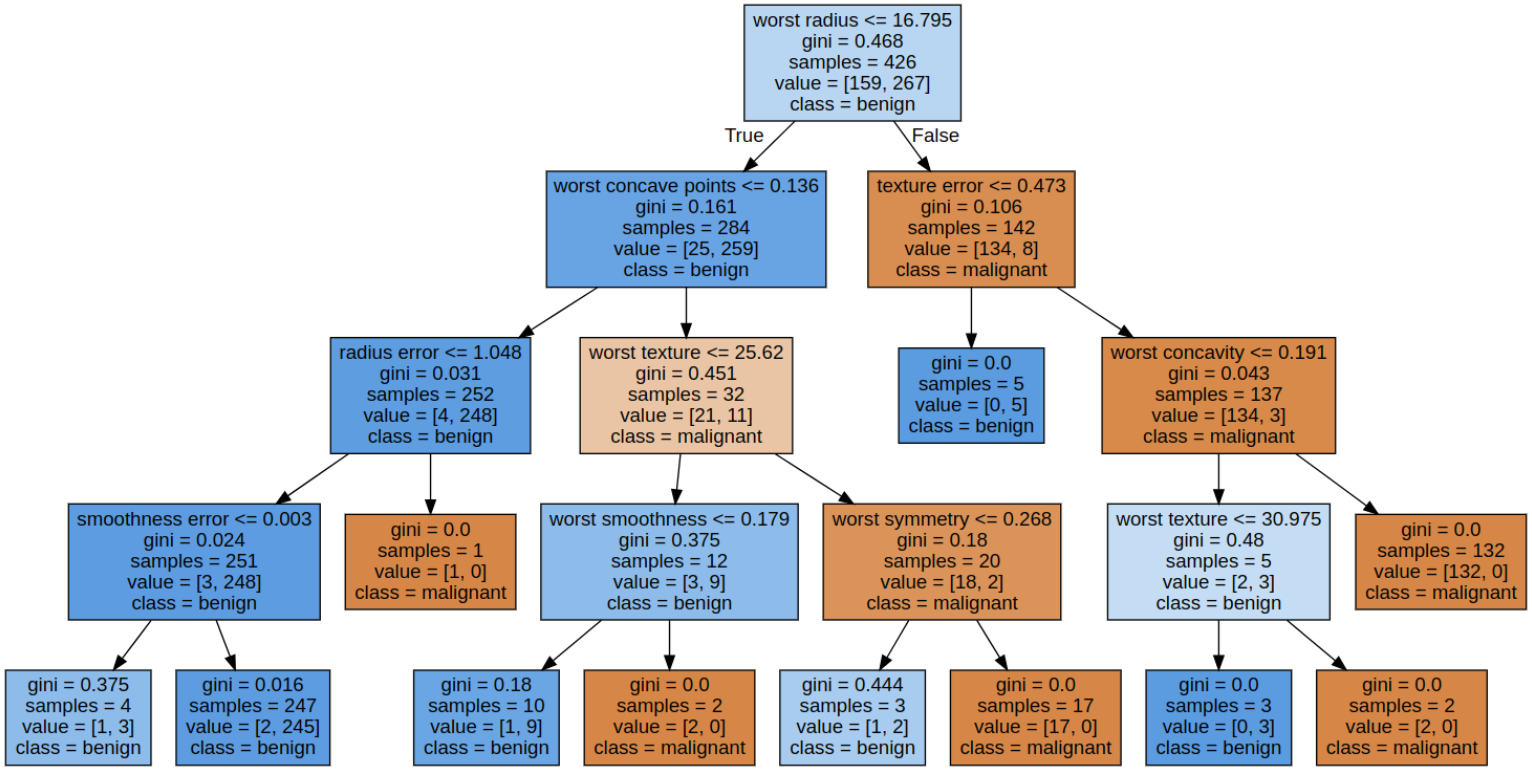

Figure: Visualization of the decision tree built on the Breast Cancer dataset

Feature importance in trees:
Instead of looking at the whole tree, which can be taxing, there are some useful properties that we can derive to summarize the workings of the tree. The most commonly used summary is feature importance, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means "not used at all" and 1 means "perfectly predicts the target." The feature importances always sum to 1:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

Supervised Learning: Decision Trees

We can visualize the feature importances in a way that is similar to the way we visualize the coefficients in the linear model:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
plot_feature_importances_cancer(tree)
```

Supervised Learning: Decision Trees

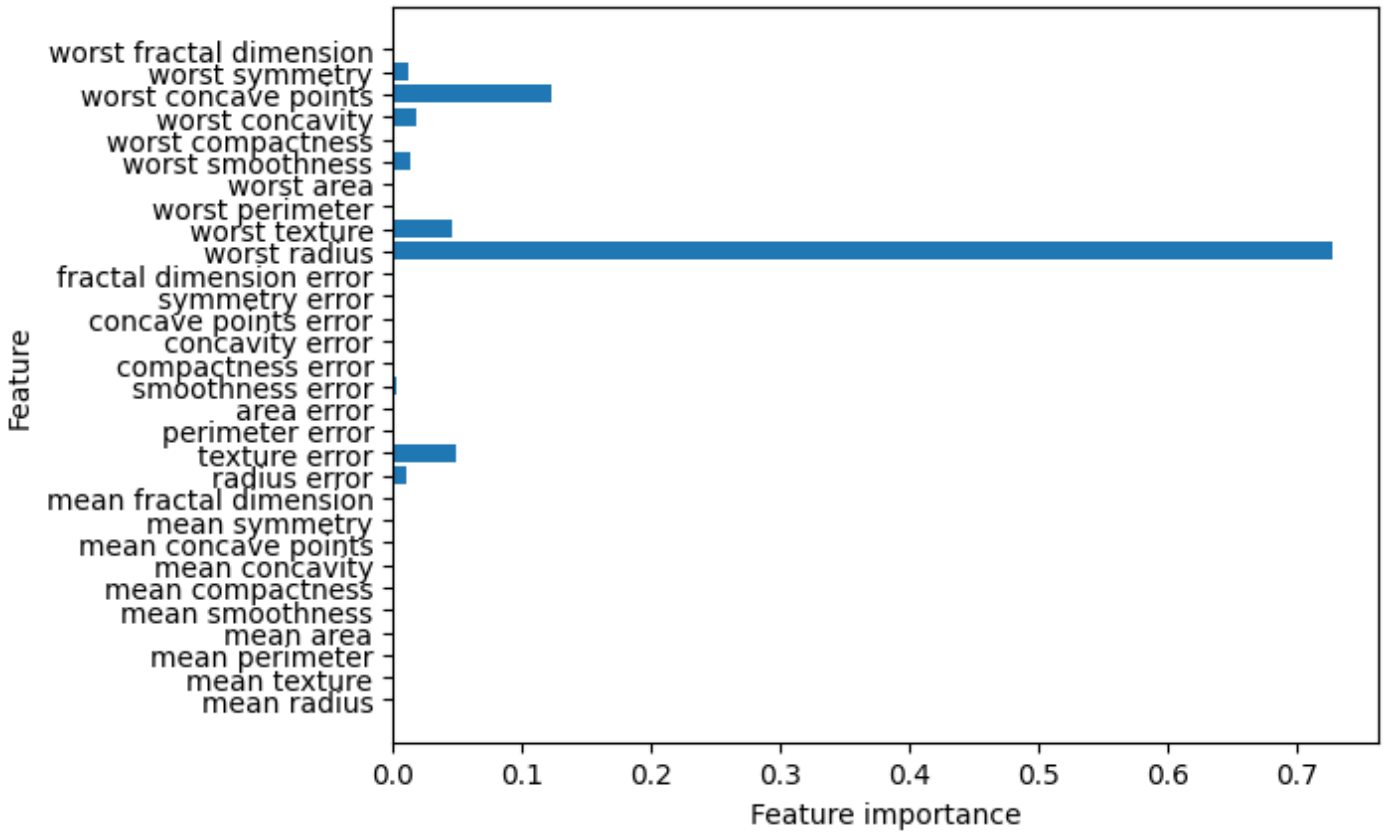

Figure: Feature importances computed from a decision tree learned on the Breast Cancer dataset

Supervised Learning: Decision Trees

Strengths, weaknesses, and parameters:

- **Strengths:**
  - Decision trees are highly interpretable and can be easily visualized and understood, especially for smaller trees.
  - Decision tree algorithms are invariant to data scaling, eliminating the need for preprocessing such as feature normalization or standardization.
  - Effective when dealing with features on different scales or a combination of binary and continuous features.
- **Weaknesses:**
  - Decision trees tend to overfit the training data, leading to poor generalization performance, even with pre-pruning.
- **Parameters:**
  - Pre-pruning parameters can be used to control model complexity:
    - `max_depth`: Limits the maximum depth of the tree.
    - `max_leaf_nodes`: Limits the maximum number of leaf nodes.
    - `min_samples_leaf`: Requires a minimum number of samples in a leaf node to continue splitting.

Supervised Learning: Ensembles of Decision Trees

Definition

Ensembles are methods that combine multiple machine learning models to create more powerful models. There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building blocks: random forests and gradient boos- ted decision trees.

Supervised Learning: Ensembles of Decision Trees: Random Forest

Definition

Random forests are an ensemble learning method that combines multiple decision trees to improve model accuracy and reduce overfitting. They utilize bootstrapping (sample selection) and random feature selection to create diverse trees, and the final prediction in classification tasks is based on a majority vote from individual trees. In regression, predictions are averaged. Random forests are known for their robustness, parallelizability, and feature importance estimation, making them a popular choice for various machine learning tasks.

Supervised Learning: Ensembles of Decision Trees: Random Forest

Building random forests:
To illustrate the process, consider a dataset with samples ['a', 'b', 'c', 'd']. A bootstrap sample may look like ['b', 'd', 'd', 'c'], and another could be ['d', 'a', 'd', 'a']. Decision trees in the random forest are built using a slightly modified algorithm. Instead of considering all features at each node, the algorithm randomly selects a subset of features, controlled by max_features, for making splits. This randomness ensures that each tree in the forest is different.

A critical parameter is max_features. High max_features makes trees more similar, fitting the data using distinctive features, while low max_features makes trees diverse, potentially requiring greater depth to fit the data.

For predictions, regression results are averaged across trees. In classification, a "soft voting" strategy is used, where each tree provides class probabilities. These probabilities are averaged, and the class with the highest probability is the final prediction. This ensemble technique reduces overfitting and enhances model robustness and predictive power.

Supervised Learning: Ensembles of Decision Trees: Random Forest

Analyzing random forests:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
random_state=42)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
# Visualization
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

Supervised Learning: Ensembles of Decision Trees: Random Forest

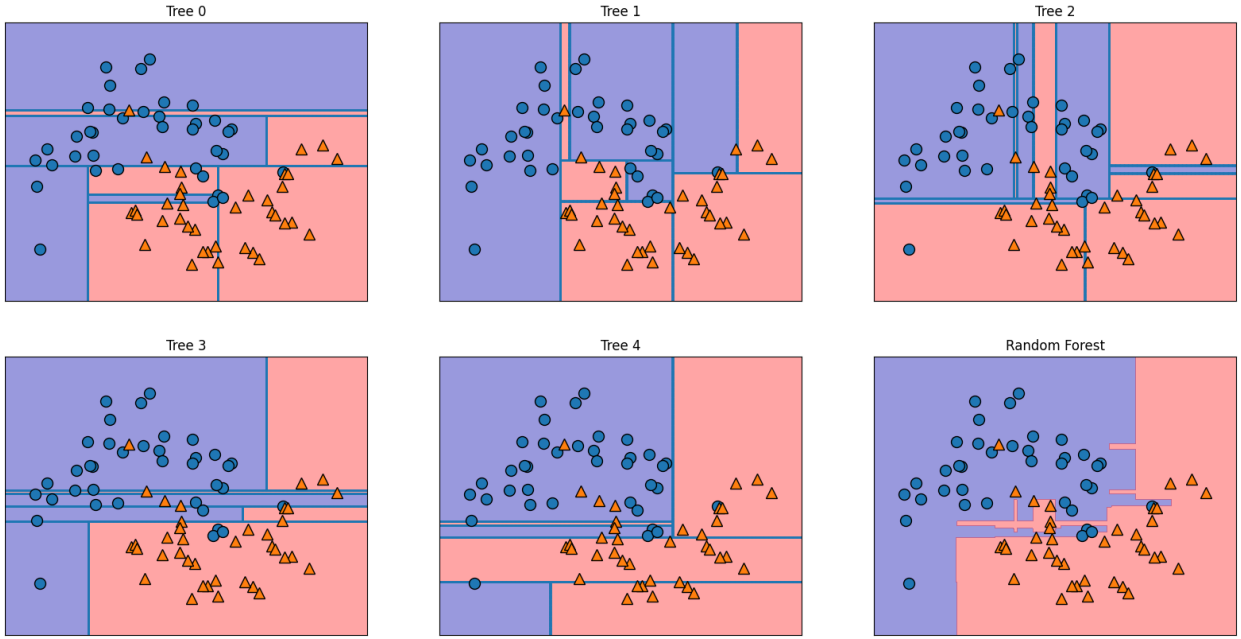

Figure: Decision boundaries found by five randomized decision trees and the decision boundary obtained by averaging their predicted probabilities

Supervised Learning: Ensembles of Decision Trees: Random Forest

Analyzing random forests: As another example, let's apply a random forest consisting of 100 trees on the Breast Cancer dataset:

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

The random forest gives us an accuracy of 97%, better than the linear models or a single decision tree, without tuning any parameters. We could adjust the max_features setting, or apply pre-pruning as we did for the single decision tree. However, often the default parameters of the random forest already work quite well.

Supervised Learning: Ensembles of Decision Trees: Random Forest

Analyzing random forests: Similarly to the decision tree, the random forest provides feature importances, which are computed by aggregating the feature importances over the trees in the forest. Typically, the feature importances provided by the random forest are more reliable than the ones provided by a single tree.

```
plot_feature_importances_cancer(forest)
```

Supervised Learning: Ensembles of Decision Trees: Random Forest

Analyzing random forests:

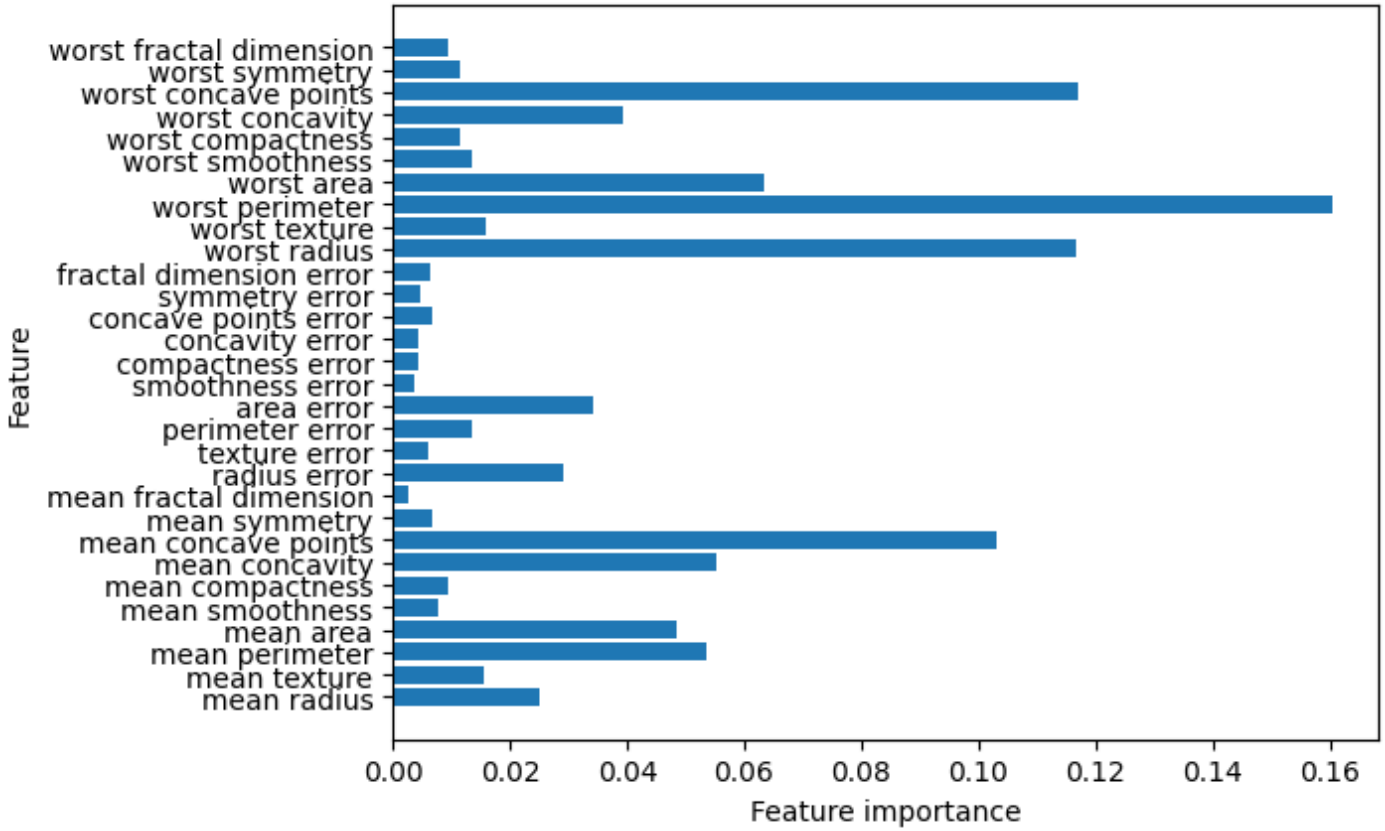

Figure: Feature importances computed from a random forest that was fit to the Breast Cancer dataset

Supervised Learning: Ensembles of Decision Trees: Random Forest

Strengths, Weaknesses, and parameters:

- **Strengths:**
  - They often work effectively without extensive parameter tuning.
  - No need for data scaling, as they are invariant to feature scaling.
  - Parallelizable on multi-core processors, making them suitable for large datasets.
  - Easily interpretable decision-making process for a single decision tree.
- **Weaknesses:**
  - Not suitable when a compact decision-making representation is needed, as they consist of many trees.
  - Interpretability can be challenging with numerous deep trees.
  - Sensitivity to random state settings; different states can result in varying models.
  - Less suitable for high-dimensional sparse data, where linear models might be more appropriate.
  - Requires more memory and is slower compared to linear models.

Supervised Learning: Ensembles of Decision Trees: Random Forest

Strengths, Weaknesses, and parameters:

- **Parameters:**
  - `n_estimators`: Larger values are generally better for better ensemble robustness but require more time and memory.
  - `max_features`: Controls randomness in each tree. Default values ($\sqrt{\text{n\_features}}$ for classification, $\log_2(\text{n\_features})$ for regression) often work well.
  - Pre-pruning options like `max_depth` can be used to limit tree depth.
  - For reproducible results, setting `random_state` is crucial.
  - Consider `max_leaf_nodes` for improved performance and reduced resource requirements.

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees (gradient boosting machines)

Definition

Gradient Boosted Regression Trees, suitable for regression and classification, sequentially build trees to correct errors from previous ones. They are memory-efficient and faster for predictions. By combining many simple models, they achieve high accuracy. Adjust key parameters like learning rate and the number of trees (n_estimators) for optimal results.

In gradient boosting, the learning_rate parameter influences how aggressively each tree corrects the errors of its predecessors. A higher learning rate allows for more substantial corrections and leads to more complex models.

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

Require: Training dataset $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, M iterations, learning rate ν, max tree depth d, loss function $L(F, y)$ (MSE or Cross-Entropy) test data $\{x_1^*, x_2^*, \ldots, x_k^*\}$

Ensure: Predicted target values $\{y_1^*, y_2^*, \ldots, y_k^*\}$

Initialization:

Initialize ensemble model: $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^{n} L(\gamma, y_i)$, γ the initial prediction

for $m = 1$ **to** M **do**

Step 1: Compute Negative Gradient:

Calculate negative gradient r_{im} w.r.t. $F_{m-1}(x_i)$ for each training example x_i:

$$r_{im} = -\left[\frac{\partial L(F_{m-1}(x_i), y_i)}{\partial F_{m-1}(x_i)}\right]_{F_{m-1}(x_i)=F_{m-1}(x_i)}$$

Step 2: Fit a Regression Tree:

Train regression tree $h_m(x)$ using r_{im} as target variable and input features x_i as predictors. Max tree depth is d.

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

1: **for** $m = 1$ **to** M **do**
2: ...
3: **Step 3: Update Ensemble Model:**
4: Update ensemble model $F_m(x)$ by adding prediction of newly trained tree scaled by ν:

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x)$$

5: **Step 4: Update Loss Function:**
6: Update loss function $L(F, y)$ by adding loss contributed by newly added tree:

$$L(F, y) = L(F_{m-1} + \nu \cdot h_m, y)$$

7: **Termination:**
8: Repeat Steps 1 to 4 for M iterations or until predefined stopping criterion is met during training.

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

1: **Inference:**
2: **for** each test sample x_i^* **do**
3: **Step 5: Predict Target Value:**
4: Predict target value y_i^* for test sample x_i^* using final ensemble model $F_M(x_i^*)$:

$$y_i^* = F_M(x_i^*)$$

5: **Output:**
6: Predicted target values $\{y_1^*, y_2^*, \ldots, y_k^*\}$

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

Example of using GradientBoostingClassifier on the Breast Cancer dataset. By default, 100 trees of maximum depth 3 and a learning rate of 0.1 are used:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

Visualize the feature importances

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
plot_feature_importances_cancer(gbrt)
```

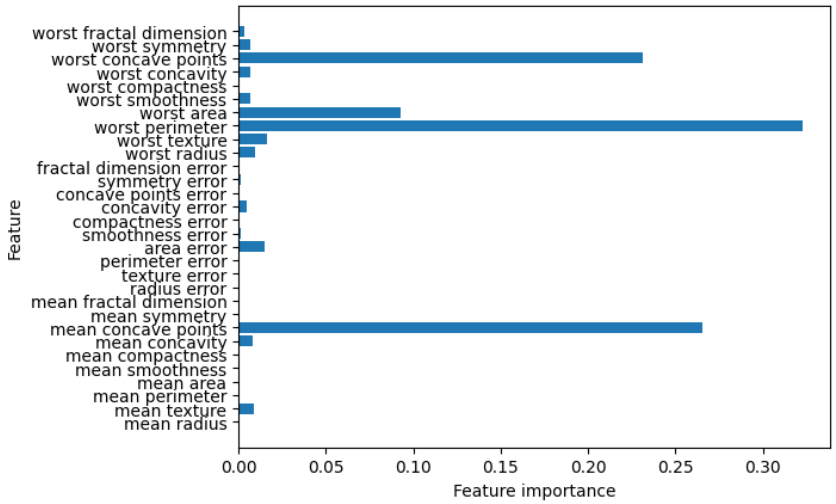

Figure: eature importances computed from a gradient boosting classifier that was fit to the Breast Cancer dataset

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

Strengths, weaknesses, and parameters:

- **Strengths:**
  - Effective for various tasks, including regression and classification.
  - They can capture complex relationships in data.
  - Suitable for a mix of binary and continuous features.
  - Works well without feature scaling.
- **Weaknesses:**
  - Requires careful parameter tuning, which can be time-consuming.
  - Training may take a considerable amount of time, especially with a large number of trees.
  - Tends not to perform well on high-dimensional sparse data.

Supervised Learning: Ensembles of Decision Trees: Gradient boosted regression trees

Strengths, weaknesses, and parameters:

- **Parameters:**
  - **n_estimators:** Number of trees in the ensemble. Higher values lead to more complex models but may risk overfitting.
  - **learning_rate:** Controls the correction strength of each tree. A lower value means more trees are needed for the same complexity.
  - **max_depth:** Maximum depth of each tree. Typically set low (e.g., not deeper than five splits) to control complexity.
  - **max_leaf_nodes:** Alternatively, you can use this to limit the number of leaf nodes in each tree.

Supervised Learning: Kernelized Support Vector Machines