Northeastern University

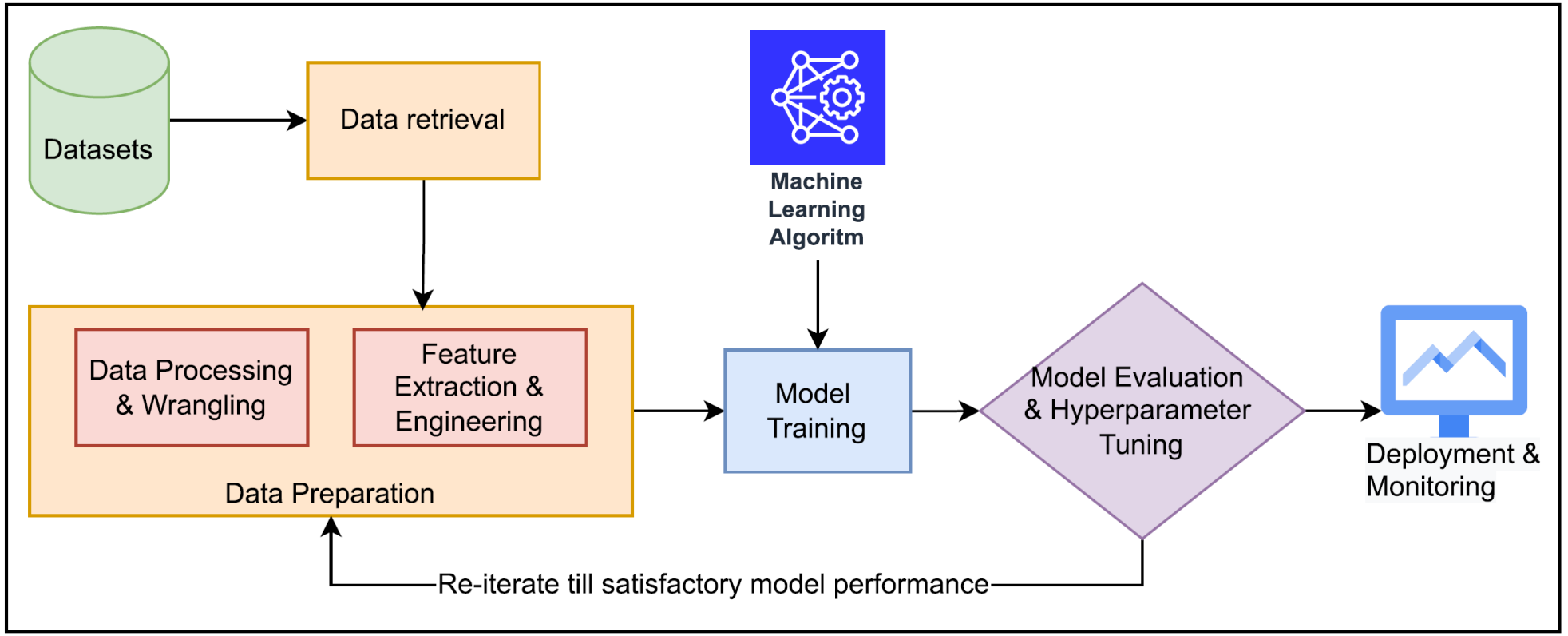
Introduction to Scikit-Learn

Roi Yehoshua

Agenda

- The machine learning pipeline
- The estimator API
- Exploratory data analysis
- Data preprocessing
- Training a model
- Evaluating a model
- Cross-validation
- Pipelines
- Hyperparameter tuning
- Model deployment
- Custom estimators

The Machine Learning Pipeline



Scikit-Learn

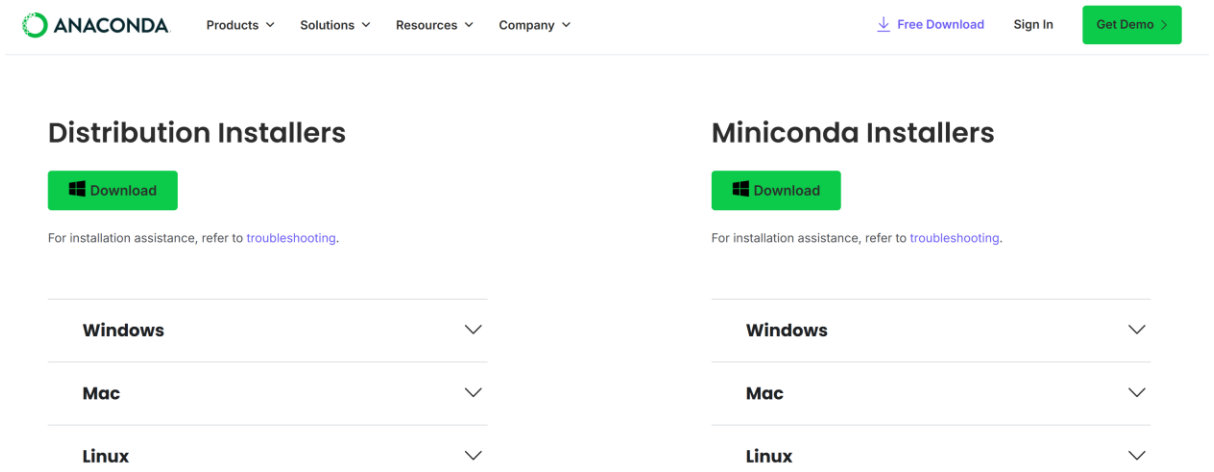
- A popular open-source library for machine learning in Python
  - Widely used both in academia and industry
- Provides efficient implementations for a wide range of ML algorithms
  - Including supervised, unsupervised, and semi-supervised algorithms
  - Most algorithms are implemented in Cython or use C libraries under the hood
- Offers a large number of utilities and tools for building ML pipelines
  - Data preprocessing, hyperparameter tuning, model evaluation, etc.
- Has a consistent and user-friendly API
- Seamless integration with other Python libraries, such as NumPy and Pandas
- Extended documentation available at http://scikit-learn.org/

Installation

- Can be easily installed using pip:

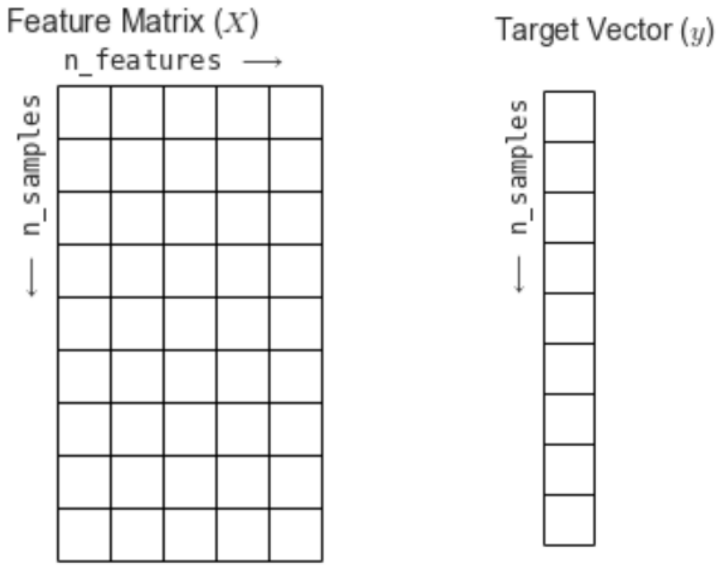
```
pip install scikit-learn
```

- Depends on other scientific libraries being installed, such as NumPy and SciPy
- Recommended option: Install Python Anaconda Distribution
  - Contains both the Python interpreter and many other scientific libraries
  - https://www.anaconda.com/download



Data Representation in Scikit-Learn

- Algorithms in Scikit-Learn expect the input data to follow a specific format
- **Input features** (X): A 2D array of shape (n_samples, n_features)
  - Can be a NumPy array or a SciPy sparse matrix
  - Pandas DataFrame is also supported in most cases (internally converted to NumPy)
- **Target vector** (y): A 1D array of shape (n_samples,)



The Estimator API

- An **estimator** is any object that learns from data
  - Must have at least a **fit()** method
- The estimator API provides a uniform interface to all algorithms in Scikit-Learn
  - Including models, transformers, pipelines, and search tools
- Main types of estimators
  - **Predictors**: Estimators that can make predictions
    - Have **fit()** and **predict()** methods
    - e.g., classifiers, regressors
  - **Transformers**: Preprocessing tools
    - Have **fit()** and **transform()** methods
    - e.g., StandardScaler for feature scaling, SimpleImputer for imputing missing values

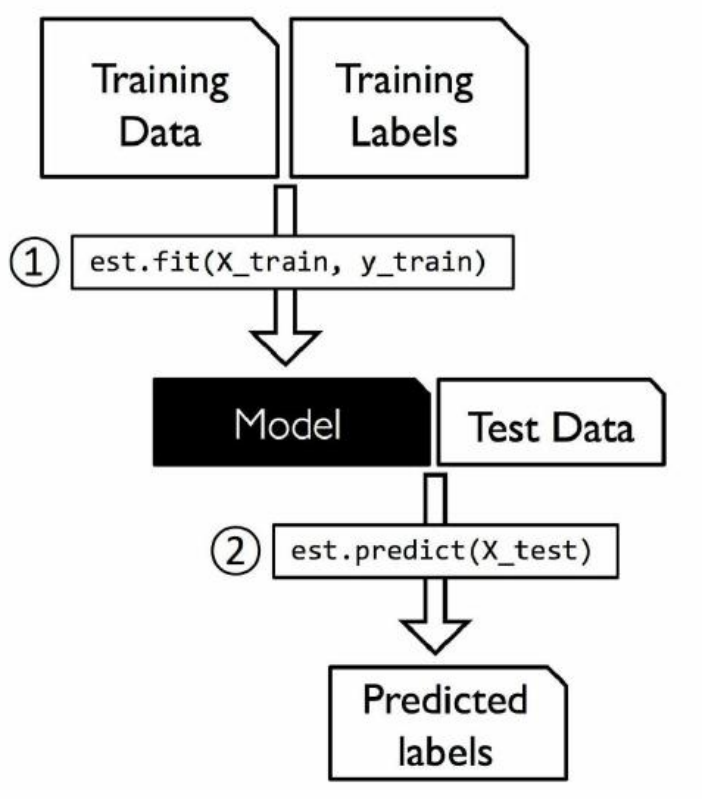
The Estimator API

- Core methods:

Method	Description	Estimator Type
fit(X, y)	Trains the estimator on the given data; learns model parameters or transformation rules	All estimators
predict(X)	Predicts target values for new inputs	Predictors
predict_proba(X)	Returns class probabilities instead of hard labels	Classifiers
score(X, y)	Computes a performance metric (e.g., accuracy)	Predictors
transform(X)	Transforms the data (e.g., scaling, encoding)	Transformers
fit_transform(X)	Fits and transforms in one step	Transformers
get_params()	Returns a dictionary of hyperparameters	All estimators
set_params(**kwargs)	Updates the estimator's hyperparameters	All estimators

Estimators

- Typical workflow of an estimator



Estimators

- Example for training a classifier and using it for predictions

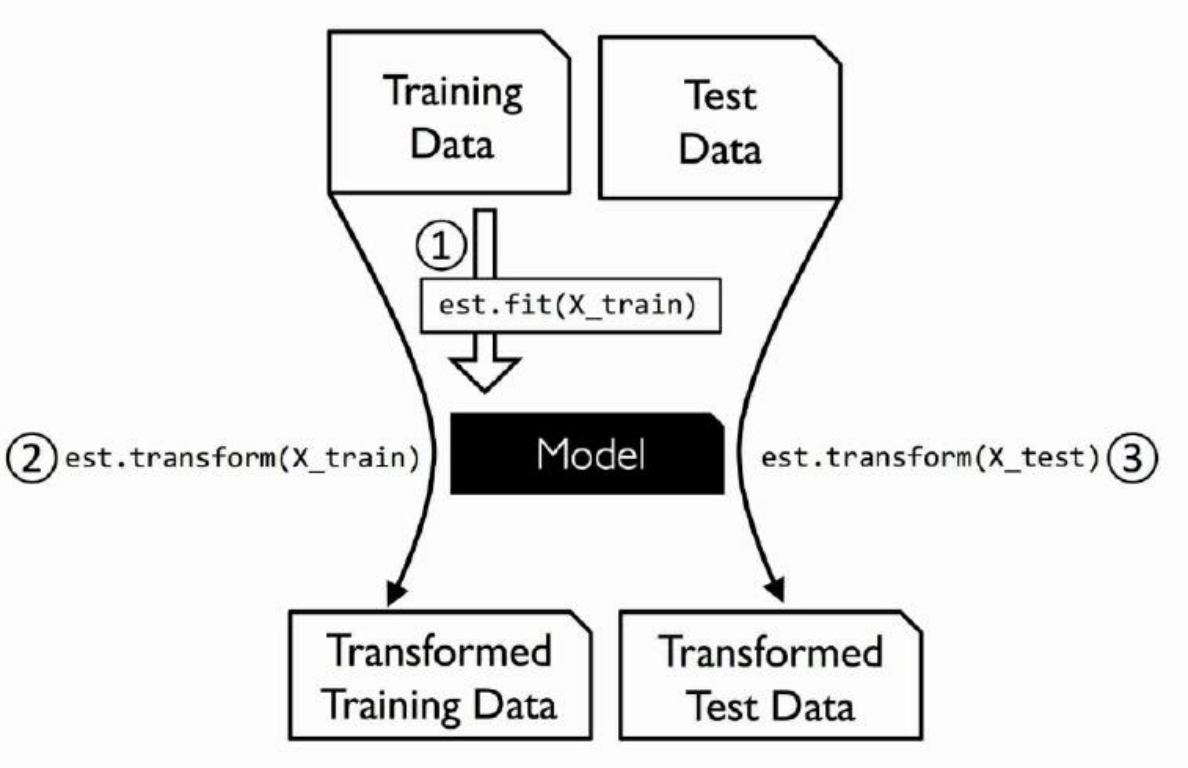
```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(max_depth=3)
X_train = [[5.1, 3.5, 1.4, 0.2],
           [4.9, 3.0, 1.4, 0.2],
           [7.0, 3.2, 4.7, 1.4],
           [6.4, 3.2, 4.5, 1.5],
           [6.3, 3.3, 6.0, 2.5]]
y_train = [0, 0, 1, 1, 2]

clf.fit(X_train, y_train);
```

```
X_test = [[4.7, 3.2, 1.3, 0.2]]

y_pred = clf.predict(X_test)
y_pred
```

```
array([0])
```

Transformers

- Typical workflow of a transformer:



Transformers

- Example for using a transformer:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled
```

```
array([[-1.04157134,  1.60018938, -1.17729571, -1.0994609 ],
       [-1.28956451, -1.47709789, -1.17729571, -1.0994609 ],
       [ 1.31436383, -0.24618298,  0.58864786,  0.27486523],
       [ 0.5703843 , -0.24618298,  0.48162097,  0.3893924 ],
       [ 0.44638772,  0.36927447,  1.2843226 ,  1.53466417]])
```

```
X_test_scaled = scaler.transform(X_test)
X_test_scaled
```

```
array([[-1.53755769, -0.24618298, -1.23080916, -1.0994609 ]])
```

Common Datasets

- Two main types of datasets used in ML
  - Synthetic (artificial) datasets
  - Real-world datasets
- Each type has its own advantages and is used for different purposes

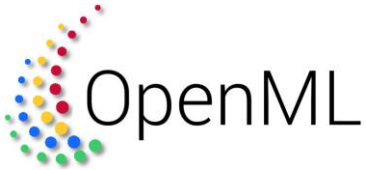
Synthetic Datasets

- Artificially generated using mathematical functions or simulations
- Cheap and fast to generate
- Allow precise control over:
  - Number of samples and features
  - Noise level and class separability
  - Distribution and structure of the objective function
- Useful for testing and debugging
- sklearn.datasets provides various functions for generating synthetic datasets
  - e.g., make_classification generates datasets with informative, redundant, and noisy features

```
sklearn.datasets.make_classification(n_samples=100, n_features=20, *,
n_informative=2, n_redundant=2, n_repeated=0, n_classes=2, n_clusters_per_class=2,
weights=None, flip_y=0.01, class_sep=1.0, hypercube=True, shift=0.0, scale=1.0,
shuffle=True, random_state=None, return_X_y=True)                    [source]
```

Real-World Datasets

- Collected from real applications
  - Sensors, user data, social media, healthcare records, etc.
- Typically noisy, incomplete, imbalanced, and high-dimensional
  - Require preprocessing, cleaning, and feature engineering
- Available in different formats, e.g., CSV, JSON, databases, ARFF files
- Common sources
  - UCI Machine Learning Repository
  - OpenML
  - Kaggle Datasets
  - Domain-specific repositories
    - e.g., ImageNet for vision, Common Crawl for NLP



Toy Datasets in Scikit-Learn

- sklearn.datasets provides several toy datasets for experimentation and learning
- Loaded directly into memory using **load_*** functions
  - return_X_y: whether to return the data split into features matrix X and label vector y
    - Otherwise, returns a dictionary with features, targets, and metadata
  - as_frame: whether to return the data as a Pandas DataFrame (default is NumPy array)

`load_iris` (*[, return_X_y, as_frame])	Load and return the iris dataset (classification).
`load_diabetes` (*[, return_X_y, as_frame, scaled])	Load and return the diabetes dataset (regression).
`load_digits` (*[, n_class, return_X_y, as_frame])	Load and return the digits dataset (classification).
`load_linnerud` (*[, return_X_y, as_frame])	Load and return the physical exercise Linnerud dataset.
`load_wine` (*[, return_X_y, as_frame])	Load and return the wine dataset (classification).
`load_breast_cancer` (*[, return_X_y, as_frame])	Load and return the breast cancer Wisconsin dataset (classification).

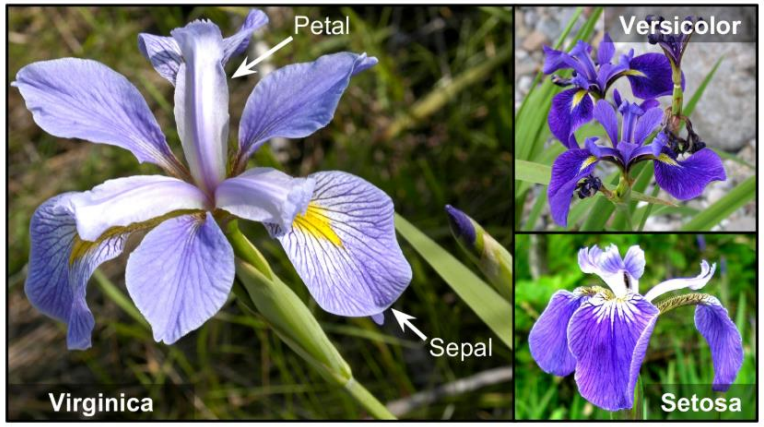
The Iris Dataset

- For example, let's load the Iris dataset using the function **load_iris()**

```
from sklearn.datasets import load_iris
```

```
X, y = load_iris(as_frame=True, return_X_y=True)
X
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...



EDA (Exploratory Data Analysis)

- Examine the dataset visually and statistically to understand the data
  - Basic structure, feature distributions, patterns, anomalies, relationships between features
- Guides preprocessing decisions
- Typical EDA steps
  - Summary statistics: mean, median, std, min, max
  - Missing value analysis
  - Correlation analysis
  - Target variable inspection (for supervised tasks)
  - Visualizations
    - Histogram, boxplots (distributions of numerical features)
    - Bar charts (categorical features)
    - Scatter plots, pair plots (relationships)

Feature Data Types

- Main types of features:
  - **Numerical**: Represent quantitative values
    - **Discrete**: Integer-valued (e.g., age, movie rating)
    - **Continuous**: Real-valued (e.g., income, temperature, weight)
  - **Categorical**: Represent qualitative labels or categories
    - **Nominal**: No natural order (e.g., color, occupation, ZIP code)
    - **Ordinal**: Ordered categories (e.g., education level, shirt size)
  - Non-tabular or complex feature types
    - e.g., text documents, images, audio, graphs
- Different feature types require different preprocessing techniques
  - e.g., scaling for numerical, encoding for categorical, vectorization for text

Feature Data Types

- The DataFrame's **info()** displays a summary of the dataset
  - Including feature names, data types, and number of non-null (non-missing) entries

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

Summary Statistics

- Numerical summaries describing the central tendency, spread, and shape of the data
- Can help you understand distributions, detect outliers, and identify skewness
- The **describe()** method displays summary statistics for the numerical features

```
X.describe()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

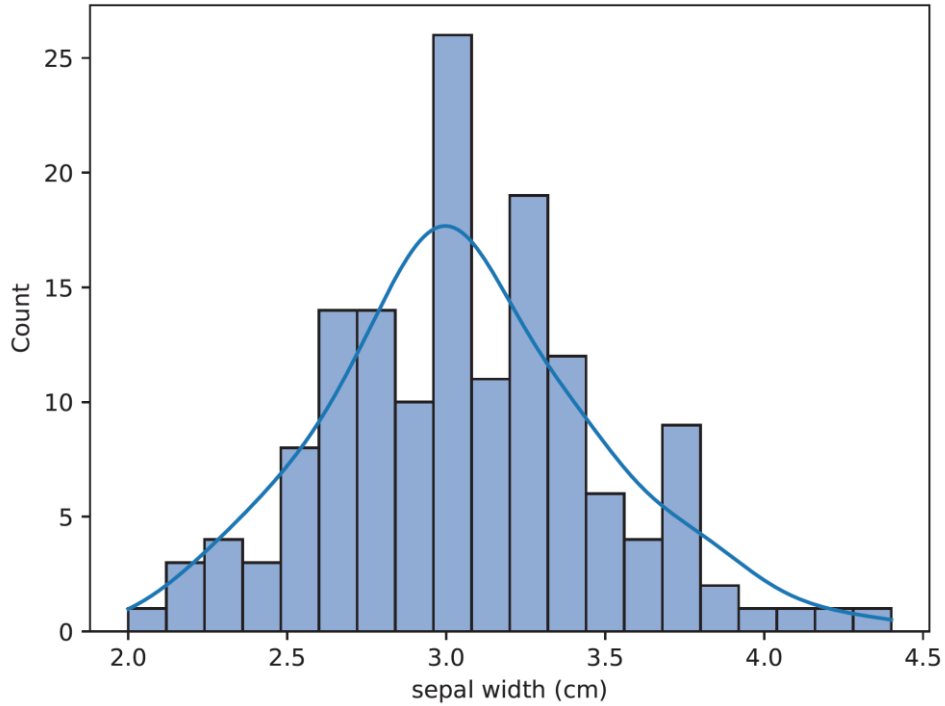
Visualization of Distributions

- Understand the shape of the data: normal, skewed, multimodal, etc.
- Detect outliers
- Compare distributions across groups or classes
- Common plots
  - Histogram: Shows frequency distribution of values
  - Boxplot: Visualizes median, quartiles, outliers
  - Bar plot: Categorical variable frequencies
  - Density plot (KDE): Smooth curve of distribution
  - Violin plot: Combines boxplot and KDE for density

Histograms

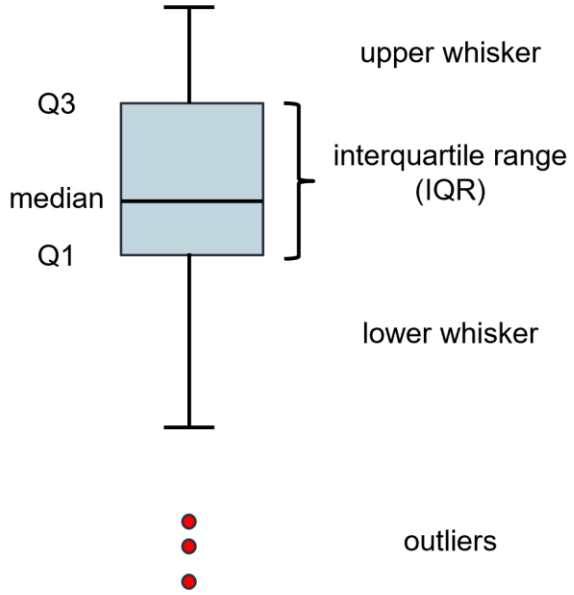
- You can use seaborn's histplot() function to plot histograms

```
import seaborn as sns

sns.histplot(data=X, x='sepal width (cm)', kde=True, bins=20)
```



Box Plots

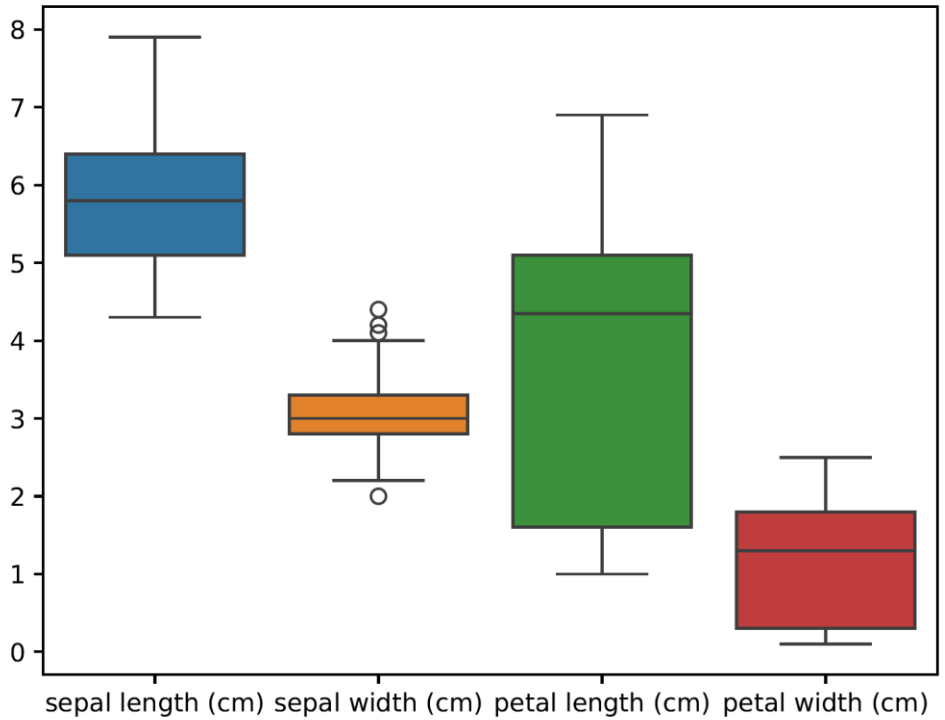
- A compact visualization of a distribution of a numerical feature
- The box spans from the 25th percentile (Q1) to the 75th percentile (Q3)
- Whiskers extend to the lowest and highest values within 1.5 × IQR
- Dots outside the whiskers represent potential outliers



Box Plots

- You can use Seaborn's **boxplot()** function to generate this plot:

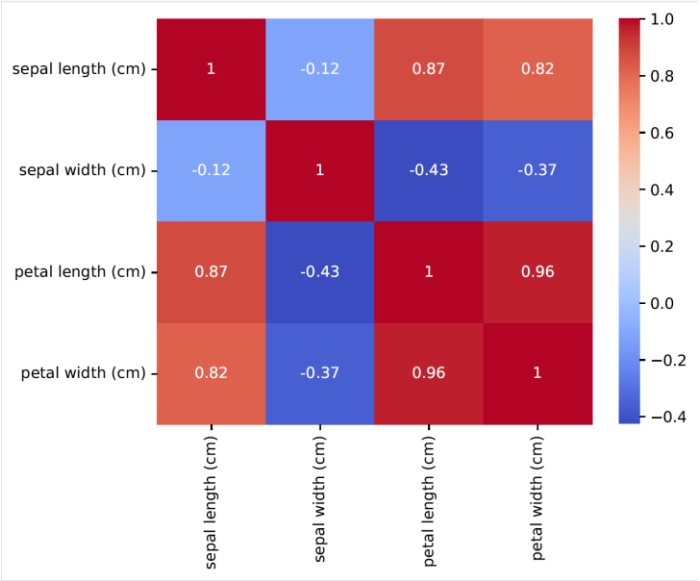
```
sns.boxplot(X)
```



Correlation Analysis

- Analyze the relationships among the features and between features and the target
  - Helps identify redundant features (highly correlated)
  - Reveals features that are strongly correlated with the target (good predictors)
- DataFrame's **corr()** method returns a matrix with pairwise correlation coefficients
  - Its **method** parameter specifies the correlation type (the default is Pearson coefficient)

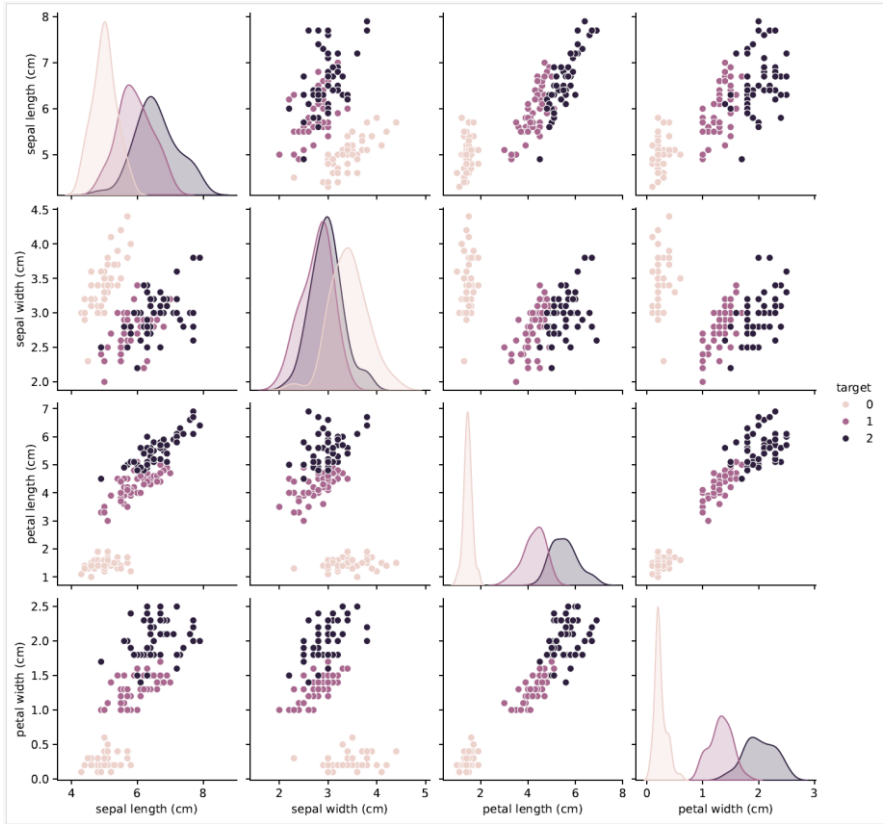
```
corr_matrix = X.corr()
sns.heatmap(X.corr(), annot=True, cmap='coolwarm')
```



Correlation Analysis

- A **pair plot** is a grid of scatter plots showing pairwise relationships between features
- Helps visualize linear and nonlinear patterns

```
import pandas as pd

df = pd.concat([X, y], axis=1)
sns.pairplot(df, hue='target')
```



Class Distribution Balance

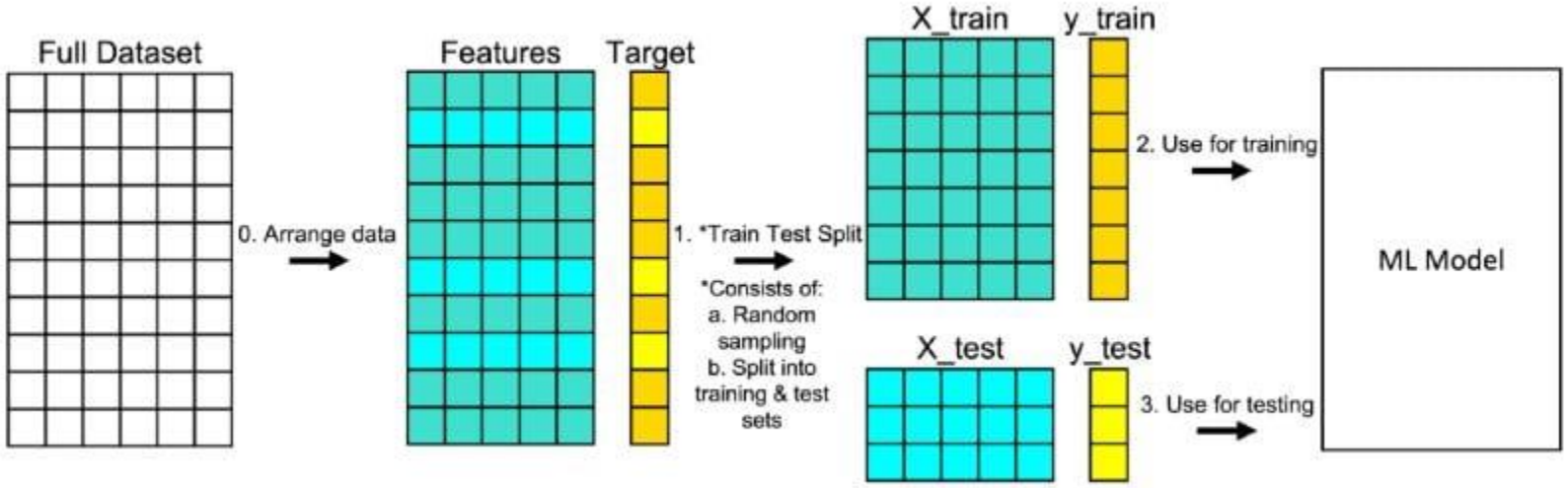
- Check how evenly distributed the class labels are in a classification dataset
- **Imbalanced datasets** have classes that are significantly underrepresented
- Imbalanced data can lead to:
  - Poor performance on minority classes
  - Misleading metrics like accuracy
    - e.g., achieving 95% accuracy by always predicting the majority class
- The **value_count()** method returns the number of instances in each class

```
y.value_counts()
```

```
target
0    50
1    50
2    50
Name: count, dtype: int64
```

Train-Test Split

- To evaluate how well the model generalizes to unseen data, we split the dataset into:
  - **Training set**: Used to fit the model
  - **Test set**: Used to evaluate model performance on new, unseen data
- Common split ratios are 70-80% for train and 20-30% for test
- The test set should remain completely unseen during model development
  - To avoid **data leakage** (using information from the test to influence the model)



Train-Test Split

- Scikit-Learn provides the **train_test_split()** function to perform the splitting:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, stratify=y)
```

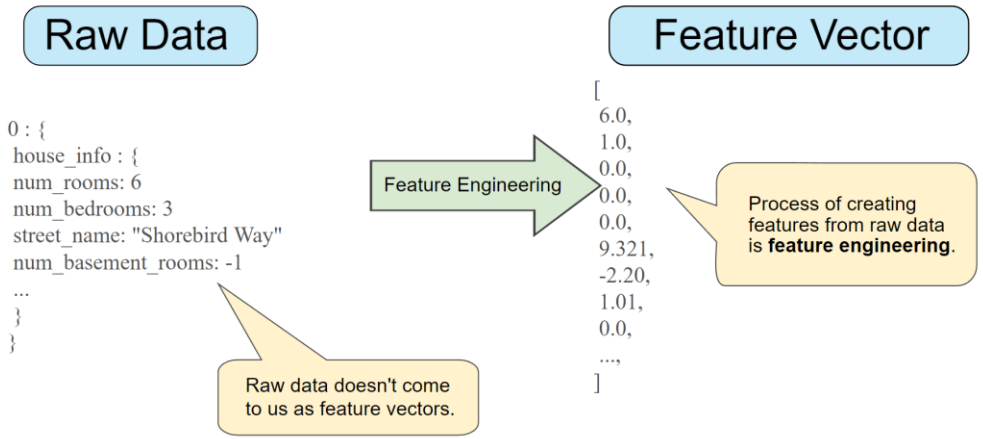
```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((112, 4), (38, 4), (112,), (38,))
```

  - Use **random_state** for reproducibility
  - Stratify the split (stratify=y) to keep the same class distribution in the training and test sets
    - Especially important for imbalanced classes
  - The default split ratio is 75% train, 25% test
    - Can be modified by specifying either **train_size** or **test_size**

Feature Engineering

- In the real world, data rarely arrives in a clean [samples, features] format
- **Feature engineering** is the process of creating useful features from raw data



- Feature engineering typically involves:
  - **Data preprocessing:** cleaning and transforming raw data into a usable format
  - **Feature selection**: choosing the most relevant features for the task
  - **Feature extraction**: creating new features by combining or transforming existing ones

Data Preprocessing

- Prepare the data for modeling by handling issues identified during EDA
- Common preprocessing tasks
  - Handling missing values
  - Encoding categorical variables
  - Scaling numerical features
  - Discretization of numerical features
  - Detecting and treating outliers
  - Extract features from unstructured data (e.g., text, images)
- sklearn.preprocessing provides many transformers for data preprocessing

Handling Missing Data

- One of the most common issues in real-world datasets
- Can arise from data entry errors, optional form fields, skipped user responses, etc.
- Most ML algorithms cannot work with missing values (e.g., NaN, null)
- Common strategies for handling missing data

Strategy	Description	When to Use
Remove rows	Drop samples with missing values	When only a few rows are affected
Remove columns	Drop features with too many missing values	If a feature is mostly missing
Mean/Median Impute	Fill with mean or median of the column	For numerical data with low variance
Mode Impute	Fill with most frequent value	For categorical features
Custom Imputation	Domain-specific values (e.g., "unknown")	When missingness has meaning
Predictive Imputation	Use models to predict the missing value	When data is missing not at random

Imputation of Missing Data

- SimpleImputer is an imputation transformer for completing missing values

```
class sklearn.impute.SimpleImputer(*, missing_values=nan, strategy='mean',
fill_value=None, copy=True, add_indicator=False, keep_empty_features=False)
```
[source]

- The **strategy** argument defines the imputation strategy
  - Options include: 'mean', 'median', 'most_frequent', and 'constant'

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='mean')
X = np.array([[np.nan, 5, np.nan],
              [2, 4, 10],
              [3, None, 5]])
imputer.fit_transform(X)
```

```
array([[ 2.5,  5. ,  7.5],
       [ 2. ,  4. , 10. ],
       [ 3. ,  4.5,  5. ]])
```

Encoding Categorical Variables

- Most machine learning models require numerical inputs
- Categorical variables must be converted to numbers
- Common encoding methods:

Strategy	Description	When to Use
One-hot encoding	Creates binary columns for each category (e.g., color_red, color_blue, color_green)	Nominal (no order) categories
Ordinal encoding	Assigns an integer to each category (e.g., low = 0, medium = 1, high = 2)	Ordinal categories, model can use numeric relationships like >, <
Label encoding	Assigns arbitrary IDs to the label categories (e.g., setosa = 0, virginica = 1, versicolor = 2)	Categories are just distinct labels, any numerical meaning is arbitrary
Target encoding	Encode based on statistics like frequency or target mean	Advanced use cases

One-Hot Encoding

- Converts categorical features into a set of binary (0/1) features, where each category becomes its own column

id	color
1	red
2	blue
3	green
4	blue

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

- Pros
  - Doesn’t introduce false ordinal relationships
  - Enables the model to learn independent mappings between each category and the target
- Cons
  - Can lead to high-dimensional data if the feature has many categories (e.g., zip codes)

One-Hot Encoding

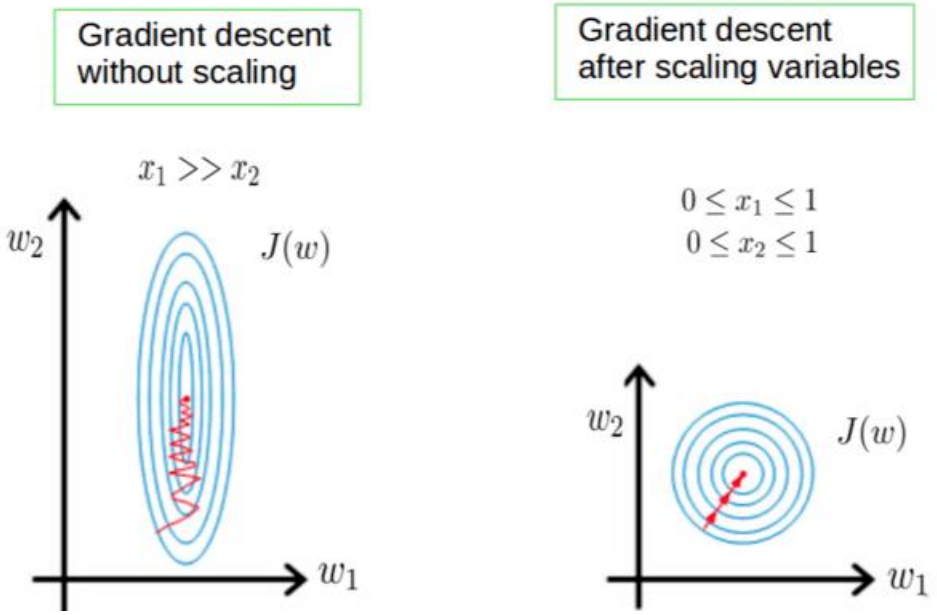
- The OneHotEncoder class can be used for one-hot encoding
- By default, it returns a SciPy sparse matrix in which only nonzero values are stored
  - Specify **sparse_output=False** to get a dense array
- Use **handle_unknown='ignore'** to avoid errors when encountering unseen categories

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
X = np.array([['Red'], ['Blue'], ['Green'], ['Blue'], ['Green']])
encoder.fit_transform(X)
```

```
array([[0., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

Feature Scaling

- Many ML algorithms are sensitive to the scale of numerical features
  - Especially algorithms that rely on **distances** or **gradient-based** optimization
- Feature scaling ensures that all features contribute equally during model training



Feature Scaling

- Common scaling methods

Strategy	Description	When to Use	Equation	Scikit-Learn Class
Standardization	Rescales to zero mean and unit variance	The most common	$x' = \frac{x - \overline{x}}{\sigma}$	StandardScaler
Min-max scaling	Rescales to a [min, max] range (typically [0, 1])	When inputs must be bounded	$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$	MinMaxScaler
Robust scaling	Uses median and IQR	Robust to outliers	$x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$	RobustScaler

Standard Scaling

- Example for using the StandardScaler on the Iris dataset:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled[:5]
```

```
array([[ 1.79213839, -0.60238047,  1.31532306,  0.92095427],
       [ 2.14531053, -0.60238047,  1.65320421,  1.05135487],
       [-0.4446185 , -1.50797259, -0.03620155, -0.25265117],
       [ 0.26172578, -0.60238047,  0.13273902,  0.13855064],
       [-0.4446185 , -1.28157456,  0.13273902,  0.13855064]])
```

```
X_train_scaled.mean(axis=0)
```

```
array([9.35759406e-16, 4.75809868e-17, 2.61695427e-16, 1.70498536e-16])
```

```
X_train_scaled.var(axis=0)
```

```
array([1., 1., 1., 1.])
```

Choosing a Model

- The choice of a machine learning model depends on multiple factors
  - Task type (regression, classification, clustering, etc.)
  - Size of dataset (small, medium, large)
  - Linearity of relationships between features and target
  - Interpretability vs. predictive power
  - Training time / inference time
- Start with a simple, fast-to-train model
  - e.g., linear regression for regression, logistic regression for classification
- Then try more complex models
  - Random forest or XGBoost for tabular data
  - Neural networks for unstructured data (text, image, audio)
- No-free-lunch theorem (no single algorithm is best for all problems)

Training a Model

- Create an instance of the desired estimator
- Specify the **hyperparameters** in the constructor of the estimator
  - Settings defined before training that influence how the model learns, e.g., learning rate
  - They control model complexity, regularization, and training behavior
  - They are not learned from the data (unlike model parameters)
  - Scikit-Learn provides default values that work well in many cases
  - Tuning these hyperparameters can improve model performance

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False,
tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100, multi_class='deprecated', verbose=0,
warm_start=False, n_jobs=None, l1_ratio=None)                    [source]
```

- Call the **fit()** method of the estimator with the training features and labels

Training a Model

- Example for training a logistic regression model on the Iris dataset:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(random_state=42)
X_train = X_train_scaled
model.fit(X_train, y_train)
```

```
LogisticRegression
LogisticRegression(random_state=42)
```

- Many algorithms are stochastic in nature (i.e., involve random choices)
  - Specifying the **random_state** ensures that repeated runs produce the same results
  - For robust evaluation, repeat each run with multiple random seeds and report the average

Learned Parameters

- Parameters are values that a model learns from the training data during .fit()
- They define the internal state of the model used to make predictions
- Stored in model attributes that end with an underscore suffix _
- Inspecting them can provide insights into how the model makes decisions
  - e.g., which features are the most important
- For example, to inspect the coefficients learned by the logistic regression model:

```
model.coef_
```

```
array([[-1.049416  ,  1.1534895 , -1.62864463, -1.56781588],
       [ 0.49537647, -0.53075959, -0.26388111, -0.80838681],
       [ 0.55403953, -0.62272991,  1.89252574,  2.37620269]])
```

Model Evaluation

- Assess whether the model is generalizing well or needs further tuning
- There are various evaluation metrics depending on the task
  - Classification: accuracy, precision, recall, F1 score, AUC
  - Regression: RMSE (Root Mean Square Error), MAE (Mean Absolute Error), R^2 score
- Evaluate the model on both training and test sets
  - If both scores are low → likely **underfitting**
  - If training score >> test score → likely **overfitting**
- The model's built-in **score()** method returns a default metric
  - Accuracy for classifiers
  - R^2 score for regressors
- Additional scoring methods are available in the sklearn.metrics module

Model Evaluation

- For example, evaluating the accuracy of our model on the training set:

```
train_accuracy = model.score(X_train, y_train)
print(f'Train accuracy: {train_accuracy:.4f}')
```

```
Train accuracy: 0.9643
```

- To evaluate the model on the test set, we must apply the same preprocessing steps
  - Use **transform()** on the test set (not fit_transform()) to avoid data leakage

```
X_test = scaler.transform(X_test)
test_accuracy = model.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

```
Test accuracy: 0.9211
```

  - While the gap suggests mild overfitting, it is more likely due to the small test size
    - Simple linear models typically don’t overfit, especially in low-dimensional spaces

Making Predictions

- Use the **predict**(X) method to make predictions
  - X is a feature matrix for a set of samples (typically test samples)
  - Returns a vector with the predicted labels for each sample
- For example, predicting the label of a single sample from the test set:

```
test_sample = X_test[5]  # extract test sample no.5
y_pred = model.predict([test_sample])
print(y_pred)  # predicted label
```

```
[1]
```

```
y_test.iloc[5] # the actual label
```

```
1
```

Confusion Matrix

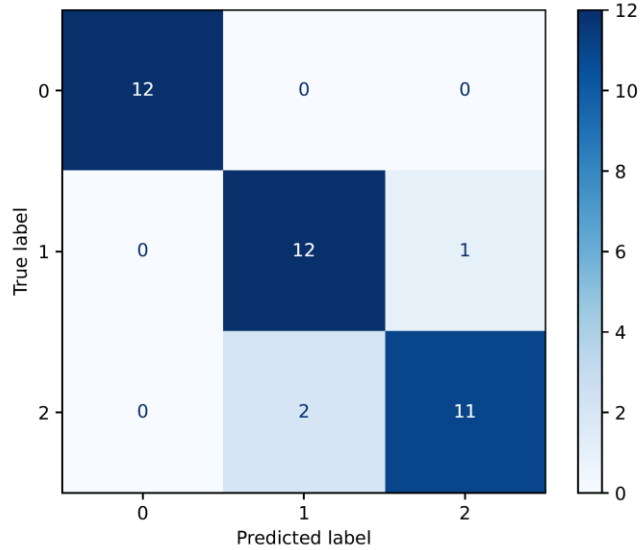
- Summarizes the performance of a classifier by comparing predicted vs. actual labels
- Provides detailed insight into type of errors made by the model

		Predicted Class	
		+	−
Actual Class	+	True Positives (TP)	False Negatives (FN)
	−	False Positives (FP)	True Negatives (TN)

- In Scikit-Learn, use confusion_matrix() to compute the matrix
  - Use ConfusionMatrixDisplay to visualize it as a heatmap

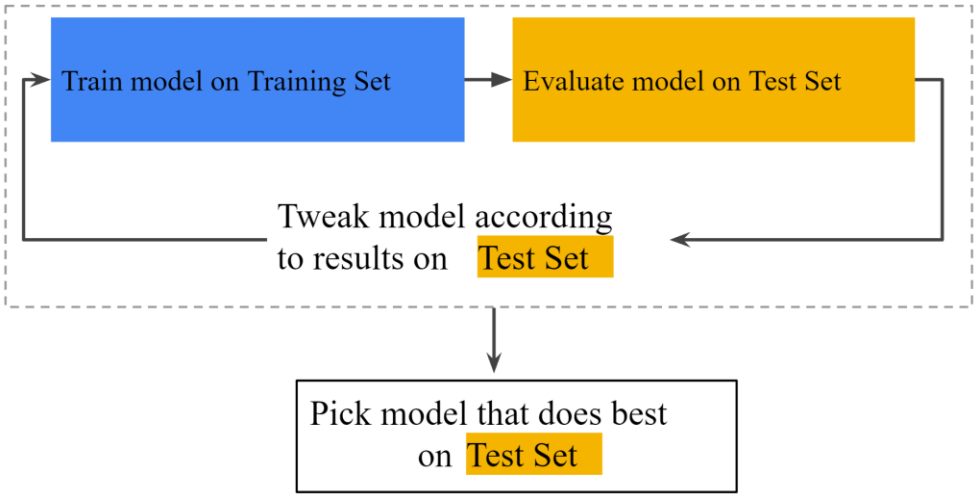
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X_train, y_train, cv=5)
print('CV scores:', np.round(scores, 4))
print(f'Average score: {scores.mean():.4f} ± {scores.std():.4f}')
```

```
CV scores: [1.     0.9565 0.9545 1.     0.9091]
Average score: 0.9640 ± 0.0339
```



Model Selection

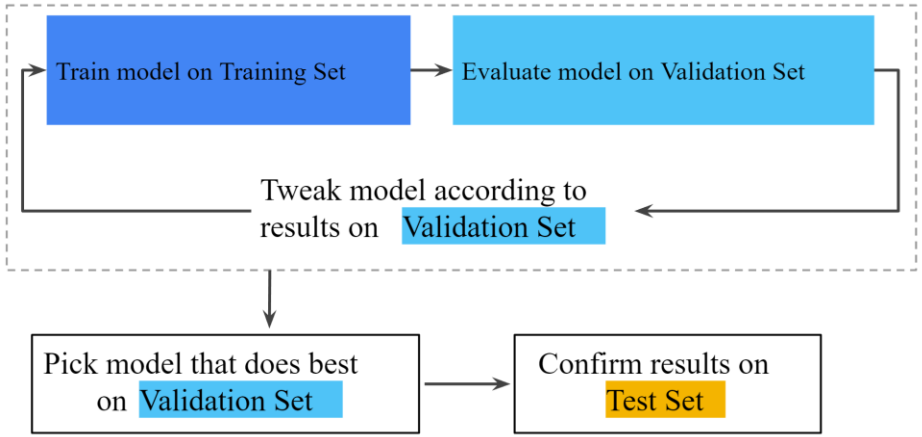
- **Goal:** choose the best model from a set of candidate models
  - across different types of models (e.g., logistic regression, SVM, KNN, etc.)
  - across models of the same type configured with different hyperparameters
- Drawbacks of using a single test set for model selection
  - Results may be unreliable if the test set is small or unrepresentative
  - Can cause overfitting to the specific test set if used repeatedly to make model choices



Validation Set

- A better approach is to split the data set into three disjoint subsets:

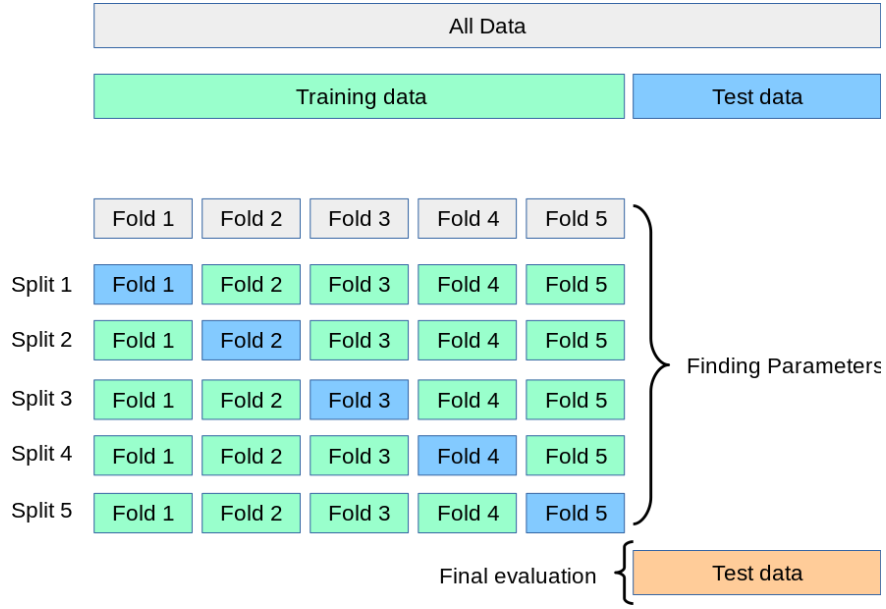
- Use the validation set to evaluate different models / hyperparameters
- Use the test set only for the final evaluation



- **Problem**: we lose a decent amount of the training data for validation

k-Fold Cross-Validation

- Evaluate the model by training and testing it on multiple subsets of the training data
- Randomly split the training set into k equal-sized folds S_1, …, S_k
- For i = 1, …, k:
  - Train the model on k – 1 folds (all folds except S_i)
  - Test the model on the remaining fold S_i
- Report the average score across all k folds



- Advantages
  - More reliable estimates of generalization performance
  - Provides variability estimation (standard deviation of the scores)
  - In small datasets, can use more data for the training (using a larger k)

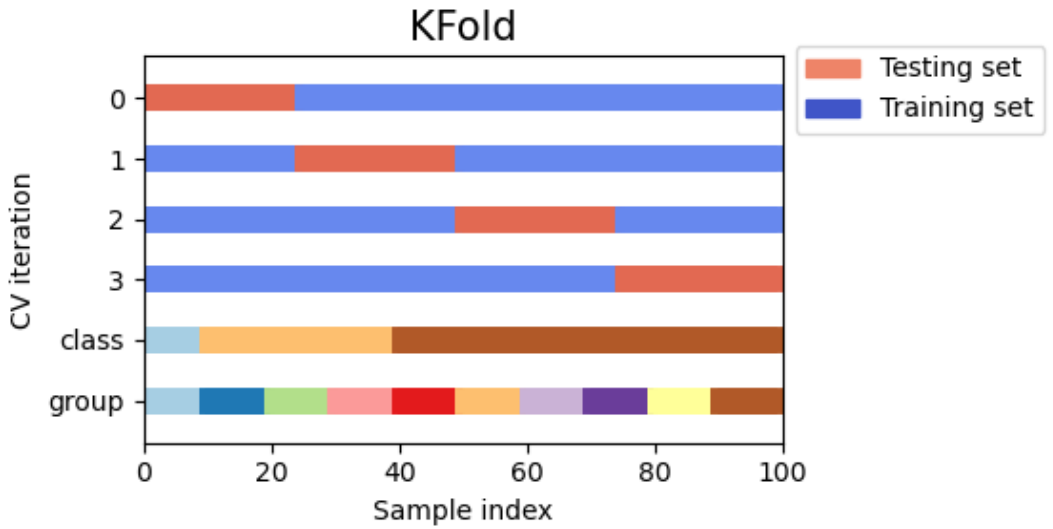
Leave-One-Out Cross Validation (LOOCV)

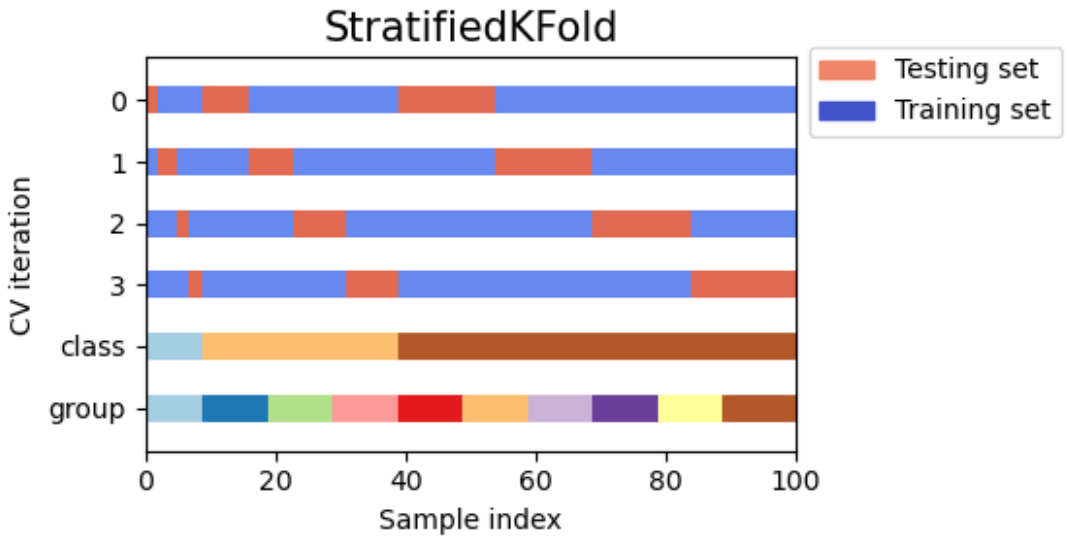
- A special case of *k*-fold cross-validation where $k = n$ (the size of the training set)
- Repeatedly train on all but one of the training examples
  - Test on that one held-out example
- The resulting *n* errors are then averaged to obtain the generalization error

- **Advantage**: utilizing as much data as possible for training
- **Disadvantages**:
  - More computationally expensive
  - The variance of the performance metric tends to be high

Stratified Sampling

- Generate folds that preserve the percentage of samples for each class





Cross-Validation in Scikit-Learn

- The method cross_val_score() can be used to perform cross-validation

```
sklearn.model_selection.cross_val_score(estimator, X, y=None, *, groups=None,
scoring=None, cv=None, n_jobs=None, verbose=0, params=None, pre_dispatch='2*n_jobs',
```

  - The **cv** argument determines the splitting strategy, options include:
    - None: uses the default 5-fold cross validation
    - an integer that specifies the number of folds
    - An iterable of (train, test) index splits (e.g., KFold, StratifiedKFold, or custom)
- Example on the Iris dataset

```python
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X_train, y_train, cv=5)
print('CV scores:', np.round(scores, 4))
print(f'Average score: {scores.mean():.4f} ± {scores.std():.4f}')
```

```
CV scores: [1.     0.9565 0.9545 1.     0.9091]
Average score: 0.9640 ± 0.0339
```

Comparing Different Models

- After establishing a baseline, test alternative models to see if they perform better
  - Use the same training and test sets for all models
  - Apply consistent preprocessing to ensure fairness
- Evaluate multiple criteria
  - Performance (accuracy, F1 score, AUC, etc.)
  - Training time and prediction speed
  - Additional aspects such as interpretability and ease of deployment
- (Advanced) Apply statistical tests to assess if differences are statistically significant
  - e.g., paired t-test, Wilcoxon signed-rank test

Comparing Different Models

- We first define a dictionary of classifiers:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier

# Create a dictionary of classifiers
models = {
    'Logistic Regression': LogisticRegression(random_state=42),
    'KNN': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(random_state=42),
    'SVM': SVC(random_state=42),
    'MLP': MLPClassifier(random_state=42, max_iter=1000)
}
```

Comparing Different Models

- Next, we iterate over all models: train, evaluate, and measure training time

```
import time

results = {}

for name, model in models.items():
    start = time.time()
    model.fit(X_train, y_train)
    end = time.time()

    results[name] = {}
    results[name]['Training Accuracy'] = model.score(X_train, y_train)
    results[name]['Test Accuracy'] = model.score(X_test, y_test)
    results[name]['CV Accuracy'] = cross_val_score(model, X_train, y_train).mean()
    results[name]['Training Time (s)'] = end - start
```

Comparing Different Models

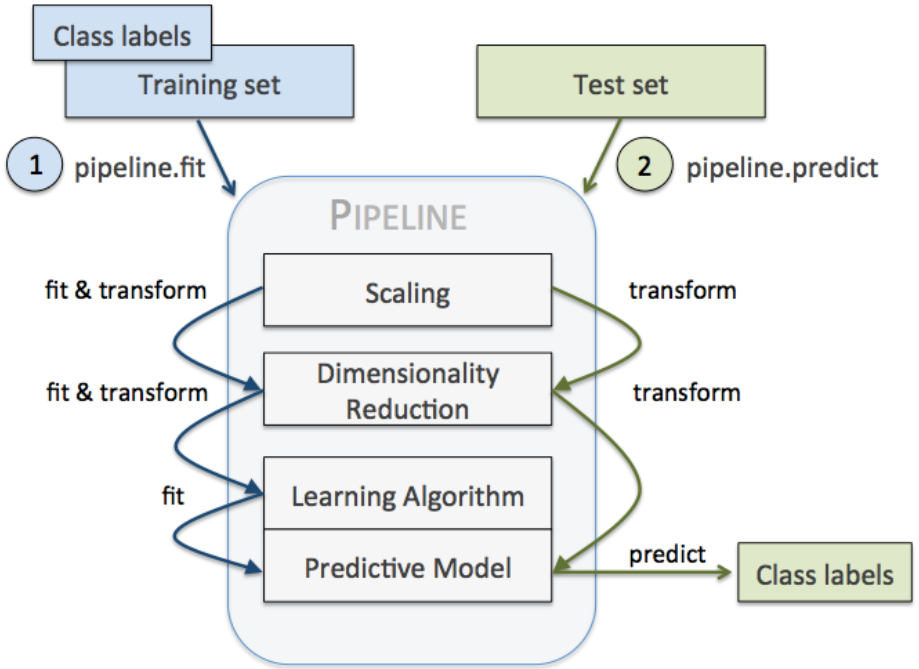
- Display the results in a DataFrame:

```
results_df = pd.DataFrame.from_dict(results).T
results_df
```

	Training Accuracy	Test Accuracy	CV Accuracy	Training Time (s)
Logistic Regression	0.964286	0.921053	0.964032	0.004986
KNN	0.982143	0.921053	0.928458	0.000000
Decision Tree	1.000000	0.894737	0.929249	0.002031
Random Forest	1.000000	0.921053	0.937549	0.120712
Gradient Boosting	1.000000	0.973684	0.937945	0.228478
SVM	0.964286	0.947368	0.954941	0.000998
MLP	0.982143	0.921053	0.945850	0.187517

Scikit-Learn Pipelines

- A pipeline allows you to chain multiple transformers with a final estimator
  - Ensures consistent preprocessing during training and inference
  - Simplifies code and makes it easier to deploy and maintain the model
  - Integrates seamlessly with other tools such as cross-validation and grid search



Scikit-Learn Pipelines

- A pipeline that combines StandardScaler with a LogisticRegression estimator:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])
```

- A pipeline supports all the methods of the final estimator
  - e.g., fit(), predict(), and score()

```
pipeline.fit(X_train, y_train)
pipeline.score(X_test, y_test)
```

```
0.9210526315789473
```

Scikit-Learn Pipelines

- Use the **steps** or **named_steps** attributes to access specific components:

```
pipeline.named_steps['scaler']
```

```
StandardScaler
StandardScaler()
```

- To update parameters, use the **set_params()** method:
  - Parameters are specified using the format <step_name>__<parameter_name>

```
pipeline.set_params(model__C=0.1)  # Change C in LogisticRegression
pipeline.named_steps['model'].C
```

```
0.1
```

ColumnTransformer

- Allows you to apply different preprocessing steps to different columns
- Essential for mixed data types (e.g., numeric + categorical)
- Keeps your pipeline modular and clean

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

numerical_features = ['age', 'income']
categorical_features = ['gender', 'city']

preprocessor = ColumnTransformer([
    ('num', StandardScaler(), numerical_features),
    ('cat', OneHotEncoder(), categorical_features)
])

pipeline = Pipeline([
    ('preprocess', preprocessor),
    ('model', LogisticRegression())
])
```

Example: The Titanic Dataset

- The Titanic dataset contains information about passengers on the Titanic
  - Including demographics, ticket details, and travel information
  - The target variable **survived** is binary (0, 1)
- We can load it using the **fetch_openml()** function:

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('titanic', version=1, return_X_y=True, as_frame=True)
X.head(3)
```

	pclass	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

Example: The Titanic Dataset

- The feature types are:

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 13 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   pclass     1309 non-null   int64
 1   name       1309 non-null   object
 2   sex        1309 non-null   category
 3   age        1046 non-null   float64
 4   sibsp      1309 non-null   int64
 5   parch      1309 non-null   int64
 6   ticket     1309 non-null   object
 7   fare       1308 non-null   float64
 8   cabin      295 non-null    object
 9   embarked   1307 non-null   category
 10  boat       486 non-null    object
 11  body       121 non-null    float64
 12  home.dest  745 non-null    object
dtypes: category(2), float64(3), int64(3), object(5)
memory usage: 115.4+ KB
```

Example: The Titanic Dataset

- We first drop columns with too many missing values or uninformative columns:

```
X = X.drop(columns=['cabin', 'boat', 'body', 'home.dest', 'name', 'ticket'])
X.head(3)
```

	pclass	sex	age	sibsp	parch	fare	embarked
0	1	female	29.0000	0	0	211.3375	S
1	1	male	0.9167	1	2	151.5500	S
2	1	female	2.0000	1	2	151.5500	S

- cabin – has too many missing values (over 75% are missing)
- boat – indicates whether someone had a lifeboat assignment (leaks the target)
- body – indicates the body ID if a passenger's body was recovered (leaks the target)
- home.dest – free text field with inconsistent formatting
- name – a free-text column

Example: The Titanic Dataset

- Identify the numerical and categorical features:

```
numerical_features = X.select_dtypes(include=['int64', 'float64']).columns.tolist()
categorical_features = X.select_dtypes(include=['object', 'category']).columns.tolist()

print('Numerical features:', numerical_features)
print('Categorical features:', categorical_features)
```

```
Numerical features: ['pclass', 'age', 'sibsp', 'parch', 'fare']
Categorical features: ['sex', 'embarked']
```

- Split the data:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
```

Example: The Titanic Dataset

- Define the pipelines for preprocessing and the ColumnTransformer:

```
# Preprocessing for numerical features
numerical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Preprocessing for categorical features
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combine using ColumnTransformer
preprocessor = ColumnTransformer([
    ('num', numerical_transformer, numerical_features),
    ('cat', categorical_transformer, categorical_features)
])
```

Example: The Titanic Dataset

- Create the full pipeline, train the model and evaluate it:

```
# Create the full pipeline
model = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', DecisionTreeClassifier(random_state=42))
])

# Train the pipeline
model.fit(X_train, y_train)

# Evaluate the model
print('Train accuracy:', np.round(model.score(X_train, y_train), 4))
print('Test accuracy:', np.round(model.score(X_test, y_test), 4))
```

```
Train accuracy: 0.9694
Test accuracy: 0.7439
```

Hyperparameter Tuning

- The process of finding the best combination of hyperparameters
  - that yields the best generalization performance on unseen data
- Common tuning methods:
  - **Manual**: Try different values based on domain knowledge or trial-and-error
  - **Grid search**: Exhaustive search over all combinations
  - **Random search**: Samples a fixed number of random combinations
  - **Bayesian optimization** (advanced): Learns which combinations to try next
- Practitioners often adopt a **hybrid approach**
  - Using intuition and experience to guide automated searches toward the most promising hyperparameters and effective ranges

Grid Search

- A method for exhaustively searching over a manually specified grid of hyperparameters
  - Define a **parameter grid** with a set of candidate values for each hyperparameter
  - Evaluate each parameter combination using **cross-validation**
  - The combination with the best average cross-validation score is selected
  - A final model is then trained on the entire training set using the best combination
- Implemented in Scikit-Learn by the class GridSearchCV

```
class sklearn.model_selection.GridSearchCV(estimator, param_grid, *, scoring=None,
n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs',
error_score=nan, return_train_score=False)                          [source]
```

  - **param_grid**: a dictionary mapping parameter names to lists of values
  - **cv**: the cross-validation splitting strategy
  - **n_jobs**: number of jobs to run in parallel (-1 means all processors)

Grid Search

- Let’s run grid search on our decision tree classifier:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'clf__criterion': ['gini', 'entropy'],
    'clf__max_depth': np.arange(1, 11),
    'clf__min_samples_split': np.arange(2, 11)
}
grid_search = GridSearchCV(model, param_grid, cv=3, n_jobs=-1)
grid_search.fit(X_train, y_train)
```

- The best hyperparameters found and their cross-validation average score:

```
print(grid_search.best_params_)
print('CV Score:', np.round(grid_search.best_score_, 4))
```

```
{'clf__criterion': 'gini', 'clf__max_depth': 3, 'clf__min_samples_split': 2}
CV Score: 0.8063
```

Grid Search

- After fitting, the GridSearchCV object behaves like the best estimator
- You can use it directly with methods like predict() and score()

```
train_accuracy = grid_search.score(X_train, y_train)
print(f'Train accuracy (after tuning): {train_accuracy:.4f}')

test_accuracy = grid_search.score(X_test, y_test)
print(f'Test accuracy (after tuning): {test_accuracy:.4f}')
```

```
Train accuracy (after tuning): 0.8114
Test accuracy (after tuning): 0.8323
```

- Test accuracy improved significantly after tuning: from 74.39% to 83.23%

Grid Search

- You can also inspect the scores of all combinations using the attribute **cv_results_**

```
df = pd.DataFrame(grid_search.cv_results_)
df = df[['params', 'mean_test_score']]
pd.set_option('display.max_colwidth', None)
df
```

	params	mean_test_score
0	{'clf__criterion': 'gini', 'clf__max_depth': 1, 'clf__min_samples_split': 2}	0.773700
1	{'clf__criterion': 'gini', 'clf__max_depth': 1, 'clf__min_samples_split': 3}	0.773700
2	{'clf__criterion': 'gini', 'clf__max_depth': 1, 'clf__min_samples_split': 4}	0.773700
3	{'clf__criterion': 'gini', 'clf__max_depth': 1, 'clf__min_samples_split': 5}	0.773700
4	{'clf__criterion': 'gini', 'clf__max_depth': 1, 'clf__min_samples_split': 6}	0.773700
...	...	...
175	{'clf__criterion': 'entropy', 'clf__max_depth': 10, 'clf__min_samples_split': 6}	0.769623
176	{'clf__criterion': 'entropy', 'clf__max_depth': 10, 'clf__min_samples_split': 7}	0.771662
177	{'clf__criterion': 'entropy', 'clf__max_depth': 10, 'clf__min_samples_split': 8}	0.769623
178	{'clf__criterion': 'entropy', 'clf__max_depth': 10, 'clf__min_samples_split': 9}	0.766565
179	{'clf__criterion': 'entropy', 'clf__max_depth': 10, 'clf__min_samples_split': 10}	0.774720

Random Search

- Samples a fixed number of random combinations from a specified parameter grid
- Can explore more diverse configurations in the same amount of time
- Implemented in Scikit-Learn by the class RandomizedSearchCV

```
class sklearn.model_selection.RandomizedSearchCV(estimator, param_distributions, *,
n_iter=10, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0,
pre_dispatch='2*n_jobs', random_state=None, error_score=nan,
return_train_score=False)                                          [source]
```

  - **param_distributions**: a dictionary mapping parameters to distributions or lists of values
    - Use scipy.stats distributions for continuous or large discrete spaces
  - **n_iter**: number of parameter combinations to sample

Random Search

- Example:

```
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'clf__criterion': ['gini', 'entropy'],
    'clf__max_depth': np.arange(1, 11),
    'clf__min_samples_split': np.arange(2, 11)
}
random_search = RandomizedSearchCV(model, param_grid, n_iter=20, cv=3, random_state=42, n_jobs=-1)
random_search.fit(X_train, y_train)
```

```
print(random_search.best_params_)
print('CV Score:', np.round(random_search.best_score_, 4))
```

```
{'clf__min_samples_split': 3, 'clf__max_depth': 3, 'clf__criterion': 'gini'}
CV Score: 0.8063
```

Model Deployment

- Making a trained machine learning model available for use in the real world
- Common deployment methods
  - **Batch scripts** (e.g., daily model runs on new data)
  - **Web APIs** (e.g., Flask, FastAPI, Django)
  - **Interactive apps** (e.g., Streamlit, Gradio for demos)
  - **Cloud services** (e.g., AWS SageMaker, GCP Vertex AI, Azure ML)
- Basic workflow
  - Save the trained model (e.g., with joblib or pickle)
  - Load and serve model via a web interface or API
  - Monitor and update as needed

Streamlit

- An open-source Python library for building interactive web apps
- Useful for quickly deploying machine learning models with a UI
- Install using pip

```
pip install streamlit
```

- To verify the installation

```
streamlit hello
```

  - This will launch a sample demo app in your browser
- Documentation available at https://streamlit.io/

Streamlit

- We first save the trained model and the metadata (feature names and class labels)

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
import pandas as pd
import joblib

# Load the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Train a Decision Tree
model = DecisionTreeClassifier(max_depth=3, random_state=42)
model.fit(X, y)

# Save the model and metadata
joblib.dump(model, "iris_model.pkl")
joblib.dump(iris.target_names, "target_names.pkl")
joblib.dump(iris.feature_names, "feature_names.pkl")
```

Streamlit

- We now build the Streamlit app in a iris_demo.py script

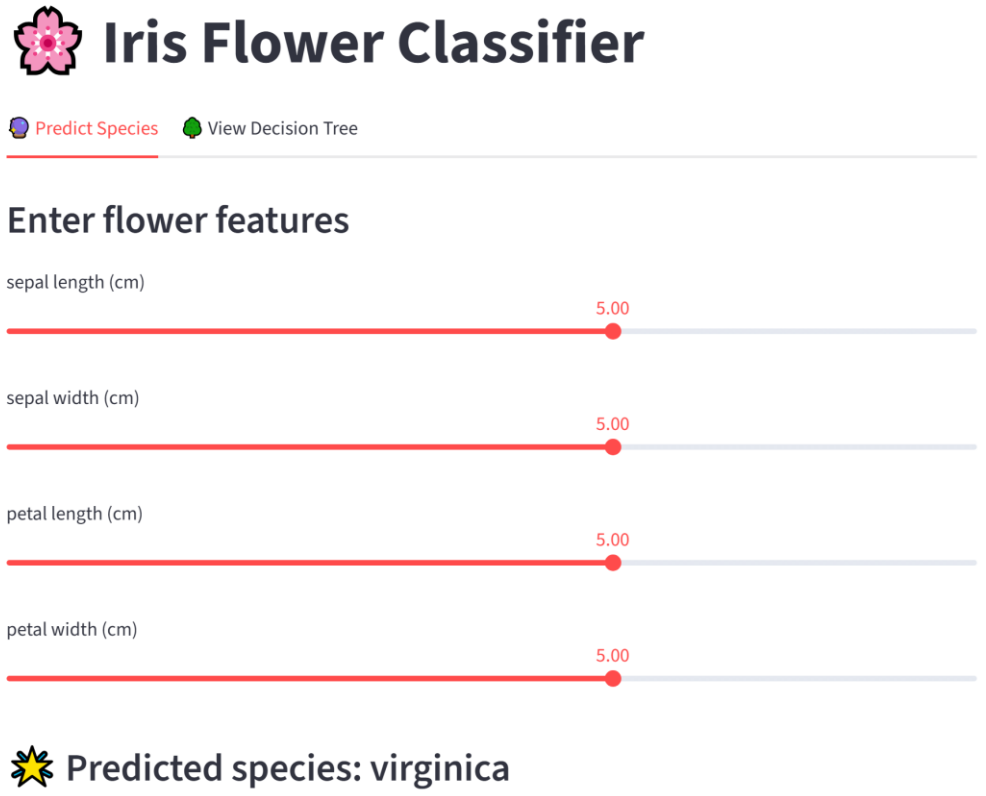
```
import streamlit as st
import numpy as np
import joblib
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Load model and metadata
model = joblib.load('iris_model.pkl')
target_names = joblib.load('target_names.pkl')
feature_names = joblib.load('feature_names.pkl')

# App title and tabs
st.title("🌸 Iris Flower Classifier")
tab1, tab2 = st.tabs(["🔮 Predict Species", "🌳 View Decision Tree"])
```

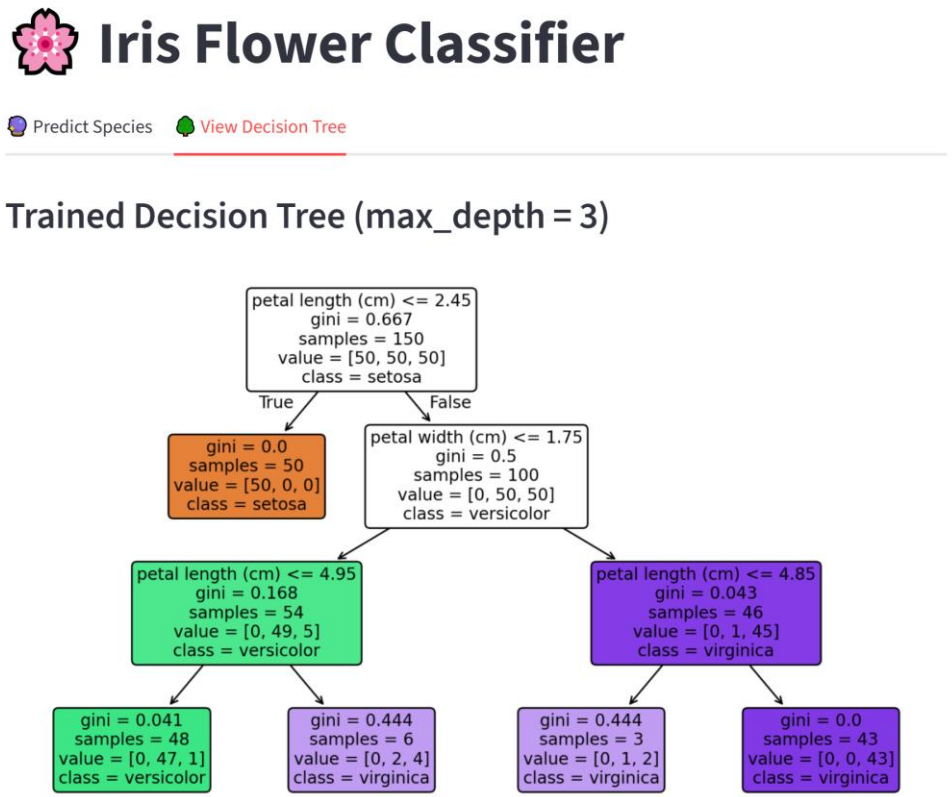
Streamlit

```
# --- TAB 1: Prediction ---
with tab1:
    st.subheader("Enter flower features")

    # User input sliders for the four features
    inputs = []
    for name in feature_names:
        val = st.slider(f"{name}", 0.0, 8.0, 5.0)
        inputs.append(val)
    input_features = np.array([inputs])

    # Predict and display result
    prediction = model.predict(input_features)[0]
    st.markdown(f"### 🌟 Predicted species: **{target_names[prediction]}**")

# --- TAB 2: Tree Visualization ---
with tab2:
    st.subheader("Trained Decision Tree (max_depth = 3)")
    fig, ax = plt.subplots(figsize=(10, 6))
    plot_tree(model, feature_names=feature_names,
              class_names=target_names, filled=True, rounded=True, ax=ax)
    st.pyplot(fig)
```

Streamlit

- Run the app from the terminal `streamlit run iris_demo.py`
- This will start the StreamLit app in your browser





Custom Estimators

- Build your own models or transformations
- User-defined classes that follow the Estimator API
  - Implement required methods like fit(), predict(), or transform()
- Allow full integration with Pipeline, GridSearchCV, and other Scikit-Learn tools
- Scikit-Learn provides base classes with helpful default implementations
  - BaseEstimator: adds get_params() and set_params()
  - ClassifierMixin: adds a default score() method (accuracy)
  - RegressorMixin: adds a default score() method (R^2 score)
  - TransformerMixin: adds a default fit_transform() implementation

Custom Estimators

- Example for a simple *k*-nearest-neighbor classifier:

```python
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from scipy.stats import mode

class SimpleKNNClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, k=5):
        self.k = k  # Number of nearest neighbors

    def fit(self, X, y):
        # Store the training examples and their labels
        self.X_train = X
        self.y_train = y
```

Custom Estimators

- Example for a simple *k*-nearest-neighbor classifier:

```
    def predict(self, X):
        predictions = []

        for x_test in X:
            # Compute Euclidean distances from x_test to all training points
            distances = np.sum((self.X_train - x_test)**2, axis=1)
            # Find indices of the k nearest neighbors
            neighbors_idx = distances.argsort()[:self.k]
            # Get their labels
            neighbors_labels = self.y_train[neighbors_idx]
            # Predict the most common label
            most_common = mode(neighbors_labels).mode
            predictions.append(most_common)

        return np.array(predictions)
```

Custom Estimators

- Testing the custom estimator:

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

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=42, stratify=y
)

clf = SimpleKNNClassifier(k=5)
clf.fit(X_train, y_train)
test_accuracy = clf.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

```
Test accuracy: 0.9737
```