Practical Machine Learning

Machine learning (ML) is a rapidly growing field within artificial intelligence (AI) that focuses on building systems capable of learning from data. Instead of being explicitly programmed with detailed rules, the ML models identify patterns and make predictions or decisions based on historical data. This approach has revolutionized many industries, including healthcare, finance, marketing, and technology, enabling applications like personalized recommendations, fraud detection, and speech recognition. As the volume of data continues to grow, understanding ML concepts and techniques becomes increasingly important for anyone interested in working with data or building intelligent systems.

This course provides a comprehensive introduction to the fundamental principles and techniques of ML. We will cover essential topics such as supervised learning, unsupervised learning, and model evaluation, as well as the basics of data preprocessing and feature selection. The participants will learn how to design and implement basic ML models using tools and frameworks commonly used in the industry. Through a combination of theory, practical exercises, and real-world examples, the participants will gain a solid foundation in ML, preparing them for further study or practical application in various domains.

By the end of the course, the participants will understand how the ML models are built, how to select the appropriate algorithms for specific problems, and how to assess model performance. In addition, the participants will be equipped with the skills to work on ML projects, from data collection and preparation to model training and evaluation. This knowledge will serve as a valuable stepping stone for those wishing to explore more advanced topics or specialize in areas such as deep learning (DL), natural language processing, or computer vision.

⚙ Prerequisites

- Familiarity with Python basics (loops, functions, lists, dictionaries) and libraries like NumPy, Pandas, and Matplotlib/Seaborn.
- Understanding vectors, matrices, matrix operations (multiplication, transpose, inverse), and eigenvalues.
- (Optional but useful): Derivatives, gradients, and optimization (especially for understanding gradient descent).

Setting Up Programming Environment

Using Personal Computer

This section provides instructions for installing the required packages and their dependencies on a local computer or server.

Install miniforge

If you already have a preferred way to manage Python versions and libraries, you can stick to that. Otherwise, we recommend you to install Python3 and all required libraries using Miniforge, a free minimal installer for the package, dependency, and environment manager Conda.

- Please follow the installation instructions to install Miniforge.
- After installation, verify that Conda is installed correctly by running:

```
$ conda --version

# Example output: conda 24.11.2
```

Configure programming environment

With Conda installed, open the **Anaconda Prompt terminal**, and run the command below to install required packages and depenencies (except PyTorch):

```
$ conda env create -y --file=https://raw.githubusercontent.com/ENCCS/practical-machine-learning/main/content/env/environment.yml
```

This creates a new environment called `practical_machine_learning`. We activate it and then install PyTorch library:

```
$ conda activate practical_machine_learning

$ conda install -y pytorch torchvision torchaudio torchtext cpuonly -c pytorch
```

Warning

Remember to activate your programming environment each time before running code examples. This ensures that the correct Python version and all required dependencies are available. If you forget to activate it, you may encounter errors or missing packages.

Validate programming environment

Once the programming environment is fully set up, open a new **Anaconda Prompt terminal** (just as you should do each time before running code examples), activate the programming environment, and launch JupyterLab by running the command below:

```
$ conda activate practical_machine_learning

$ jupyter lab
```

This will start a Jupyter server and automatically open the JupyterLab interface in your web browser.

To verify that all required packages are properly installed, follow these steps:

- Open JupyterLab (see instructions above).
- Create a new Jupyter Notebook by selecting **File** → **New** → **Notebook**.
- Copy the code examples listed below into a cell of the notebook.
- Run the cell (press **Shift + Enter** or click the **Run** button).

```
import numpy;      print('Numpy version: ',            numpy.__version__)
import pandas;     print('Pandas version: ',           pandas.__version__)
import scipy;      print('Scipy version: ',            scipy.__version__)
import matplotlib; print('Matplotlib version: ',      matplotlib.__version__)
import seaborn;    print('Seaborn version: ',          seaborn.__version__)
import sklearn;    print('Scikit-learn version: ',     sklearn.__version__)
import keras;      print('Keras version: ',            keras.__version__)
import tensorflow; print('Tensorflow version: ',      tensorflow.__version__)
import torch;      print('Pytorch version: ',          torch.__version__)
import umap;       print('Umap-learn version: ',       umap.__version__)
import notebook;   print('Jupyter Notebook version: ', notebook.__version__)
```

You should see output similar to the figure below. The exact package versions may vary depending on when you installed them.

```
import numpy;      print('Numpy version: ',            numpy.__version__)
import pandas;     print('Pandas version: ',           pandas.__version__)
import scipy;      print('Scipy version: ',            scipy.__version__)
import matplotlib; print('Matplotlib version: ',      matplotlib.__version__)
import seaborn;    print('Seaborn version: ',          seaborn.__version__)
import sklearn;    print('Scikit-learn version: ',     sklearn.__version__)
import keras;      print('Keras version: ',            keras.__version__)
import tensorflow; print('Tensorflow version: ',      tensorflow.__version__)
import torch;      print('Pytorch version: ',          torch.__version__)
import umap;       print('Umap-learn version: ',       umap.__version__)
import notebook;   print('Jupyter Notebook version: ', notebook.__version__)
```

```
Numpy version:  2.0.2
Pandas version:  2.3.2
Scipy version:  1.16.1
Matplotlib version:  3.10.6
Seaborn version:  0.13.2
Scikit-learn version:  1.7.1
Keras version:  3.11.2
Tensorflow version:  2.18.1
Pytorch version:  2.6.0
Umap-learn version:  0.5.9.post2
Jupyter Notebook version:  7.4.5
```

If the code runs without errors, it means that all packages are correctly installed and your programming environment is ready to use.

Warning

For Windows OS users, you might encounter an error (`ImportError: DLL load failed while importing _C: The specified procedure could not be found`) as described below.

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[1], line 9
      7 import keras;      print('Keras version: ',           keras.__version__)
      8 import tensorflow; print('Tensorflow version: ',     tensorflow.__version__)
----> 9 import torch;      print('Pytorch version: ',        torch.__version__)
     10 import umap;       print('Umap-learn version: ',     umap.__version__)
     11 import notebook;   print('Jupyter Notebook version: ', notebook.__version__)

File D:\00_program\241011-anaconda\envs\practical_machine_learning\Lib\site-packages\torch\__init__.py:405
    403     if USE_GLOBAL_DEPS:
    404         _load_global_deps()
--> 405     from torch._C import *  # noqa: F403
    408 class SymInt:
    409     """
    410     Like an int (including magic methods), but redirects all operations on the
    411     wrapped node. This is used in particular to symbolically record operations
    412     in the symbolic shape workflow.
    413     """

ImportError: DLL load failed while importing _C: The specified procedure could not be found.
```

It is a very common Windows-specific PyTorch issue, and it means that the underlying C++/CUDA DLLs that torch depends on could not be loaded correctly.

You should reinstall the correct matching build via the command below.

```
$ conda install -y pytorch torchvision torchaudio torchtext cpuonly -c pytorch
```

Note

If you are using VS Code, you can select the installed `practical_machine_learning` programming environment as follows:

- Open your project folder in VS Code.
- In the upper-right corner of the editor window (when working with a Python file or Jupyter Notebook), click on **Select Kernel**.
- From the list of **Python Environments**, locate and choose the `practical_machine_learning` environment (which you have installed earlier).
- Once selected, VS Code will use this environment for running Python code and Jupyter Notebooks, ensuring that all required packages are available.

(Optional) Setting Up PyTorch with GPU Support

For Windows OS users, if your computer has a GPU card, you can install PyTorch with GPU support. Below are step-by-step instructions to update the `practical_machine_learning` programming environment.

First check your CUDA version. Open a terminal (Linux/macOS) or PowerShell (Windows) and run:

```
$ nvcc --version
```

If `nvcc` is not in your PATH, you can instead run `nvidia-smi`.

```
$ nvidia-smi
```

Here is the output from my Windows machine:

```
(base) C:\Users\wangy>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Mon_Apr__3_17:36:15_Pacific_Daylight_Time_2023
Cuda compilation tools, release 12.1, V12.1.105
Build cuda_12.1.r12.1/compiler.32688072_0

(base) C:\Users\wangy>nvidia-smi
Wed Oct  1 22:01:49 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.94                 Driver Version: 560.94         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf           Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce GT 1030       WDDM  |   00000000:01:00.0  On |                  N/A |
| N/A   39C    P5             N/A /   30W |    1440MiB /   2048MiB |      2%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      4264    C+G   ...es (x86)\Dropbox\Client\Dropbox.exe      N/A      |
|    0   N/A  N/A      6164    C+G   ...\221203_tunhderbird\thunderbird.exe      N/A      |
|    0   N/A  N/A      6332    C+G   ...crosoft\Edge\Application\msedge.exe      N/A      |
|    0   N/A  N/A      8140    C+G   ...t_VSCode\Microsoft VS Code\Code.exe      N/A      |
|    0   N/A  N/A      9524    C+G   ...ogram\211003_gaussview_16\gview.exe      N/A      |
|    0   N/A  N/A     10232    C+G   ....Search_cw5n1h2txyewy\SearchApp.exe      N/A      |
|    0   N/A  N/A     10672    C+G   ...CBS_cw5n1h2txyewy\TextInputHost.exe      N/A      |
|    0   N/A  N/A     13900    C+G   ...oogle\Chrome\Application\chrome.exe      N/A      |
|    0   N/A  N/A     19812    C+G   ...y\AppData\Roaming\Zoom\bin\Zoom.exe      N/A      |
|    0   N/A  N/A     21264    C+G   ...\221203_tunhderbird\thunderbird.exe      N/A      |
|    0   N/A  N/A     22568    C+G   ...on\140.0.3485.94\msedgewebview2.exe      N/A      |
|    0   N/A  N/A     23356    C+G   ...siveControlPanel\SystemSettings.exe      N/A      |
|    0   N/A  N/A     23656    C+G   C:\Windows\explorer.exe                     N/A      |
|    0   N/A  N/A     30008    C+G   ...5n1h2txyewy\ShellExperienceHost.exe      N/A      |
+-----------------------------------------------------------------------------------------+
```

Second, remove any CPU-only versions of PyTorch that may have been installed (for example, those coming from Conda's defaults or conda-forge channels), and hten install an older, CUDA-compatible version of PyTorch directly using `pip`. Here `cu121` indicates the CUDA version (12.1) that the PyTorch build was compiled with.

```
$ conda activate practical_machine_learning

$ conda remove pytorch torchvision torchaudio torchtext

$ pip install torch==2.4.0 torchvision==0.19.0 torchaudio==2.4.0 --index-url
https://download.pytorch.org/whl/cu121
```

Third, verify your installation in a Jupyter Notebook. Run the following command and ensure it returns `True` for `torch.cuda.is_available()`.

```
import torch

print(torch.__version__)               # 2.4.0+cu121
print(torch.cuda.is_available())       # True
print(torch.cuda.get_device_name(0))   # NVIDIA GeForce GT 1030
```

Using Google Colab

You can also run all the code examples in tutorials using Google Colab, a free cloud-based platform that provides Jupyter Notebook environments with preinstalled ML libraries.

Download Jupyter Notebooks

- You can open each Jupyter Notebook (usually with the `.ipynb` extension) from HERE, and then select **Download raw file** to save it locally.
- Alternatively, you can download the entire repository at HERE by clicking the green `<> Code` button and choosing **Download ZIP** file. After unzipping the downloaded ZIP file, you will find all Jupyter Notebooks in the directory **practical-machine-learning-main/content/jupyter-notebooks**.

Upload Jupyter Notebooks to Google Drive

Sign in to your Google Drive, then upload the downloaded Jupyter Notebooks to a convenient folder. You can simply drag and drop the files directly into Google Drive or use the option **New → File upload**.

Open Jupyter Notebooks in Google Colab

Once uploaded, right-click the Jupyter Notebooks file in Google Drive and select **Open with → Google Colaboratory**. This will launch the notebook in Google Colab, where you can view, edit, and run the code cells interactively.

Connect to a Hosted Runtime

In Google Colab, go to the top-right corner and click **Connect** to link your notebook to a Google-hosted runtime environment. If you need GPU or TPU acceleration, select **Runtime → Change runtime type**, then choose the desired hardware accelerator.

Run the Code

After connecting, follow the instructions inside the Jupyter Notebooks. You can run each cell individually by pressing **Shift** + **Enter**, or execute the entire notebook using **Runtime → Run all**.

Introduction to Machine Learning

Objectives

- Provide a general overview of machine learning.
- Explain the relationship between artificial intelligence, machine learning, and deep learning.

- Explore representative real-world applications of machine learning.

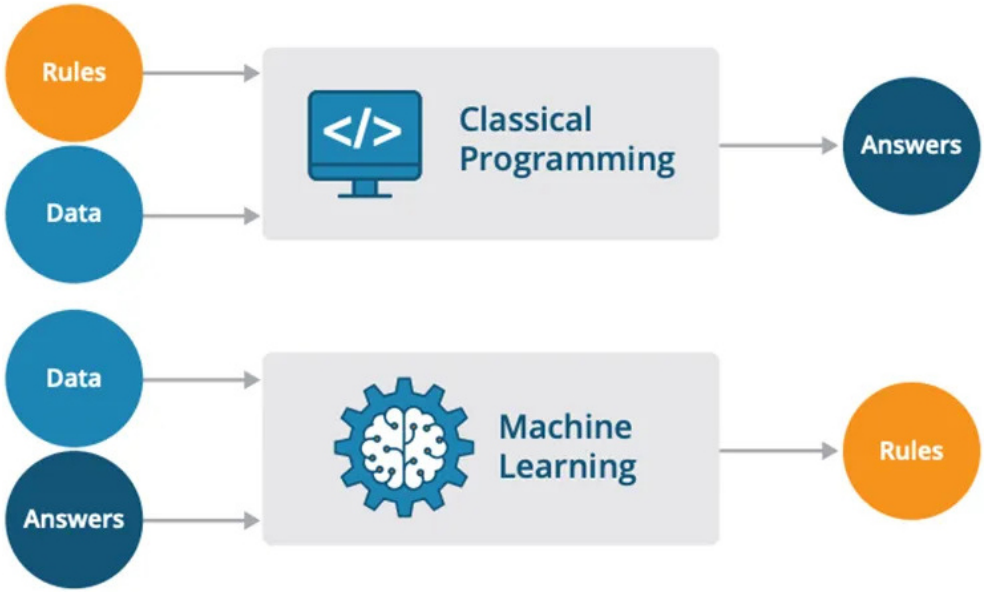
Instructor note

- 15 min teaching
- 0 min exercising

What is Machine Learning

Machine learning (ML) is a field of computer science that studies algorithms and techniques for automating solutions to complex problems that are hard to program using conventional programing methods.

In conventional programming, the programmer explicitly codes the logic (rules) to transform inputs (data) into outputs (answers), making it suitable for well-defined, rule-based tasks. In ML, the system learns the logic (rules) from data and answers, making it ideal for complex, pattern-based tasks where explicit rules are hard to define. The choice between them depends on the problem, data availability, and complexity.



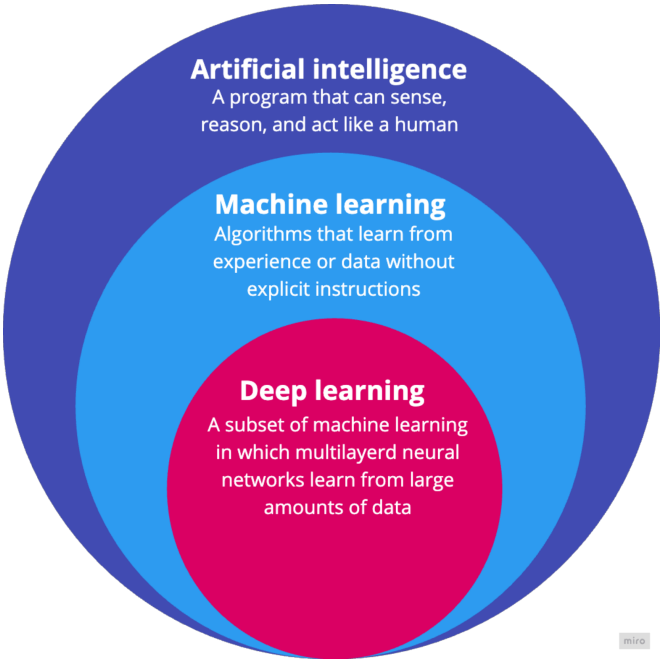
Classic programming vs. machine learning. Source

Relation with Artificial Intelligence and Deep Learning

Artificial intelligence (AI) is the broadest field, encompassing any technique that enables computers to mimic human intelligence, such as reasoning, problem-solving, perception, and decision-making. AI includes a wide range of approaches, from rule-based systems (like expert systems) to modern data-driven methods. It aims to create systems that can perform tasks that typically require human intelligence, such as playing chess, recognizing images, or understanding language.

ML is a subset of AI that focuses on algorithms and models that learn patterns from data to make predictions or decisions **without being explicitly programmed**. ML is one of the primary ways to achieve AI. It enables systems to improve performance over time by learning from experience (data) rather than relying solely on hardcoded rules. ML includes various techniques like supervised learning (*e.g.*, regression, classification), unsupervised learning (*e.g.*, clustering, dimensionality reduction), and reinforcement learning.

Deep learning (DL) is a specialized subset of ML, and it leverages artificial neural networks inspired by the human brain to tackle tasks like image recognition, speech processing, and natural language understanding. DL excels in handling unstructured data (*e.g.*, images, audio, text) and requires significant computational power and large datasets for training.



The relationship between artificial intelligence, machine learning, and deep learning. Source

Why Machine Learning?

ML is transforming how we solve complex problems in the real world by enabling systems to learn directly from data, rather than relying on explicitly programmed rules. In many real-world scenarios, such as medical diagnosis, stock market prediction, or natural language processing, the relationships between inputs and outputs are too complex or dynamic to define manually. ML models can uncover hidden patterns and make accurate predictions or decisions, making them essential tools in fields like healthcare, finance, transportation, and cybersecurity.

Another crucial advantage of ML is its ability to adapt and improve over time as more data becomes available. Unlike traditional rule-based systems that require constant manual updates, ML models can retrain and adjust themselves to new data, trends, or anomalies, ensuring that the system stays relevant and effective. For example, in fraud detection, ML algorithms can evolve as fraud tactics change, providing a stronger defense compared to static rules that may become outdated. This adaptability makes ML particularly powerful in dynamic, real-time environments where traditional programming methods fall short.

In addition, ML empowers the automation of complex tasks that were previously dependent on human expertise and intuition. From voice recognition in virtual assistants to autonomous driving, ML algorithms can process vast amounts of unstructured data such as text, images, and audio, which are traditionally challenging for computers to handle. By enabling machines to "learn" from experience and improve their performance over time, ML not only enhances productivity but also opens new frontiers for innovation across industries, creating smarter systems that can make meaningful contributions to society.

Machine Learning Applications

Problems can be solve with ML

ML is used across a wide range of industries and real-world problems in healthcare, finance, natural language processing, computer vision, transportation, manufacturing industry, retail, and cybersecurity.

Below are key categories of problems that can be applied using ML.

Application area	Example use Cases
Healthcare	Disease prediction & diagnosis, medical image analysis, drug discovery
Finance	Fraud detection, credit scoring, algorithmic trading
Retail & e-commerce	Product recommendations, customer segmentation, demand forecasting
Transportation & autonomous systems	Self-driving cars, traffic prediction, route optimization
Natural language processing (NLP)	Chatbots and virtual assistants, sentiment analysis, language translation
Manufacturing & industry	Predictive maintenance, quality control, supply chain optimization
Computer Vision	Facial recognition, object detection, image classification

Problems can't be solve with ML

ML is powerful, but it's not magic. It's a tool for finding patterns in data but has no idea what the patterns mean. Therefore it is not a substitute for human reasoning, creativity, or ethical judgment.

Below are key categories of problems that cannot be solved with ML due to inherent limitations, regardless of data or computational advancements.

- Problems with insufficient or poor-quality data: ML relies heavily on data. If data is scarce, noisy, biased, or unrepresentative, models fail to generalize. For example, predicting rare events with limited historical data (*e.g.*, catastrophic asteroid impacts, spread of pandemic) is unreliable.

- Problems requiring reasoning, understanding, or deep logic. ML models approximate patterns but don't understand them. They lack reasoning and common sense unless explicitly designed (*e.g.*, symbolic AI).
- Problems that involve subjective judgments or value-based decisions. ML models don't "know" what's right or wrong – they reflect patterns in the data, including biases.
- Problems outside of distribution generalization. A model trained on photos of cats can't accurately classify dogs if it never saw dogs. ML models interpolate between known data. They struggle with novel scenarios far outside the training set.

Problems can be, but shouldn't be solved with ML

There are many problems where ML (DL) could technically be applied, but shouldn't be – either because of the simplicity of the problem or due to ethical, practical, or societal concerns.

- Tasks for modelling well defined systems, where the equations governing them are known and understood.
- Problems at high-stakes domains with unacceptable error rates: ML can predict outcomes in fields like medical diagnosis or aviation safety, but even small errors can lead to catastrophic consequences. Over-reliance on ML without human oversight risks lives when models fail in edge cases.
- Privacy-sensitive applications: ML can analyze personal data (*e.g.*, health records, browsing habits) to predict behaviors, but using it for invasive profiling, surveillance, or targeted manipulation (*e.g.*, hyper-personalized propaganda) raises serious privacy and autonomy concerns.
- Reinforcing harmful social norms: ML can optimize systems like targeted advertising or content recommendation, but doing so can amplify harmful behaviors (*e.g.*, echo chambers, misinformation, or addiction to social media) if not carefully regulated.

Keypoints

- Machine learning focuses on building systems that can learn patterns from data **without being explicitly programmed**.
- The relationship between artificial intelligence, machine learning, and deep learning.
- The importance of machine learning in daily work and lives.
- Examples of real-world problems that can, cannot, or can but shouldn't, be solved using machine learning.

Fundamentals of Machine Learning

Objectives

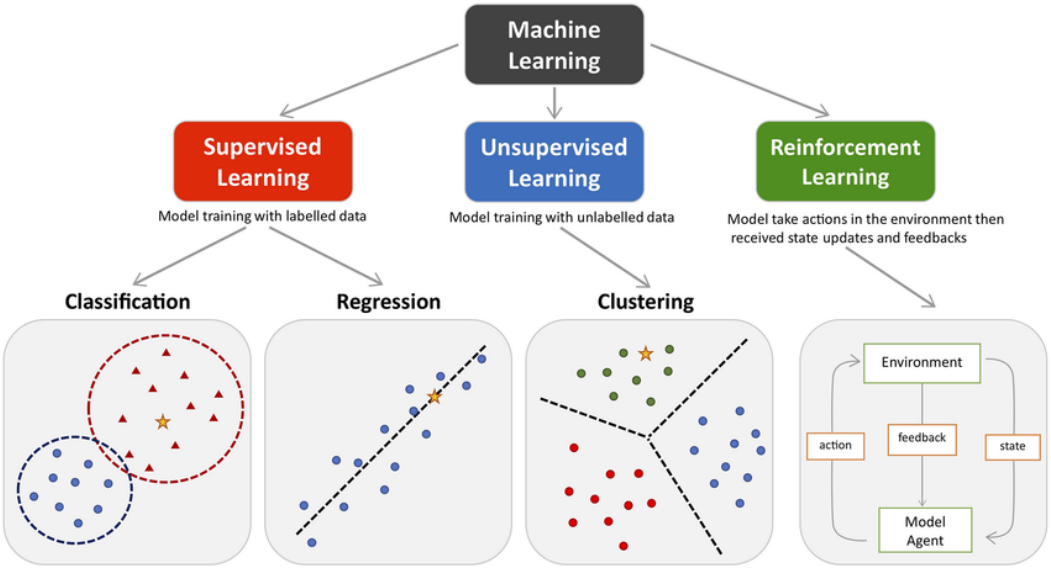
- Describe the representative types of machine learning (supervised, unsupervised, semi-supervised, reinforcement learning).
- Explain the general workflow of a machine learning project.
- Introduce representative machine learning libraries and discuss their pros and cons.

Instructor note

- 25 min teaching
- 0 min exercising

Types of Machine Learning

Machine learning (ML) can be broadly categorized into three main types depending on how the models learn from input data and the nature of the input data they process.



Three main types of machine learning. Main approaches include classification and regression under the supervised learning and clustering under the unsupervised learning. Reinforcement learning enhance the model performance by interacting with environment. Coloured dots and triangles represent the training data. Yellow stars represent the new data which can be predicted by the trained model. This figure was taken from the paper Machine Learning Techniques for Personalised Medicine Approaches in Immune-Mediated Chronic Inflammatory Diseases: Applications and Challenges .

Supervised learning

In supervised learning, the model is trained on a labeled dataset, where each input is paired with a corresponding output (label). The goal is to learn a mapping from inputs to outputs to make predictions on new, unseen data.

Supervised learning has two subtypes: **Classification** (predicting discrete categories) and **Regression** (predicting continuous values).

Here are representative examples of these two subtypes in real-word problems:

- **Classification**: email spam detection (spam/ham), image recognition (cat/dog), medical diagnosis (disease/no disease).
- **Regression**: house price prediction, weather forecasting.

Unsupervised learning

In unsupervised learning, the model works with unlabeled data, identifying patterns, structures, or relationships within the data without explicit guidance on what to predict.

Unsupervised learning also has two subtypes: **Clustering** (grouping similar data points together) and **Dimensionality Reduction** (simplifying data by reducing features while preserving important information)

Representative examples of these two subtypes in real-word problems:

- **Clustering**: customer segmentation in marketing (grouping users by behavior), image segmentation (grouping similar pixels).
- **Dimensionality Reduction**: compressing high-dimensional data (*e.g.*, reducing image features for faster processing), anomaly detection.

Reinforcement learning

The model (agent) learns by interacting with an environment. It takes actions, receives feedback (rewards or penalties), and learns a strategy (policy) to maximize long-term rewards.

Representative examples of reinforcement learning in real-word problems: game-playing AI (*e.g.*, AlphaGo), robot navigation, autonomous driving.

Other subtypes

In addition to supervised and unsupervised learning, there are other important paradigms in ML.

- **Semi-supervised learning** bridges the gap between supervised and unsupervised learning by using a small amount of labeled data together with a large amount of unlabeled data, helping models learn more effectively when labeling is expensive or time-consuming (*e.g.*, medical image analysis).
- **Self-supervised learning** is a form of unsupervised learning where the model generates its own labels from the data – typically for pretraining models on tasks like image or language understanding, enabling them to learn robust representations without explicit labels (*e.g.*, predicting the next word in a sentence, and filling in missing image patches)
- **Transfer learning** involves applying knowledge from a pretrained model, trained on a large, general dataset, to a new, related task, significantly reducing training time and data requirements (*e.g.*, fine-tuning a speech recognition model for a new dialect).

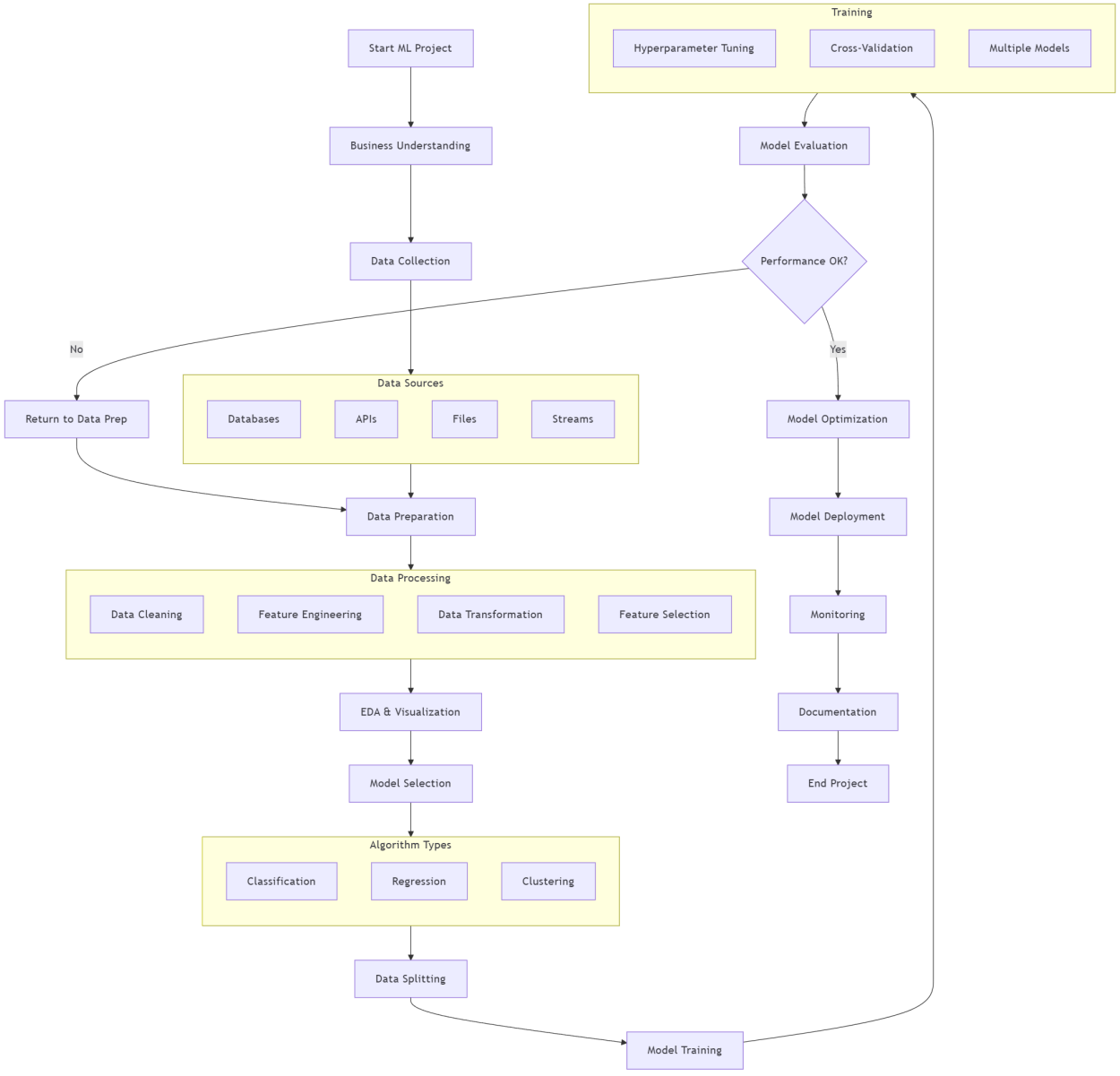
These techniques expand the capabilities and versatility of ML across data-limited or computationally constrained environments.

Machine Learning Workflow

What is a workflow for ML?

A ML workflow is a structured approach for developing, training, evaluating, and deploying ML models. It typically involves several key phases, including data collection, preprocessing, model training and evaluation, and finally, deployment to production.

Here is a graphical representation of ML workflow, and a concise overview of the key steps are described below.



Problem definition and project setup

Problem Definition is the first and most critical phase of any ML project. It sets the direction, scope, and goals for the entire project.

- We should understand the problem domain: what is the real-world problem we are trying to solve? are we predicting, classifying, or grouping data? (*e.g.*, predict house prices, detect spam emails, cluster customers).
- We should determine if ML is the appropriate solution for the problem.
- We then should identify the expected outputs: what will the ML model produce? (*e.g.*, a number, a label, or a probability).
- We define the type of ML task (*e.g.*, classification and regression tasks for supervised learning, clustering, dimensionality reduction for unsupervised learning, and decision-making tasks for reinforcement learning).

Project Setup is to set up the programming/development environment for the project.

- Hardware requirements (CPU, SSD, GPU, cloud platforms, *etc.*).
- Software requirements (programming languages and libraries, ML (DL) frameworks, and development tools, IDEs, Git/Docker, *etc.).
- Project structure: organize the project for clarity and scalability.

A typical ML project structure looks like this:

```
ML_Project/
├── data/                # raw and processed data
│   ├── raw/             # original, unprocessed data
│   ├── processed/       # cleaned, preprocessed data
├── notebooks/           # jupyter notebooks for EDA & modeling
├── src/                 # source code
│   ├── utils/           # utility functions (*e.g.*, metrics, logging)
│   ├── preprocessing.py  # data cleaning script
│   └── train.py         # model training script
├── models/              # trained model files (*e.g.*, .pkl, .h5)
├── tests/               # unit and integration tests
├── README.md            # project overview and setup instructions
├── requirements.txt     # project dependencies
├── config.yaml          # configuration file for hyperparameters and paths
```

Data collection and preprocessing

In ML, data collection and preprocessing are crucial steps that significantly affect the performance of a model. High-quality, well-processed data leads to better predictions, while poor data can result in unreliable models.

- **Data collection**: Gather the necessary data from various sources (*e.g.*, databases, APIs (twitter, linkedin, *etc.*), or manual collection), and ensure that data is representative and sufficient for the problem.
- **Data preprocessing**: Clean and prepare data by handling missing values (drop, impute, or predict), removing duplicates or irrelevant data, fixing inconsistencies (*e.g.*, “USA” vs. “United States”), normalizing/scaling features, encoding categorical variables, and addressing outliers, and other data quality issues.

- **Exploratory data analysis** (EDA): Analyze data to uncover distributions, correlations, patterns, anomalies, and insights using visualizations and statistical methods. This helps in feature selection and understanding data distribution.
- **Feature engineering**: Create or select relevant features to improve model performance. This may involve dimensionality reduction (*e.g.*, PCA (principal component analysis)) or creating new features based on domain knowledge.
- **Data splitting**: Divide the dataset into training, validation, and test sets (*e.g.*, 70-15-15 split) to evaluate model performance and prevent overfitting.

Model selection and training

Model Selection and Training refer to the process of choosing an appropriate model architecture and training it to learn patterns from data to solve a specific task. It involves selecting the appropriate algorithms (*e.g.*, linear/logistic regression, decision trees, neural networks, Gradient Boosting) based on the problem type, configuring its hyperparameters, and optimizing its parameters using training data to minimize error or maximize performance metrics.

Model evaluation and assessment

Model evaluation and assessment in ML refers to the process of measuring and analyzing a model's performance to determine its effectiveness in solving a specific task. It involves using metrics and techniques to quantify how well the model generalizes to unseen data, identifies patterns, and meets desired objectives, typically using a test dataset separate from the training data.

Below are common evaluation metrics by task types:

Task types	Evaluation metrics
Classification	Accuracy, precision, recall, F1-score, ROC-AUC, *etc.*
Regression	Mean Squared Error (MSE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), R-squared, *etc.*
Clustering	Silhouette score, Davies-Bouldin index, Calinski-Harabasz index
Ranking	Mean Reciprocal Rank (MRR), Normalized Discounted Cumulative Gain (NDCG)
NLP or generative tasks	BLEU, ROUGE, perplexity (often overlaps with deep learning)

Here are representative techniques and processes for the assessment:

- **Train-validation-test split**: Divide data into training (model learning), validation (hyperparameter tuning), and test (final evaluation) sets to prevent overfitting.
- **Cross-validation**: Use k-fold cross-validation to assess model stability across multiple data subsets.
- **Confusion matrix**: For classification, visualize true positives, false negatives, *etc.*

- **Learning curves**: Plot training *vs.* validation performance to diagnose underfitting or overfitting.
- **Comparison with baselines**: Comparing model performance against simple baselines (*e.g.*, random guessing, linear models) to ensure meaningful improvement.
- **Robustness testing**: Evaluate performance under noisy, adversarial, or out-of-distribution data.
- **Fairness and bias analysis**: Assess model predictions for fairness across groups (*e.g.*, demographics).

Hyperparameter tuning

Hyperparameter tuning is the process of optimizing the settings (hyperparameters) of a model that are not learned during training but significantly affect its performance. These include parameters like learning rate, number of hidden layers, or batch size, which control the model's behavior and training process.

The goal of this process is to find the best combination of hyperparameters that maximizes performance metrics (*e.g.*, accuracy, precision) on a validation set.

Model deployment, monitoring, and improvement

Model deployment, monitoring, and improvement refer to the processes involved in taking a trained ML model from development to production, ensuring it performs effectively in real-world applications, and continuously enhancing its performance.

- **Model deployment** indicates an integration of a trained model into a production environment (APIs or cloud platforms) where it can make predictions or decisions on new, unseen data.
- Once deployed, the model's performance must be continuously tracked to ensure it remains accurate and reliable over time, which is termed as **model monitoring**.
- As the models degrade over time, so continuous improvement is necessary. **Model improvement** involves updating or retraining the model to maintain or enhance its performance based on monitoring insights or new data.

Machine Learning Libraries

Scikit-learn

Scikit-learn is a widely-used, open-source Python library designed for **classical machine learning**, offering a variety of algorithms and tools for for tasks, such classification, regression, clustering, and dimensionality reduction. It supports supervised learning (*e.g.*, SVM (support vector machine), decision trees, random forests), unsupervised learning (*e.g.*, k-means, PCA (principal component analysis)), and semi-supervised learning, with robust tools for data preprocessing, model evaluation, and hyperparameter tuning via `GridSearchCV`. Built on NumPy, SciPy, and Matplotlib, it is designed for ease of use, making it ideal for beginners and rapid prototyping. Scikit-Learn excels in handling small to medium-sized datasets and

includes utilities for data preprocessing, model evaluation, hyperparameter tuning, and pipeline construction. However, it lacks support for DL and GPU acceleration, limiting its scalability for large datasets or complex neural network tasks.

Keras

Keras is a high-level neural networks API that simplifies the process of building and training DL models. Originally an independent library, Keras is now tightly integrated with TensorFlow as its official high-level interface (but also usable standalone), offering an accessible way to experiment with DL without sacrificing performance. Keras provides user-friendly abstractions for layers, models, loss functions, and optimizers, allowing users for quick prototyping of neural networks for tasks like image classification, text generation, and time series forecasting with minimal code. Keras abstracts away much of the complexity of TensorFlow while retaining flexibility, making it ideal for beginners and those who need fast experimentation.

TensorFlow

Developed by Google, **TensorFlow** is a powerful open-source library primarily for DL but versatile enough for a broad range of ML tasks. It provides a flexible ecosystem for building complex models, including neural networks for computer vision, natural language processing, and time series analysis. TensorFlow supports distributed computing across CPUs, GPUs, and TPUs, making it suitable for both research and production at scale. Its robust features, such as TensorBoard for visualization, TensorFlow Serving for model deployment, and TensorFlow Lite for mobile inference, make it a comprehensive framework for end-to-end ML development. TensorFlow's high-level Keras API simplifies model building, while its low-level operations provide flexibility for advanced research. TensorFlow is well-suited for tasks like image recognition, natural language processing (NLP), and reinforcement learning, though its complexity can pose a steeper learning curve for beginners compared to alternatives like PyTorch.

PyTorch

Developed by Facebook's AI Research Lab (FAIR), PyTorch is auser-friendly and open-source DL library that has gained significant popularity in academia and industry. Known for its intuitive design and "define-by-run" (eager execution) approach, PyTorch allows developers to build, train, and debug models in a flexible and interactive manner. Its strong support for GPU acceleration and extensive ecosystem-ranging from computer vision (TorchVision) to NLP (TorchText) and audio (TorchAudio) – make it an excellent choice for cutting-edge DL research and production. Popular in academia and increasingly in industry, PyTorch excels in rapid prototyping and experimentation but is less optimized for production deployment compared to TensorFlow. Its active community and support for GPU acceleration make it a favorite for cutting-edge ML and DL research.

XGBoost & LightGBM

XGBoost (Extreme Gradient Boosting) and **LightGBM** (Light Gradient Boosting Machine) are high-performance gradient boosting libraries that have become go-to solutions for structured data problems, such as tabular datasets. Both libraries implement optimized gradient boosting algorithms that deliver fast training speeds, high accuracy, and scalability to large datasets. XGBoost is known for its robustness and versatility, while LightGBM offers further speed and memory efficiency through histogram-based algorithms and leaf-wise growth strategies. These libraries have become essential tools for data scientists working with structured data, outperforming traditional models in many real-world scenarios.

Hugging Face Transformers

Hugging Face Transformers is a cutting-edge library that provides access to state-of-the-art pre-trained models for NLP tasks and computer vision, including text classification, translation, summarization, and question answering. The library's pre-trained models and tokenizers simplify NLP workflows by enabling rapid experimentation with large language models, and in addition, this library supports both TensorFlow and PyTorch backends, integrating with datasets via Hugging Face's datasets library, and has a vibrant community contributing to its continuous development.

FastAI

FastAI is a high-level DL library built on PyTorch, designed to make AI accessible to a wider audience by simplifying complex tasks. It provides high-level abstractions and best practices out-of-the-box, allowing users to train powerful models with minimal code and optimal defaults. FastAI is particularly well-known for its transfer learning capabilities, enabling quick adaptation of pre-trained models for tasks like image classification and text generation. With its focus on practical usage, education, and strong community support, FastAI is ideal for beginners and practitioners who want to quickly deploy models without deep theoretical expertise.

JAX

JAX, developed by Google, combines NumPy-like syntax with automatic differentiation and GPU/TPU acceleration, making it ideal for high-performance ML research. It enables composable function transformations (gradients, JIT compilation) and scales efficiently across hardware. While not as high-level as TensorFlow or PyTorch, JAX is favored for cutting-edge numerical computing, physics simulations, and advanced neural network research where speed and flexibility are crucial.

These libraries cater to different needs: Scikit-learn for classical ML, TensorFlow and PyTorch for DL and scalability, Keras for simplicity, XGBoost for high-performance tabular data tasks, and Hugging Face for transformer-based applications. The choice of these libraries depends on the task, data type, scalability needs, user expertise, and whether the focus is research, prototyping, or production deployment.

A summary of best features and key strengths of these libraries are summarized below.

Library	Best Feature	Key Strength
Scikit-Learn	Simple and consistent API for classical machine learning tasks (classification, regression, clustering) and small/medium datasets	Seamless integration with NumPy/Pandas and extensive documentation for ease-of-use with wide algorithm support
PyTorch	Dynamic computation graph (define-by-run) for flexible model building and debugging	Flexible, intuitive framework with strong adoption for academic research in DL tasks
TensorFlow	Scalability with GPU/TPU acceleration for complex deep learning models	Excellent ecosystem (Keras, TF Hub, TF-Agents) for production-scale applications
Keras	High-level, user-friendly API for rapid prototyping	Simplifies construction of DL models, making it beginner-friendly and efficient with TensorFlow compatibility for quick model development
XGBoost & LightGBM	Optimized gradient boosting algorithms	Extremely effective for high-performance supervised learning with tabular/structured data
Hugging Face Transformers	Extensive pretrained transformer models for easy fine-tuning	Community-driven ecosystem with user-friendly pipelines for NLP and vision tasks
FastAI	Transfer learning made easy for NLP & vision tasks	Fast prototyping with minimal code and strong performance for applied deep learning
JAX	NumPy + autodiff + GPU/TPU acceleration	Cutting-edge numerical computing, works with PyTorch/TensorFlow via interoperability libraries, but offers lower-level control

Keypoints

- Representative types of machine learning include supervised learning, unsupervised learning, semi-supervised, reinforcement learning, and the other subtypes. Supervised and unsupervised learnings with specific tasks will be covered in this workshop.
- The general workflow of a machine learning project include identification of problems, data collection, data preprocessing and processing, training and evaluating model performance, and fine-tuning model hyperparameters, and finally depolyment.
- Representative machine learning libraries include Scikit-learn, Keras, TensorFlow, PyTorch, *etc.*.

Scientific Data for Machine Learning

Objectives

- Gain an overview of different formats for scientific data.
- Understand common performance pitfalls when working with big data.

- Describe representative data storage formats and their pros and cons.
- Understand data structures for machine learning.

Instructor note

- 30 min teaching
- 20 min exercising

Big Data

💬 Discussion

- How large is the data you are working with?
- Are you experiencing performance bottlenecks when you try to analyse it?

Big Data refers to datasets that are so large, complex, or fast-changing that traditional data processing tools cannot handle them efficiently. It encompasses not only the sheer volume of data but also its variety, velocity, and veracity — often summarized as the **4 Vs** of big data. These datasets can come from numerous sources, including social media, sensor networks, scientific experiments, and transactional systems. The ability to collect and analyze such massive amounts of information allows organizations and researchers to uncover trends, correlations, and insights that would be impossible to detect with smaller datasets.

The emergence of big data has transformed multiple domains, from business analytics and healthcare to climate science and genomics. Advanced computational methods, distributed storage systems, and parallel processing frameworks such as Hadoop and Spark have become essential for managing and analyzing these vast datasets. Efficient handling of big data enables organizations to make data-driven decisions, optimize operations, and identify opportunities for innovation.

In the context of ML, big data provides the raw material that fuels the learning process. Large and diverse datasets allow ML models to capture complex patterns, generalize well to unseen data, and improve predictive performance. Without sufficient and high-quality data, even the most sophisticated algorithms cannot perform effectively. From image recognition to natural language processing, every ML application depends on properly curated datasets for training, validation, and testing. Therefore, data is the backbone of ML as it serves as the foundation for training models to recognize patterns, make predictions, and generate insights. In addition, data determines the applicability and scalability of ML solutions across domains, from scientific research to real-world applications.

In this episode, we will dive into the world of scientific data, examining the various types and forms it can take and understanding how it is organized and stored. We will explore different data storage formats, highlighting representative formats along with their respective advantages and limitations. This will provide a solid foundation for making informed

decisions about how to handle and manipulate data effectively. More importantly, we will focus on the data structures that are commonly used in ML and DL projects. Understanding these structures is essential for efficiently preparing, processing, and feeding data into models, ultimately enabling accurate predictions and insights. By the end of this session, you will have a clear understanding of how scientific data is organized and how it can be structured to support ML and DL workflows.

Understanding Scientific Data

Scientific data refers to any form of data that is collected, observed, measured, or generated as part of scientific research or experimentation. This data is used to support scientific analysis, develop theories, and validate hypotheses. It can come from a wide range of sources, including experiments, simulations, observations, or surveys across various scientific fields.

In general, scientific data can be described ty two terms: **types of data** and **forms of data**. They are related but distinct — types describe the nature of the data, while forms describe the how the data is structured and formatted (and stored, which will be discussed below).

Types of scientific data

Types of scientific data refer to what the data represents. It focuses on the nature or category of the data content.

- **Bit and byte**: The smallest unit of storage in a computer is a **bit**, which holds either a 0 or a 1. Typically, eight bits are grouped together to form a **byte**. A single byte (8 bits) can represent up to 256 distinct values. By organizing bytes in various ways, computers can interpret and store different types of data.
- **Numerical data**: Different numerical data types (*e.g.*, integer and floating-point numbers) require different binary representation. Using more bytes for each value increases the range or precision, but it consumes more memory.
  - For example, integers stored with 1 byte (8 bits) have a range from [-128, 127], while with 2 bytes (16 bits) the range becomes [-32768, 32767]. Integers are whole numbers and can be represented exactly given enough bytes.
  - In contrast, floating-point numbers (used for decimals) often suffer from representation errors, since most fractional values cannot be precisely expressed in binary. These errors can accumulate during arithmetic operations. Therefore, in scientific computing, numerical algorithms must be carefully designed to minimize error accumulation. To ensure stability, floating-point numbers are typically allocated 8 bytes (64 bits), keeping approximation errors small enough to avoid unreliable results.
  - In ML/DL, half, single, and double precision refer to different formats for representing floating-point numbers, typically using 16, 32, and 64 bits, respectively.
    - **Single precision** (32-bit) is commonly used as a balance between computational efficiency and numerical accuracy.

    - **Half precision** (16-bit) offers faster computation and reduced memory usage, making it popular for training large models on GPUs, though it may suffer from lower numerical stability.
    - **Double precision** (64-bit) provides higher accuracy but is slower and more memory-intensive, so it's mainly used when high numerical precision is critical.
    - Many modern frameworks, like TensorFlow and PyTorch, support mixed precision training, combining half and single precision to optimize performance while maintaining stability.
- **Text data**: When it comes to text data, the simplest character encoding is ASCII (American Standard Code for Information Interchange), which was the most widely used encoding until 2008 when UTF-8 took over. The original ASCII uses only 7 bits for representing each character and therefore can encode 128 specified characters. Later, it became common to use an 8-bit byte to store each character, resulting in extended ASCII with support for up to 256 characters. As computers became more powerful and the need for including more characters from other alphabets, UTF-8 became the most common encoding. UTF-8 uses a minimum of one byte and up to four bytes per character. This flexibility makes UTF-8 ideal for modern applications requiring global character support.
- **Metadata**: Metadata encompasses diverse information about data, including units, timestamps, identifiers, and other descriptive attributes. While most scientific data is either numerical or textual, the associated metadata is usually domain-specific, and different types of data may have different metadata conventions. In scientific applications, such as simulations and experimental results, metadata is typically integrated with the corresponding dataset to ensure proper interpretation and reproducibility.

Forms of scientific data

Forms of scientific data refer to how the data is structured or formatted. It focuses on the presentation or shape of the data.

- **Tabular data structure** (numerical arrays) is a collection of numbers arranged in a specific structure that one can perform mathematical operations on. Examples of numerical arrays are scalar (0D), row or column vector (1D), matrix (2D), and tensor (3D), *etc.*
- **Textual data structure** is a format for storing and organizing text-based data. It represents unstructured or semi-structured information as sequences of characters (letters, numbers, symbols, punctuation) arranged in strings.
- **Images, videos, and audio** are forms of scientific data that represent information through visual and auditory formats. Images capture static visual information as pixel arrays, videos combine sequential frames to show temporal changes, and audio encodes sound signals as time-series data for analysis.
- **Graphs and networks** are forms of scientific data that represent relationships between entities as nodes and connections as edges. They are used to model complex systems such as social networks, molecular interactions, and ecological food webs, capturing the structure and connectivity of scientific phenomena.

Data Storage Format

Representative data storage format

When it comes to data storage, there are many types of storage formats used in scientific computing and data analysis. There isn't one data storage format that works in all cases, so choose a file format that best suits your data.

For tabular data, each column usually has a name and a specific data type while each row is a distinct sample which provides data according to each column (including missing values). The simplest way to save tabular data is using the so-called CSV (comma-separated values) file, which is human-readable and easily shareable. However, it is not the best format to use when working with big (numerical) data.

Gridded data is another very common data type in which numerical data is normally saved in a multi-dimensional grid (array). Common field-agnostic array formats include:

- **Hierarchical Data Format** (HDF5) is a high performance storage format for storing large amounts of data in multiple datasets in a single file. It is especially popular in fields where you need to store big multidimensional arrays such as physical sciences.
- **Network Common Data Form version 4** (NetCDF4) is a data format built on top of HDF5, but exposes a simpler API with a more standardised structure. NetCDF4 is one of the most used formats for storing large data from big simulations in physical sciences.
- **Zarr** is a data storage format designed for efficiently storing large, multi-dimensional arrays in a way that supports scalability, chunking, compression, and cloud-readiness.
- There are more file formats like feather, parquet, xarray and npy to store arrow tables or data frames.

Overview of data storage format

Below is an overview of common data formats (✅ for *good*, 🟨 for *ok/depends on a case*, and ❌ for *bad*) adapted from Aalto university's Python for scientific computing.

Name	Human readable	Space efficiency	Arbitrary data	Tidy data	Array data	Long term storage/sharing
Pickle	❌	🟨	✅	🟨	🟨	❌
CSV	✅	❌	❌	✅	🟨	✅
Feather	❌	✅	❌	✅	❌	❌
Parquet	❌	✅	🟨	✅	🟨	✅
npy	❌	🟨	❌	❌	✅	❌
HDF5	❌	✅	❌	❌	✅	✅
NetCDF4	❌	✅	❌	❌	✅	✅

Name	Human readable	Space efficiency	Arbitrary data	Tidy data	Array data	Long term storage/sharing
JSON	✅	❌	🟨	❌	❌	✅
Excel	❌	❌	❌	🟨	❌	🟨
Graph formats	🟨	🟨	❌	❌	❌	✅

Data Structures for ML/DL

ML (and DL) models require numerical input, so we must collect adaquate numerical data before training. For ML tasks, multimedia data like image, audio, or video formats should be converted into tabular data or numerical arrays that ML models can process. This conversion enables models to extract meaningful features, such as pixel intensities, audio frequencies or motion patterns, for tasks like classification or prediction.

Numerical array

Numerical array is a collection of numbers arranged in a specific structure that one can perform mathematical operations on. Examples of numerical arrays are scalar (0D), row or column vector (1D), matrix (2D), and tensor (3D), *etc.*

Python offers powerful libraries like NumPy, PyTorch, TensorFlow, and Dask (parallel Numpy) to work with numerical arrays (0D to *n*D).

```
import numpy as np

# 0D (Scalar)
scalar = np.array(5)

# 1D (Vector)
vector = np.array([1, 2, 3])

# 2D (Matrix)
matrix_2D = np.array([[1, 2], [3, 4]])

# 3D (Matrix)
matrix_3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(matrix_3D.shape)
```

Tensor

In ML and DL, a tensor is a mathematical object used to represent and manipulate multidimensional data. It generalizes scalars, vectors, and matrices to higher dimensions, serving as the fundamental data structure in frameworks like TensorFlow and PyTorch.

Why to use tensors in ML/DL (advantages of Tensor)?

- Generalization of scalars/vectors/matrices: Tensors extend these concepts to any number of dimensions, which is essential for handling complex data like images (3D) and videos (4D+).
- Consistency: Tensors unify data structures across ML/DL frameworks, simplifying model building, training, and deployment.
- Efficient computation: Frameworks like TensorFlow and PyTorch optimize tensor operations for speed (using GPUs/TPUs).
- Neural network representations: Input data (images, text) is converted to tensors.
- Automatic differentiation: Tensors support gradient tracking, which is vital for backpropagation in neural networks.

✍ Tensor Creation and Operations

In Jupyter Notebook we provide a tutorial about Tensor including

- Tensor creation
- Tensor's properties (`shape`, `dtype`, `ndim`)
- Tensor operations
  - indexing, slicing, transposing
  - element-wise operations: addition, subtraction, *etc.*
  - matrix multiplication(`np.dot`, `torch.matmul`)
  - reshaping, flattening, squeezing, unsqueezing
  - reduction operations: sum, mean, max along axes
  - broadcasting: Rules and examples
- Tensors in DL frameworks
  - moving tensors between CPUs and GPUs (suppose that you can access to GPU cards)

Keypoints

- Key characteristics of big data — volume, variety, velocity, and veracity.
- High-quality, large datasets are essential for training effective machine learning models
- Scientific data has different types, including numerical, textual, and multimedia data, and these data can take different forms, such as tabular arrays, grids, and images.
- Scientific data are stored in various formats including CSV, HDF5, and others with their respective advantages and limitations.
- Tensors as a generalization of numerical data in machine learning (deep learning).
- Tensors allow models to efficiently handle complex, multidimensional data such as images, videos, and audio.

Data Preparation for Machine Learning

Objectives

- Provide an overview of data preparation.
- Load the Penguins dataset.
- Use pandas and seaborn to analyze and visualize the data.
- Identify and manage missing values and outliers in the dataset.
- Encode categorical variables into numerical values suitable for machine learning models.

Instructor note

- 30 min teaching
- 20 min exercising

In Episode 2: Fundamentals of Machine Learning, it is clearly shown that data preparation and processing often consume a significant portion of the ML workflow — often more time than the actual model training, evaluation, and optimization. Cleaning, transforming, and structuring raw data into a usable format ensures that algorithms can efficiently extract valuable insights. Additionally, the choice of data formats, such as CSV for simplicity or HDF5 for large-scale datasets, can significantly impact data storage, accessibility, and computational efficiency during both model training and deployment.

In this episode, we will provide an overview of data preparation and introduce available public datasets. Using the Penguins dataset as an example, we will offer demonstrations and hands-on exercises to develop a comprehensive understanding of data preparation including handling missing values and outliers, encoding categorical variables, and other essential preprocessing techniques for ML workflows.

What is Data Preparation

Data preparation refers to the process of cleaning, structuring, and transforming raw data into a structured, high-quality format ready for statistical analysis and ML. It's one of the most critical steps in the ML workflow because high-quality data leads to better model performance. Key procedures include:

- collecting data from multiple sources,
- handling missing values (imputation or removal),
- detecting and treating outliers,
- encoding categorical variables,
- normalizing or scaling features,
- feature selection and feature engineering.

Collecting Data from Multiple Sources

Data preparation begins with collecting raw data from a wide variety of sources, including databases, sensors, APIs, web scraping, surveys, and existing public datasets.

During the data collection process, it is important to ensure consistency and compatibility across all sources. Different sources may have different formats, units, naming conventions, or levels of quality. Careful integration, cleaning, and normalization are required to create a unified dataset suitable for analysis or modeling. Proper documentation of sources and collection methods is also essential to maintain reproducibility and data governance.

Public datasets provide an excellent resource for learning, experimentation, and benchmarking. Some widely used datasets across different domains include:

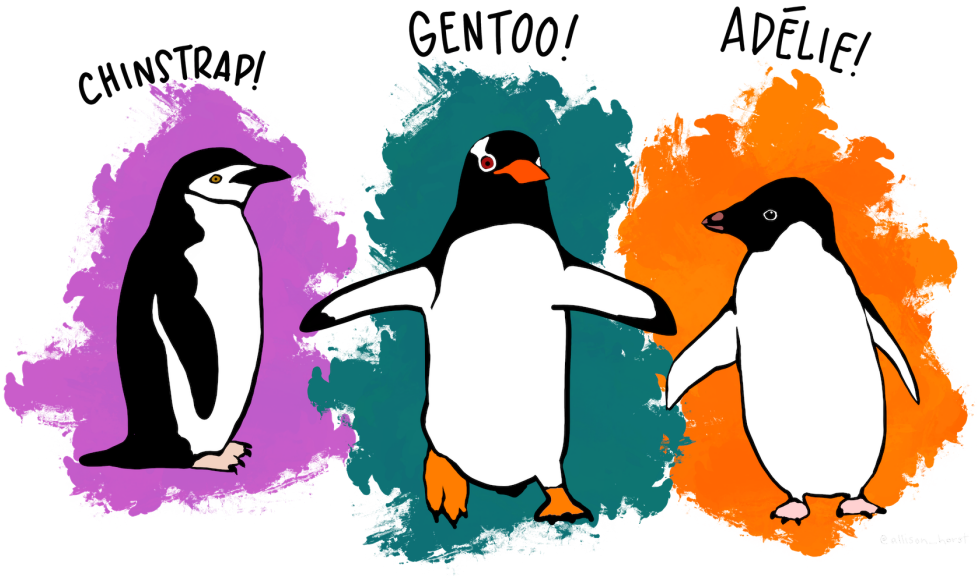
- Tabular datasets: Iris, **Penguins**, Titanic, Boston Housing, Wine, *etc.*
- Image datasets: MNIST, CIFAR-10, CIFAR-100, COCO, ImageNet.
- Text datasets: IMDB Reviews, 20 Newsgroups, Sentiment140.
- Audio datasets: LibriSpeech, UrbanSound8K, ESC-50.
- Video datasets: UCF101, Kinetics, HMDB51.

These datasets are available on platforms like Kaggle, UCI Machine Learning Repository, TensorFlow Datasets, and Hugging Face Datasets, providing accessible resources for practice and innovation.

It should be noted that most of the data available by default is too raw to perform statistical analysis. Proper preprocessing is essential before the data can be used to identify meaningful patterns or to train models for prediction. In the following sections, we use the Penguins dataset as an example to demonstrate essential data preprocessing steps. These include handling missing values, detecting and treating outliers, encoding categorical variables, and performing other necessary transformations to prepare the dataset for ML tasks. Proper preprocessing ensures data quality, reduces bias, and improves the performance and reliability of the models we build.

The Penguins Dataset

The Palmer Penguins dataset is a widely used open dataset in data science and ML education. This dataset contains information on three penguin species that inhabit islands near the Palmer Archipelago in Antarctica: Adelie, Chinstrap, and Gentoo. Each row in the dataset corresponds to a single penguin and records both physical measurements and categorical attributes. The key numerical features include flipper length (mm), culmen length and depth (bill measurements, in mm), and body mass (g). Alongside these, categorical variables such as species, island, and sex are provided.



These data were collected from 2007 - 2009 by Dr. Kristen Gorman with the Palmer Station Long Term Ecological Research Program, part of the US Long Term Ecological Research Network. The data were imported directly from the Environmental Data Initiative (EDI) Data Portal, and are available for use by CC0 license ("No Rights Reserved") in accordance with the Palmer Station Data Policy.

Importing Dataset

Seaborn provides the Penguins dataset through its built-in data-loading functions. We can access it using `sns.load_dataset('penguin')` and then have a quick look at the data (code examples are availalbe in the Jupyter Notebook):

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

penguins = sns.load_dataset('penguins')
penguins
```

Note

If you have your own dataset stored in a CSV file, you can easily load it into Python using Pandas with the `read_csv()` function. This is one of the most common ways to bring tabular data into a DataFrame for further analysis and processing.

Beyond CSV files, Pandas also supports a wide variety of other file formats, making it a powerful and flexible tool for data handling. For example, you can use `read_excel()` to import data from Microsoft Excel spreadsheets, `read_hdf()` to work with HDF5 binary stores, and `read_json()` to load data from JSON files. Each of these formats also has a corresponding method for saving data back to disk, such as `to_csv()`, `to_excel()`, `to_hdf()`, and `to_json()`.

	species	island	bill_length (mm)	bill_depth (mm)	flipper length (mm)	body_mass (g)	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
...	...	...	...	...	...	...	...
339	Gentoo	Biscoe	NaN	NaN	NaN	NaN	NaN
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0	Female
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0	Male
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0	Female
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0	Male

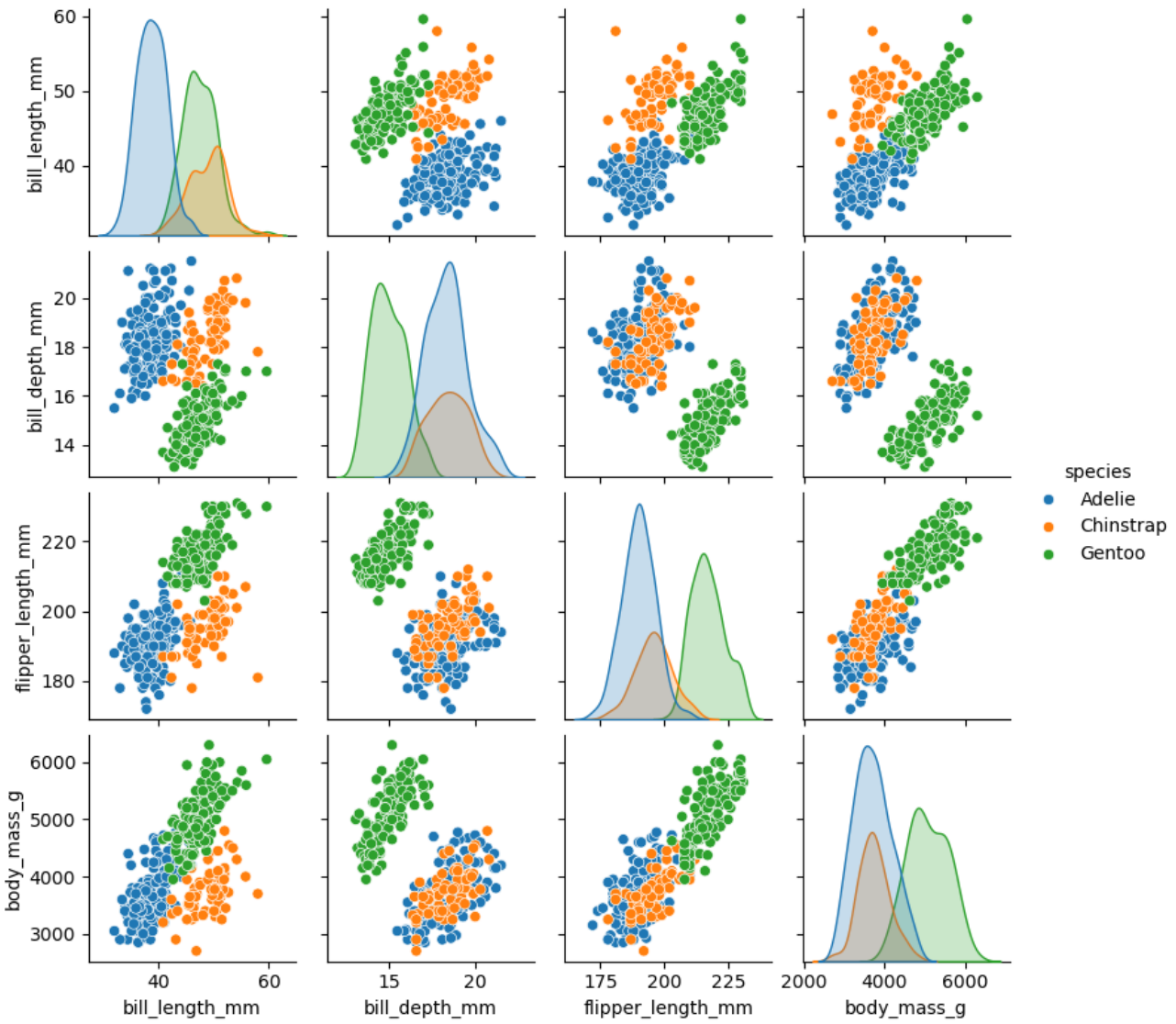
There are seven columns include:

- *species*: penguin species (Adelie, Chinstrap, Gentoo)
- *island*: island where the penguin was found (Biscoe, Dream, Torgersen)
- *bill_length_mm*: length of the bill
- *bill_depth_mm*: depth of the bill
- *flipper_length_mm*: length of the flipper
- *body_mass_g*: body mass in grams
- *sex*: male or female

Looking only at the raw numbers in the `penguins` DataFrame, or even examining the statistical summaries provided by `penguins.info()` and `penguins.describe()`, often does not give us a clear intuition about the patterns and relationships in the data. To truly understand the dataset, we generally prefer to visualize the data, since graphical representations can reveal trends, groupings, and anomalies that may remain hidden in numerical summaries alone.

One nice visualization for datasets with relatively few attributes is the **Pair Plot**, which can be created using `sns.pairplot(...)`. It shows a scatterplot of each attribute plotted against each of the other attributes. By using the `hue='species'` setting for the pairplot the graphs on the diagonal are layered kernel density estimate plots for the different values of the `species` column.

```
sns.pairplot(penguins[["species", "bill_length_mm", "bill_depth_mm",
"flipper_length_mm",
        "body_mass_g"]], hue="species", height=2.0)
```



💬 Discussion

Take a look at the pairplot we created. Consider the following questions:

- Is there any class that is easily distinguishable from the others?
- Which combination of attributes shows the best separation for all 3 class labels at once?
- For pairplot with `hue="sex"`, which combination of features distinguishes the two sexes best?
- What about the one with `hue="island"`?

Handling Missing Values

Upon loading the Penguins dataset into a pandas DataFrame, the initial examination reveals the presence of `NaN` (Not a Number) values within several rows (highlighted in the Jupyter notebook). These placeholders explicitly indicate missing or unavailable data for certain measurements, such as bill length or the sex of particular penguins.

Recognizing these missing values is an important first step, as they must be properly handled before performing any data analysis.

```
penguins_test = pd.concat([penguins.head(5), penguins.tail(5)])
penguins_test.style.highlight_null(color = 'red')
```

Numerical features

For numerical features such as bill length, bill depth, flipper length, and body mass, several strategies can be applied. A straightforward approach is to **remove any rows with missing values**, but this is often wasteful and reduces the sample size.

A more effective method is imputation: replacing missing numerical values with a suitable estimate. Common choices include the `mean` or `median` of the feature, depending on the distribution. Before applying imputation to handle missing numerical values, it is important to first identify where the NaN values occur in the dataset. We can

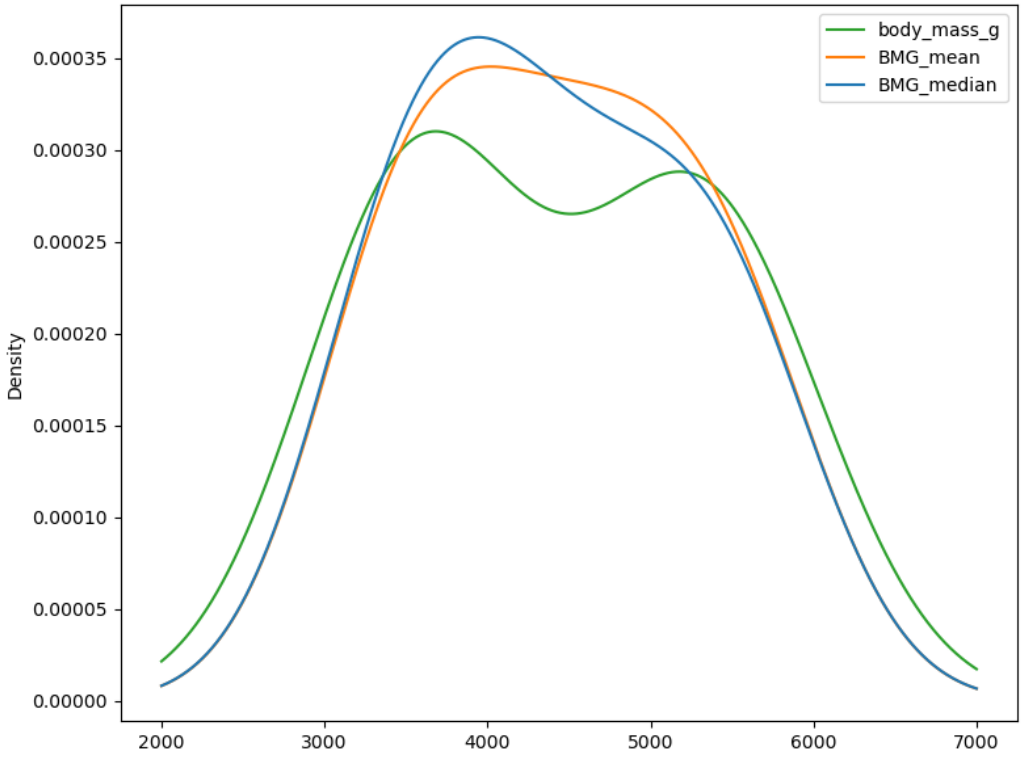
- run `penguins_test.style.highlight_null(color = 'red')`, where NaN values are hightlighed in the output,
- run `print(penguins_test.info(), '\n')`, which provide the number of non-null entries in each column,
  - By comparing the number of non-null entries with the total number of rows, we can quickly identify which features have missing values and how severe the issue is. For example, if the column sex has fewer non-null values than the total number of penguins, we know that sex information is missing for some individuals.
- run `print(penguins_test.isnull().mean())`, which computes the fraction of missing values in each column, giving us a normalized view of missingness across the dataset.
  - unlike `.info()`, which only shows counts, this method highlights the relative proportion of missing values, which is particularly helpful when working with large datasets.
  - For instance, the `.isnull().mean()` reports that 0.2 (20%) of the entries in `body_mass_g` are missing, we can decide whether to impute those values or simply drop the rows without significantly reducing the dataset size.

The next step is to calculate the `mean` and `median` values for the numerical features. To illustrate this process, we can take the `body_mass_g` feature as an example.

```
body_mass_g_mean = penguins_test.body_mass_g.mean()
body_mass_g_median = penguins_test.body_mass_g.median()

print(f"  mean value of body_mass_g is {body_mass_g_mean}")
print(f"median value of body_mass_g is {body_mass_g_median}")

#   mean value of body_mass_g is 4431.25
# median value of body_mass_g is 4325.00
```

Rather than directly replacing the missing values, we concatenate new columns containing the `mean` and `median` values for the `body_mass_g` feature to the end of the Penguins dataset, and than visualize the distribution of this feature.

```
penguins_test['BMG_mean'] = penguins_test.body_mass_g.fillna(body_mass_g_mean)
penguins_test['BMG_median'] = penguins_test.body_mass_g.fillna(body_mass_g_median)
```



✍ Exercise

How to mutate the missing values with `mean` or `median` values in place for all numerical values (code examples are availalbe in the Jupyter Notebook).

- for one numerical feature like `bill_length_mm`?
- for all numerical features in the Penguins dataset?

✔ Solution

- 1. using the following script

```
penguins_test2.loc[penguins_test2["bill_length_mm"].isnull(), "bill_length_mm"]
= penguins_test2["bill_length_mm"].mean()
```

- 2 using the following code

```
# first, select only the numerical columns
numerical_cols = penguins_test2.select_dtypes(include=['number']).columns

# second, find which of these numerical columns have any missing values
numerical_cols_with_nulls =
numerical_cols[penguins_test2[numerical_cols].isnull().any()]

for col in numerical_cols_with_nulls:
        # calculate the mean for the specific column
        col_mean = penguins_test2[col].mean()
        # use `.loc` to replace NaNs only in that specific column with its own
mean
        penguins_test2.loc[penguins_test2[col].isnull(), col] = col_mean
```

In certain scenarios, imputing missing data with values at the **end of distribution** (EoD) of a variable can be a considered strategy. This approach offers the advantage of computational speed and can theoretically capture the significance of missing entries. Typically, the EoD is calculated as an extreme value such as `mean + 3*std`, where the `mean` and `std` of the feature can be obtained using the `.describe()` method.

However, as demonstrated in the Penguins dataset tutorial, this type of imputation often generates unrealistic values and distorts the original distribution, particularly for features like `body_mass_g`. Consequently, it can lead to biased analyses and should be used with caution.

Categorical features

For categorical features such as sex, the approach differs.

- One simple method is to replace missing categories with the most frequent value (`mode`), which assumes the missing value follows the majority distribution.

```
penguins_test.sex.mode()

penguins_test.fillna({'sex': penguins_test['sex'].mode()[0]}, inplace=True)
```

- Alternatively, missing values can be treated as a separate category, labeled for example as “Unknown” or “Missing” which allows models to learn if missingness itself carries information.

```
penguins_test.fillna({'sex': "Missing"}, inplace=True)
```

- Another option is to apply model-based imputation, where missing categorical values are predicted from other features using classification algorithms.

For all ML tasks we will perform in the following episodes, we adopt a straightforward approach of removing any rows that contain missing values (`penguins_test.dropna()`). This ensures that the dataset is complete and avoids potential errors or biases caused by NaN entries. Although more sophisticated imputation methods exist, dropping rows is a simple and effective strategy when the proportion of missing data is relatively small.

Note

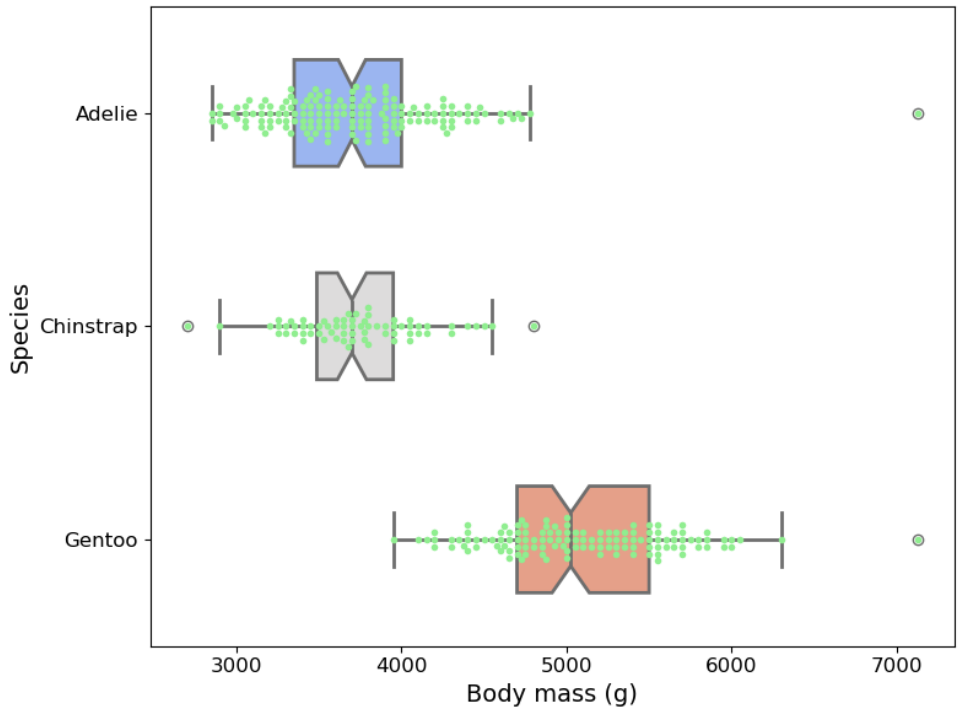
For the other dataset, the strategy for handling missing values is not one-size-fits-all; it depends heavily on whether the missing data is numerical or categorical and the underlying mechanism causing the data to be missing. Ignoring these missing entries, such as by simply dropping the affected rows, may introduce significant bias, reduce statistical power, and ultimately lead to inaccurate or misleading conclusions. After any imputation, it is essential to perform sanity checks to ensure the imputed values are plausible and to document the methodology transparently. Properly handling missing data in this way transforms an incomplete dataset into a robust and reliable foundation for generating accurate insights and building powerful predictive models.

Handling Outliers

Outliers are values that are too far from the rest of observations in columns. For instance, if the body mass of most of penguins in the dataset varies between 3000-6000 g, an observation of 7500 g will be considered as an outlier since such an observation occurs rarely.

We obtain the EoD value of the `body_mass_g` feature and then check if this value is the outlier for this feature.

```
penguins['body_mass_g'].mean() + 3 * penguins_test['body_mass_g'].std()

# EoD value = 7129.0199920504665
```



There are several approaches to identify outliers, and one of the most commonly used methods is the **Interquartile Range (IQR)** method. The IQR measures the spread of the middle 50% of the data and is calculated by subtracting the first quartile (25th percentile, Q1) from the third quartile (75th percentile, Q3). Once the IQR is obtained, we can determine the boundaries for detecting outliers. The lower limit is defined as Q1 minus 1.5 times the IQR, and the upper limit is defined as Q3 plus 1.5 times the IQR.

```
print(f"25% quantile = {penguins_test_BMG_outlier["body_mass_g"].quantile(0.25)}")
print(f"75% quantile = {penguins_test_BMG_outlier["body_mass_g"].quantile(0.75)}\n")

IQR = penguins_test_BMG_outlier["body_mass_g"].quantile(0.75) -
penguins_test_BMG_outlier["body_mass_g"].quantile(0.25)
lower_bmg_limit = penguins_test_BMG_outlier["body_mass_g"].quantile(0.25) - (1.5 * IQR)
upper_bmg_limit = penguins_test_BMG_outlier["body_mass_g"].quantile(0.75) + (1.5 * IQR)

print(f"lower limt of IQR = {lower_bmg_limit} and upper limit of IQR =
{upper_bmg_limit}")

# 25% quantile = 3550.00
# 75% quantile = 4781.25
# lower limt of IQR = 1703.125 and upper limit of IQR = 6628.125
```

Any data points that fall below the lower limit or above the upper limit are considered outliers, and these points are subsequently removed from the dataset.

```
penguins_test_BMG_outlier[penguins_test_BMG_outlier["body_mass_g"] > upper_bmg_limit]

penguins_test_BMG_outlier[penguins_test_BMG_outlier["body_mass_g"] < lower_bmg_limit]

penguins_test_BMG_outlier_remove_IQR =
penguins_test_BMG_outlier[penguins_test_BMG_outlier["BMG_eod"] < upper_bmg_limit]
```

Note

There are four main techniques for handling outliers in a dataset:

- Remove outliers entirely — This approach simply deletes the rows containing outlier values, which can be effective if the outliers are errors or rare events that are not relevant to the analysis.
- Treat outliers as missing values — Outliers can be replaced with NaN and then handled using imputation methods described in previous sections, such as replacing them with the mean, median, or mode.
- Apply discretization or binning — By grouping numerical values into bins, outliers are included in the tail bins along with other extreme values, which reduces their impact while preserving the overall structure of the data.
- Cap or censor outliers — Extreme values can be limited to a maximum or minimum threshold, often derived from statistical techniques such as the IQR or standard deviation limits. This approach reduces the influence of outliers without completely removing them from the dataset.

✍ The mean–standard deviation approach

Instead of using the IQR method, the upper and lower thresholds for detecting outliers can also be calculated with the mean-std deviation approach.

In this exercise (code examples are availalbe in the Jupyter Notebook), you will

- Compute the `mean` and `std` of the `body_mass_g` feature.
- Calculate the upper and lower limits for outlier detection using the formulas.

$$\begin{aligned}lower\_limit = mean - 3.0 \times std\\upper\_limit = mean + 3.0 \times std\end{aligned}$$

- Identify the outliers and replace them with either the `mean` or the `median` values of the `body_mass_g` feature.

✔ Solution

- `mean = penguins_test_BMG_outlier["body_mass_g"].mean()` and `std = penguins_test_BMG_outlier["body_mass_g"].std()`
- `lower_bmg_limit = mean - (3.0 * std)` and `upper_bmg_limit = mean + (3.0 * std)`
- determination of outliers
  - `penguins_test_BMG_outlier[penguins_test_BMG_outlier["body_mass_g"] > upper_bmg_limit]` and `penguins_test_BMG_outlier[penguins_test_BMG_outlier["body_mass_g"] > upper_bmg_limit]`
- imputation outliers with `mean` or `median`

- `penguins_test_BMG_outlier.loc[penguins_test_BMG_outlier["body_mass_g"] < lower_bmg_limit, "body_mass_g"] = mean`
- `penguins_test_BMG_outlier.loc[penguins_test_BMG_outlier["body_mass_g"] < lower_bmg_limit, "body_mass_g"] = penguins_test_BMG_outlier["body_mass_g"].median()`

Encoding Categorical Variables

In the previous sections, we adopted a straightforward approach to handling missing values by simply removing any rows that contained NaN values, whether they were in numerical or categorical features. While this step gives us a cleaner dataset, it is not sufficient on its own to proceed with ML tasks.

The reason is that most ML algorithms are designed to work only with numerical data. They cannot directly process textual or symbolic categories such as species, island, or sex in the Penguins dataset. To make these categorical variables usable in modeling, we need to encode categorical variables into a numerical format while preserving their information.

There are two widely used encoding techniques: **One-hot encoding (OHE)** and **Label Encoding**.

One-hot encoding

The OHE method creates a new binary column for each category in a feature. For example, the species feature with values {Female, Male, and NaN} would be transformed into three new columns, each indicating the presence (1) or absence (0) of that category for a given row.

```
from sklearn.preprocessing import OneHotEncoder

penguins_sex = penguins[["species", "island", "sex"]].head(10)

encoder = OneHotEncoder(sparse_output=False)  # `sparse_output=False` to get a dense array
encoded = encoder.fit_transform(penguins_sex[['sex']])

encoded = pd.DataFrame(encoded, columns=["Female", "Male", "NaN"])

penguins_sex_onehotencoding = pd.concat([penguins_sex, encoded], axis=1)
penguins_sex_onehotencoding
```

	species	island	sex	Female	Male	NaN
0	Adelie	Torgersen	Male	0.0	1.0	0.0
1	Adelie	Torgersen	Female	1.0	0.0	0.0
2	Adelie	Torgersen	Female	1.0	0.0	0.0
3	Adelie	Torgersen	NaN	0.0	0.0	1.0
4	Adelie	Torgersen	Female	1.0	0.0	0.0
5	Adelie	Torgersen	Male	0.0	1.0	0.0
6	Adelie	Torgersen	Female	1.0	0.0	0.0
7	Adelie	Torgersen	Male	0.0	1.0	0.0
8	Adelie	Torgersen	NaN	0.0	0.0	1.0
9	Adelie	Torgersen	NaN	0.0	0.0	1.0

Warning

OHE works well when the number of categories is small and when categories are unordered. However, it can lead to very large datasets if a feature contains many unique categories (high cardinality).

Label encoding

Label encoding transforms a categorical feature into a single numerical column by assigning a unique integer to each category in a feature. For example, the sex feature with values {Female, Male, NaN} could be encoded as Female = 0, Male = 1, and missing values handled separately or imputed beforehand (in our case, NaN is encoded as NaN = 2).

Unlike one-hot encoding, which creates multiple columns, label encoding produces only one column, making it more memory-efficient. However, it introduces an artificial ordinal relationship between categories (*e.g.*, implying Dream > Biscoe), which may not be meaningful and can affect algorithms that assume numeric order matters, such as linear regression or distance-based methods.

```
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
encoded = encoder.fit_transform(penguins_sex['sex'])
encoded = pd.DataFrame(encoded, columns=["sex_LE"])

penguins_sex_labelencoding = pd.concat([penguins_sex, encoded], axis=1)
```

Beyond these two common methods, there are several other encoding strategies, especially useful for more complex datasets:

- **Ordinal Encoding**: A variation of label encoding designed specifically for variables that have a natural and meaningful order (*e.g.*, Small = 0, Medium = 1, Large = 2).
- **Binary Encoding**: Converts categories into binary code, achieving a good compromise between one-hot and label encoding for high-cardinality data.
- **Target Encoding** (Mean Encoding): Replaces each category with the average value of the target variable for that category. It is powerful but prone to overfitting if not carefully implemented.
- **Frequency Encoding**: Replaces categories with their frequency of occurrence in the dataset. This can be useful for dealing with high-cardinality features.

The choice of encoding method is a consequential modeling decision. It should be guided by the type of categorical data (nominal *vs.* ordinal), the number of unique categories, the type of ML algorithm being used. A proper encoding ensures that categorical features are accurately represented in numerical form, allowing algorithms to learn patterns effectively without introducing bias or noise.

The `get_dummies()` function in Pandas

In addition to OHE and LE, we can also use the `.get_dummies()` in Pandas to handle categorical variables by converting them into a one-hot encoded (dummy variable) format.

```
dummy_encoded = pd.get_dummies(penguins_sex['sex']).astype(np.int8)
penguins_sex_dummyencoding = pd.concat([penguins_sex, dummy_encoded], axis=1)
```

Note

This function ignores `NaN` values by default (it simply leaves them out).

Feature Engineering

Feature engineering is a part of the broader data processing pipeline in ML workflows. It involves using domain knowledge to select, modify, or create new features — variables or attributes — from existing data to help algorithms better understand patterns and relationships.

Feature engineering is crucial because the quality of features directly impacts a model's predictive power. Well-crafted features can simplify complex patterns, reduce overfitting, and improve model interpretability, leading to better generalization and performance on unseen data. By tailoring features to the problem at hand, feature engineering bridges the gap between raw data and actionable insights, often making the difference between a mediocre and a high-performing model.

Feature engineering is closely related to data preprocessing, but they serve different purposes.

- Data processing (or data preprocessing) is about cleaning and preparing data — handling missing values, removing duplicates, correcting data types, and ensuring consistency. This step makes the data **usable**.
- Feature engineering, on the other hand, comes after basic processing and focuses on improving the predictive power of dataset.
- In essence, **data preprocessing ensures data quality**, while **feature engineering enhances data value** for ML models.
- Both are essential steps in building effective and accurate predictive systems.

Keypoints

- Data Preparation is a critical, foundational step in the machine learning workflow.
- Introduction to the Palmer Penguins dataset, and demonstration of loading the dataset into a pandas DataFrame for analysis.
- Statistical analysis and Visualization of the Penguins dataset using Pandas and Seaborn.
- Identification and handling missing numerical values and outliers.
- Convertion of categorical features into numerical formats.
- Feature engineering is the process of creating, transforming, or selecting the right features from raw data to improve the performance of machine learning models.

Supervised Learning (I): Classification

Objectives

- Describe the basic concepts of classification tasks, including inputs (features), outputs (labels), and common algorithms.
- Preprocess data in the Penguins dataset by handling missing values, managing outliers, and encoding categorical features.
- Perform classification tasks using representative algorithms (*e.g.*, k-NN, Logistic Regression, Naive Bayes, Support Vector Machine, Decision Tree, Random Forest, Gradient Boosting, Multi-Layer Perceptron, and Neural Networks).
- Evaluate model performance with metrics such as accuracy, precision, recall, F1-score, and confusion matrices.

Instructor note

- 40 min teaching/demonstration
- 40 min exercises

Classification

Classification is a supervised ML task in which a model predicts discrete class labels based on input features. It involves training the model on labeled data so that it can assign new and unseen data to predefined categories or classes by learning patterns from the training dataset.

In binary classification, the model predicts one of two classes, such as spam or not spam for emails. Multiclass classification extends this to multiple categories, like classifying images as cats, dogs, or birds.

Common algorithms for classification tasks include k-Nearest Neighbors (KNN), Logistic Regression, Naive Bayes, Support Vector Machine (SVM), Decision Trees, Random Forests, Gradient Boosting, and Neural Networks.

In this episode we will perform supervised classification to categorize penguins into three species — Adelie, Chinstrap, and Gentoo — based on their physical measurements (flipper length, body mass, *etc.*). We will build and train multiple classifier models, and then evaluate their performance using metrics such as accuracy, precision, recall, and F1 score. By comparing the results, we aim to identify which model provides the most accurate and reliable classification for this task.

Data Preparation

In the previous episode, Episode 4: Data Preparation for Machine Learning, we discussed data preparation steps, including handling missing values, detecting outliers, and encoding categorical variables.

In this episode, we will revisit these steps, with particular emphasis on encoding categorical variables. For the classification task, we will treat the categorical variable `species` as the label (target variable) and use the remaining columns as features to predict the penguins species. To achieve this, we transform the categorical features `island` and `sex`, as well as the `species` label, into numerical format (code examples are availalbe in the Jupyter Notebook).

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

penguins = sns.load_dataset('penguins')
penguins_classification = penguins.dropna()

# encode `species` column with 0=Adelie, 1=Chinstrap, and 2=Gentoo
penguins_classification.loc[:, 'species'] =
encoder.fit_transform(penguins_classification['species'])

# encode `island` column with 0=Biscoe, 1=Dream and 2=Torgersen
penguins_classification.loc[:, 'island'] =
encoder.fit_transform(penguins_classification['island'])

# encode `sex` column with 0=Female, and 1=Male
penguins_classification.loc[:, 'sex'] =
encoder.fit_transform(penguins_classification['sex'])
```

💬 Discussion

- why to use `species`?
- why not to use the other categorical variables (`island` and `sex`)?

Data Processing

In this episode, data processing will focus on two essential steps: **data splitting** and **feature scaling**

Data splitting

Data splitting involves two important substeps: splitting into features and labels, and splitting into training and testing sets.

The first substep is to split the dataset into features and labels. Features (also called predictors or independent variables) are the input values used to make predictions, while labels (or target variables) represent the output the model is trying to predict.

```
X = penguins_classification.drop(['species'], axis=1)
y = penguins_classification['species'].astype('int')
```

The second substep is to divide the Penguins dataset into training and testing sets. The training set is used to fit and train the models, allowing it to learn patterns and relationships from the data, and the testing set, on the other hand, is reserved for evaluating the model's performance on unseen data.

A common split is 80% for training and 20% for testing, which provides enough data for training while still retaining a meaningful set for testing. This step is typically performed using the `train_test_split` function from `sklearn.model_selection`, where setting a fixed `random_state` ensures reproducibility of the results.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)

print(f"Number of examples for training is {len(X_train)} and test is {len(X_test)}")
```

Feature scaling

Feature scaling is to standardize or normalize the range of independent variables (features) in a dataset. In many datasets, features can have different units or scales. For example, in the Penguins dataset, body mass is measured in grams and can range in the thousands, while flipper length is measured in millimeters and typically ranges in the hundreds. These differences in scale can unintentionally bias ML algorithms, making features with larger values dominate the learning process, leading to biased and inaccurate models.

Scaling transforms these features to a common, limited range, such as [0, 1] or a distribution with a mean of 0 and a standard deviation of 1, without distorting the differences in the ranges of values or losing information. This is particularly important for algorithms that rely on distance calculations, such as k-Nearest Neighbors (k-NN), Support Vector Machines (SVM), and clustering methods. Similarly, gradient-based optimization methods (used in neural networks and logistic regression) converge faster and more reliably when input features are scaled. Without scaling, the algorithm might oscillate inefficiently or struggle to find the optimal solution. Furthermore, it helps ensure that regularization penalties are applied uniformly across all coefficients, preventing the model from unfairly penalizing features with smaller natural ranges.

Two of the most common methods for feature scaling are **Normalization** (Min-Max Scaling) and **Standardization** (Z-score Normalization).

- Normalization (Min-Max Scaling)
  - This technique rescales the features to a fixed range, typically [0, 1].
  - It is calculated by subtracting the minimum value of the feature and then dividing by the range (max - min), and its formula is

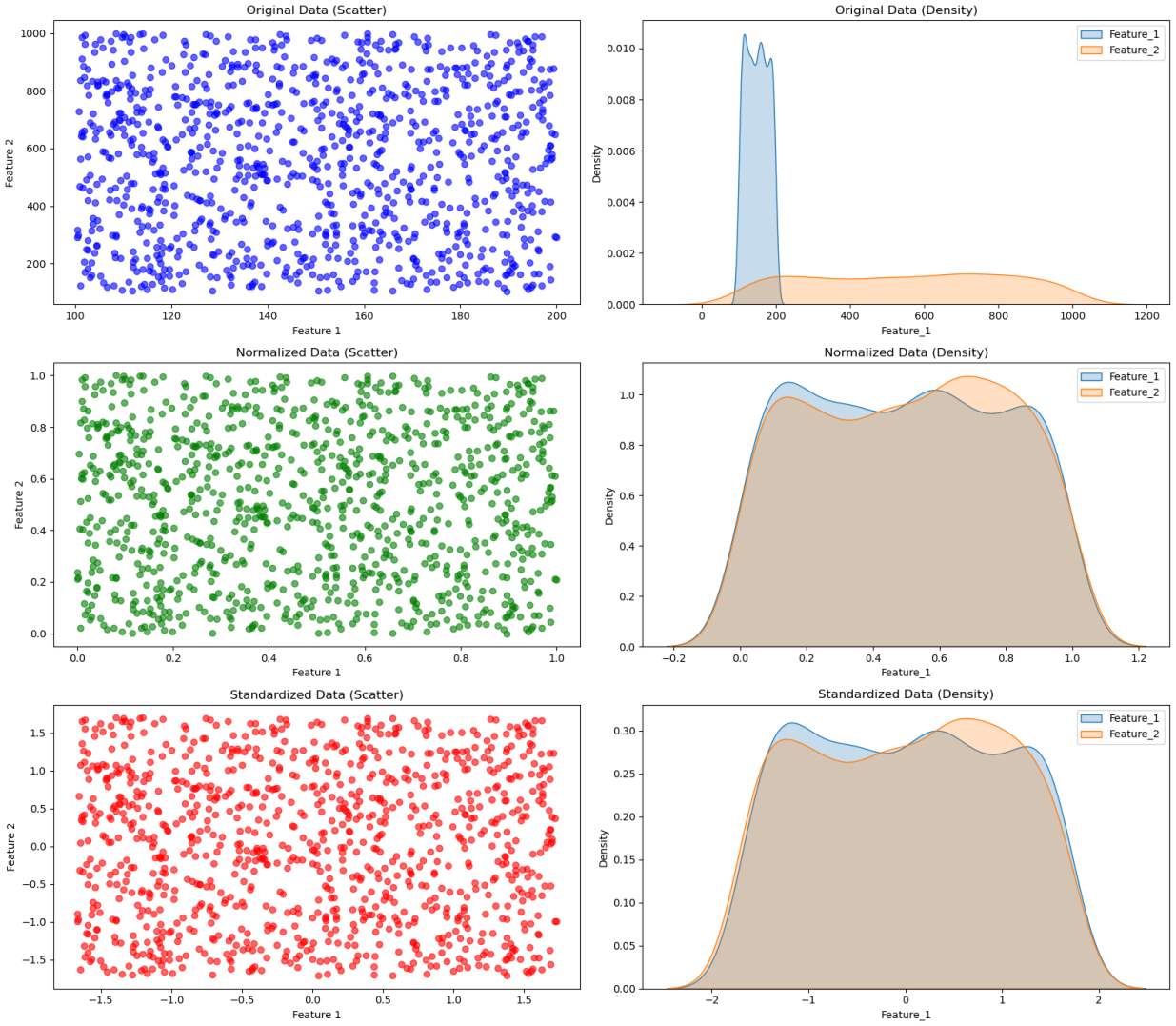
    \[X_scaled = \frac{(X - X_min)}{(X_max - X_min)}\]

  - This method is useful when the distribution is not Gaussian or when the algorithm requires input values bounded within a specific range (*e.g.*, neural networks often use activation functions that expect inputs in the [0,1] range).
- Standardization (Z-score Normalization)

- This technique transforms the data to have a mean of 0 and a standard deviation of 1.
- It is calculated by subtracting the mean value (μ) of the feature and then dividing by the standard deviation (σ), and its formula is

$$X\_scaled = \frac{X - \mu}{\sigma}.$$

- Standardization is less affected by outliers than Min-Max scaling and is often the preferred choice for algorithms that assume data is centered (like SVM and PCA).



In practice, these transformations are easily applied using libraries like scikit-learn with the `MinMaxScaler` and `StandardScaler` classes, which efficiently learn the parameters (`mean`, `min`, `max`) from the training data and apply them consistently to avoid data leakage.

In this episode, we will apply feature standardization to both the training and testing sets. The implementation can be easily achieved using `StandardScaler` from `sklearn.preprocessing`, as shown in the code below.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Training Model & Evaluating Model Performance

After preparing the Penguins dataset by handling missing values, encoding categorical variables, and splitting it into features/labels and training/testing sets, the next step is to apply classification algorithms. In this episode, we will experiment with k-Nearest Neighbors (KNN), Naive Bayes, Decision Trees, Random Forests, and Neural Networks to predict penguins species based on their physical measurements. Each of these algorithms offers a distinct approach to pattern recognition and generalization. By applying them to the same prepared dataset, we can make a fair and meaningful comparison of their predictive performance.

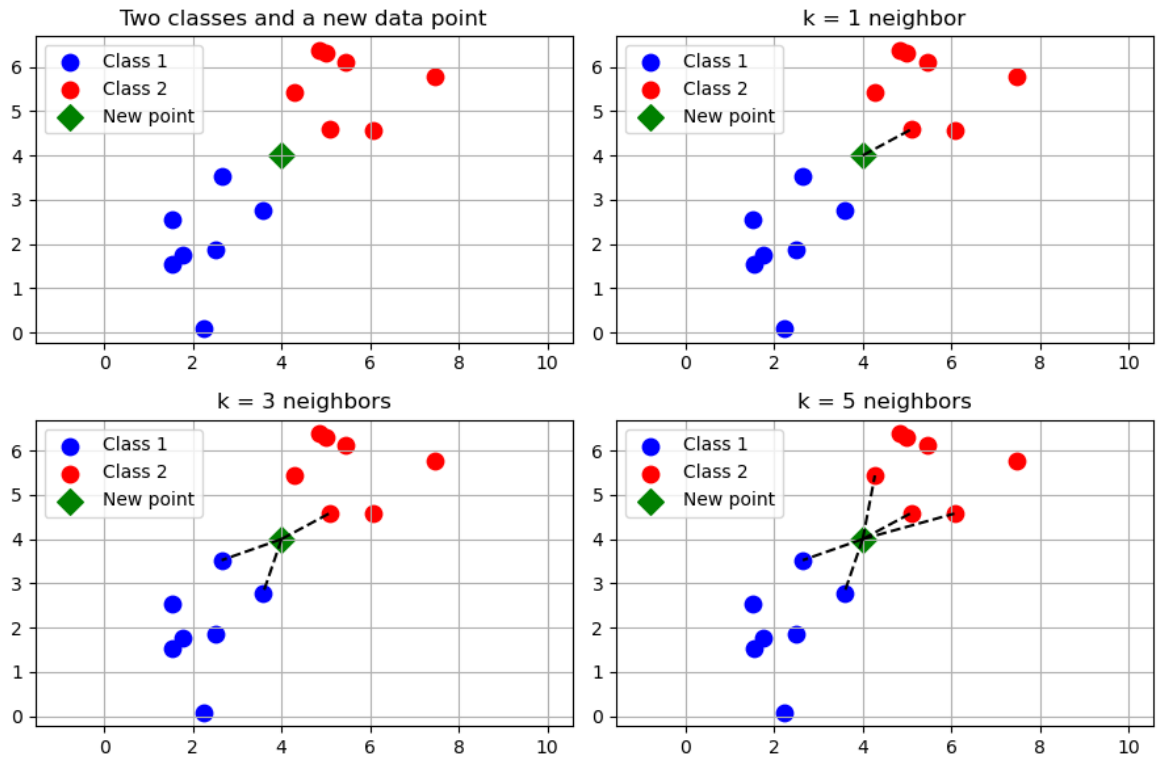
The workflow for training and evaluating a classification model generally follows these steps:

- Choose a model class and import it, `from sklearn.neighbors import XXX`.
- Set model hyperparameters by instantiating the class with desired values, `xxx_model = XXX(<... hyperparameters ...>)`.
- Train the model on the preprocessed training data using the `.fit()` method, `xxx_model.fit(X_train_scaled, y_train)`.
- Make predictions on the testing data with the `.predict()` method, `y_pred_xxx = xxx_model.predict(X_test_scaled)`.
- Evaluate model performance using appropriate metrics, `score_xxx = accuracy_score(y_test, y_pred_xxx)`.
- Visualize the results, for example by plotting a confusion matrix or other diagnostic charts to better understand model performance.

k-Nearest Neighbors (KNN)

One intuitive and widely used method for classification is the k-Nearest Neighbors (KNN) algorithm. KNN is a non-parametric, instance-based approach that predicts a sample's label by considering the majority class of its *k* closest neighbors in the training set. Unlike many other algorithms, **KNN does not require a traditional training phase**; instead, **it stores the entire dataset and performs the necessary computations at prediction step**. This makes it a lazy learner — simple to implement but potentially expensive during inference, especially with large datasets.

Below is an example illustrating how KNN determines the class of a new query point. Given a query point, KNN first calculates the distance between this point and all points in the training set. It then identifies the *k* closest points, and the class that appears most frequently among these neighbors is assigned as the predicted label for the query point. The choice of *k* plays a crucial role in performance: a small *k* can make the model overly sensitive to noise, while a large *k* may oversmooth the decision boundaries and obscure important local patterns.



Let's create a KNN model. Here, we set `k = 3`, meaning that the algorithm will consider the 3 nearest neighbors to determine the class of a data point. We then train the model on the training set using the `.fit()` method.

```
from sklearn.neighbors import KNeighborsClassifier

knn_model = KNeighborsClassifier(n_neighbors=3)
knn_model.fit(X_train_scaled, y_train)
```

After fitting the model to the training dataset, we use the trained KNN model to predict the species on the testing set and evaluate its performance.

For classification tasks, metrics such as accuracy, precision, recall, and the F1-score provide a comprehensive assessment of model performance:

- **Accuracy** measures the proportion of correctly classified instances across all species (Adelie, Chinstrap, Gentoo). It provides an overall sense of how often the model is correct but can be misleading when the dataset is imbalanced.
- **Precision** quantifies the proportion of correct positive predictions for each species, while **recall** measures the proportion of actual positives that are correctly identified.
- **F1-score** is the harmonic mean of precision and recall, offering a balanced metric for each class. It is particularly useful when dealing with imbalanced class distributions, as it accounts for both false positives and false negatives.

Relations among different matrics

In classification tasks, model predictions can be compared against the true labels to assess performance. This comparison is often summarized using four key concepts: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN).

Suppose we focus on identifying Adelie penguins as the positive class.

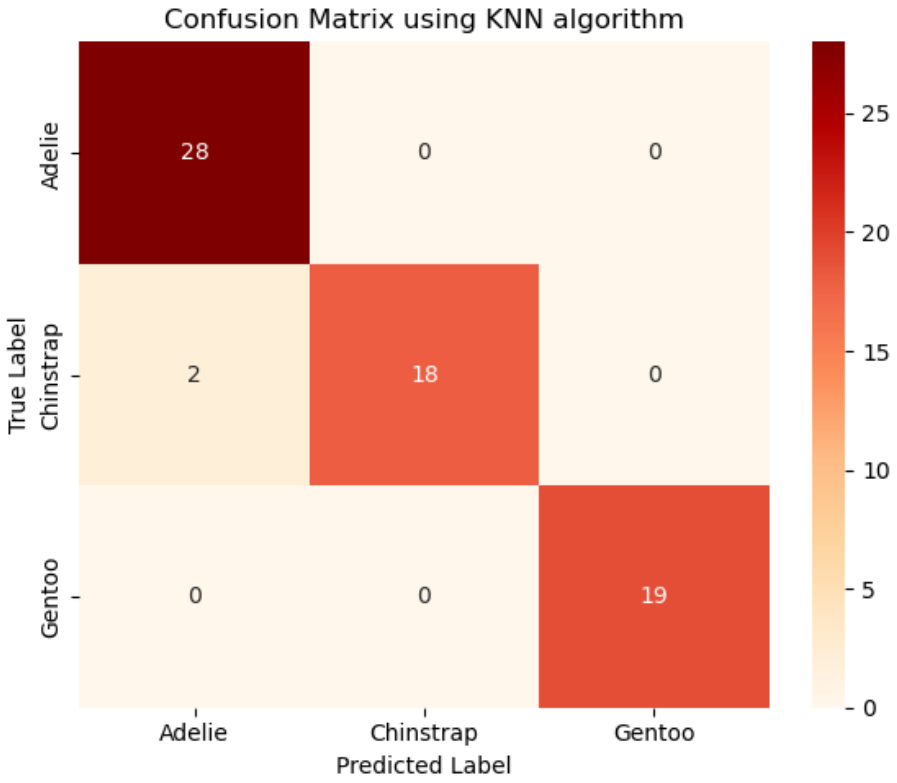
- A True Positive occurs when the model correctly predicts a penguin as Adelie and it truly belongs to that species.
- A True Negative happens when the model correctly identifies a penguin as not Adelie (*i.e.*, Chinstrap or Gentoo).
- A False Positive arises when the model incorrectly predicts a penguin as Adelie when it is actually another species.
- A False Negative occurs when an Adelie penguin is mistakenly predicted as Chinstrap or Gentoo.

These four outcomes form the basis of performance metrics such as accuracy, precision, recall, and F1-score, which help evaluate how well the model distinguishes between species.

```
# predict on testing data
y_pred_knn = knn_model.predict(X_test_scaled)

# evaluate model performance
from sklearn.metrics import classification_report, accuracy_score

score_knn = accuracy_score(y_test, y_pred_knn)
print("Accuracy for k-Nearest Neighbors:", score_knn)
print("\nClassification Report:\n", classification_report(y_test, y_pred_knn))
```

In classification tasks, a **confusion matrix** is a powerful tool for evaluating model performance by comparing predicted labels with true labels. For a multiclass problem like the Penguins dataset, the confusion matrix is an **N x N** matrix, where N represents the number of target classes (here, **N=3** for the three penguins species). Each cell *(i, j)* shows the number of instances where the true class was *i* and the model predicted class *j*. Diagonal elements correspond to correct predictions, while off-diagonal elements indicate misclassifications. This visualization provides an intuitive overview of how often the model predicts correctly and where it tends to make errors.

```
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(conf_matrix, title, fig_name):
    plt.figure(figsize=(6, 5))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='OrRd',
                xticklabels=["Adelie", "Chinstrap", "Gentoo"],
                yticklabels=['Adelie', 'Chinstrap', 'Gentoo'], cbar=True)

    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(title)
    plt.tight_layout()
    plt.savefig(fig_name)


cm_knn = confusion_matrix(y_test, y_pred_knn)

plot_confusion_matrix(cm_knn, "Confusion Matrix using KNN algorithm", "5-confusion-matrix-knn.png")
```



The first row: there are 28 Adelie penguins in the test data, and all these penguins are identified as Adelie (valid). The second row: there are 20 Chinstrap pengunis in the test data, with 2 identified as Adelie (invalid), and 18 identified as Chinstrap (valid). The third row: there are 19 Gentoo penguins in the test data, and all these penguins are identified as Gentoo (valid).

Warning

The choice of `k` can greatly affect the accuracy of KNN. Always try multiple `k` values and compare their performance. For the Penguins dataset, test different k values (*e.g.*, 3, 5, 7, 9, ...) to find the optimal k that gives the best classification results (accuracy score).

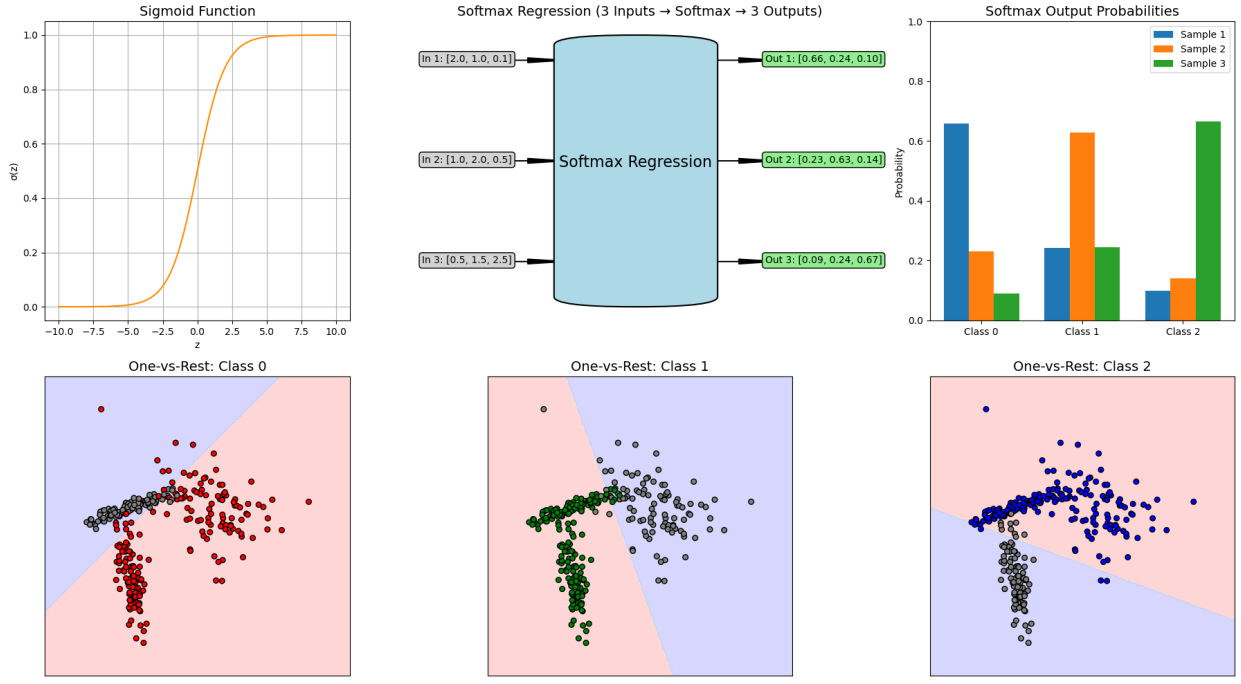
Logistic Regression

Logistic Regression is a fundamental classification algorithm to predict categorical outcomes. Despite its name, logistic regression is not a regression algorithm but a classification method that predicts the **probability** of an instance belonging to a particular class.

For binary classification, it uses the logistic (**sigmoid**) function to map a linear combination of input features to a probability between 0 and 1, which is then thresholded (typically at 0.5) to assign a class.

For multiclass classification, logistic regression can be extended using approaches such as one-vs-rest (OvR) or softmax regression.

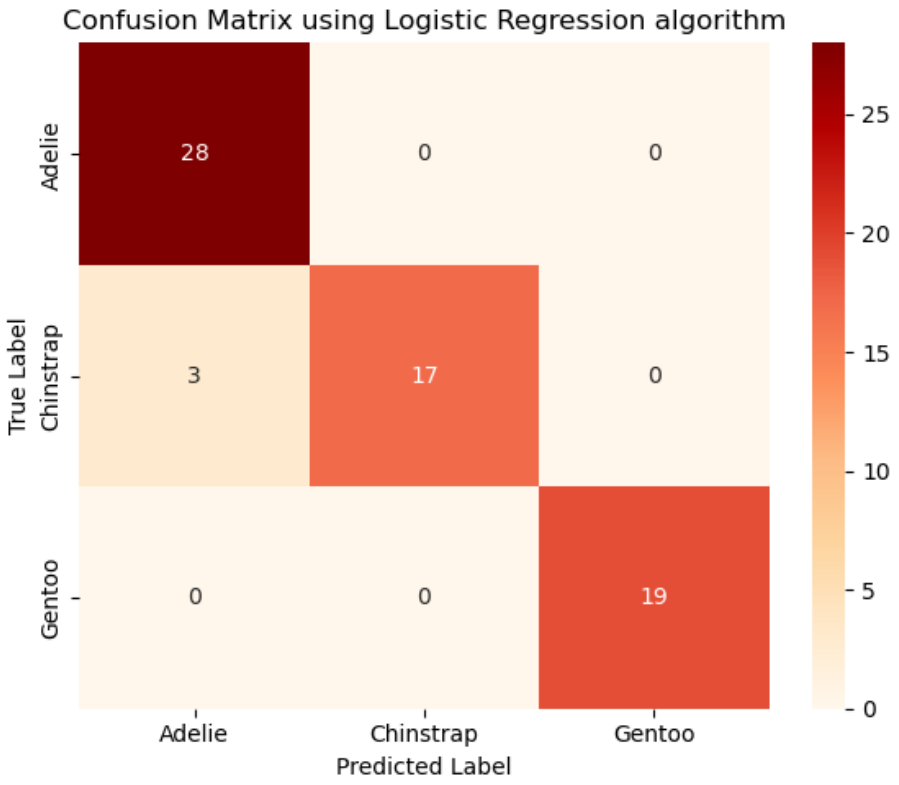
- In OvR, a separate binary classifier is trained for each species, treating that species as the positive class (blue area) and all other species as the negative class (red area).
- Softmax regression generalizes the logistic function to compute probabilities across all classes simultaneously, assigning each instance to the class with the highest predicted probability.



(Upper left) the sigmoid function; (upper middle) the softmax regression process: three input features to the softmax regression model resulting in three output vectors where each contains the predicted probabilities for three possible classes; (upper right) a bar chart of softmax outputs in which each group of bars represents the predicted probability distribution over three classes; (lower subplots) three binary classifiers distinguish one class from the other two classes using the one-vs-rest approach.

The process of creating a Logistic Regression model and fitting it to the training data is very similar to the approach used for the KNN model described earlier, with the main difference being the choice of classifier. A code example and the resulting confusion matrix plot are provided below.

```
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(random_state = 123)
lr_model.fit(X_train_scaled, y_train)

y_pred_lr = lr_model.predict(X_test_scaled)

score_lr = accuracy_score(y_test, y_pred_lr)
print("Accuracy for Logistic Regression:", score_lr )
print("\nClassification Report:\n", classification_report(y_test, y_pred_lr))

cm_lr = confusion_matrix(y_test, y_pred_lr)

plot_confusion_matrix(cm_lr, "Confusion Matrix using Logistic Regression algorithm",
"5-confusion-matrix-lr.png")
```



Naive Bayes

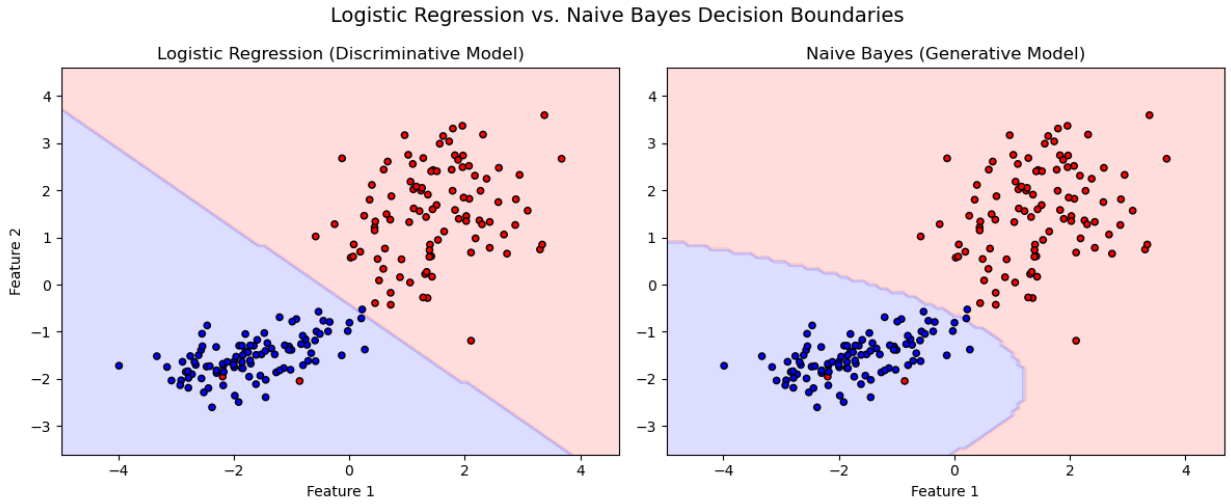
The **Naive Bayes** algorithm is a simple yet powerful probabilistic classifier based on Bayes' Theorem. It assumes that all features are and equally important — a condition that often does not hold in practice, which can introduce some bias. However, this independence assumption greatly simplifies computations by allowing conditional probabilities to be expressed as the product of individual feature probabilities. Given an input instance, the algorithm calculates the posterior probability for each class and assigns the instance to the class with the highest probability.

Logistic Regression and Naive Bayes are both popular algorithms for classification tasks, but they differ significantly in their approach, assumptions, and underlying mechanics. Below is an example comparing Logistic Regression and Naive Bayes decision boundaries on a

synthetic dataset with two features. The visualization highlights their fundamental differences: **Logistic Regression learns a linear decision boundary directly, whereas Naive Bayes models feature distributions for each class under the independence assumption**.

❗ Logistic Regression *vs.* Naive Bayes

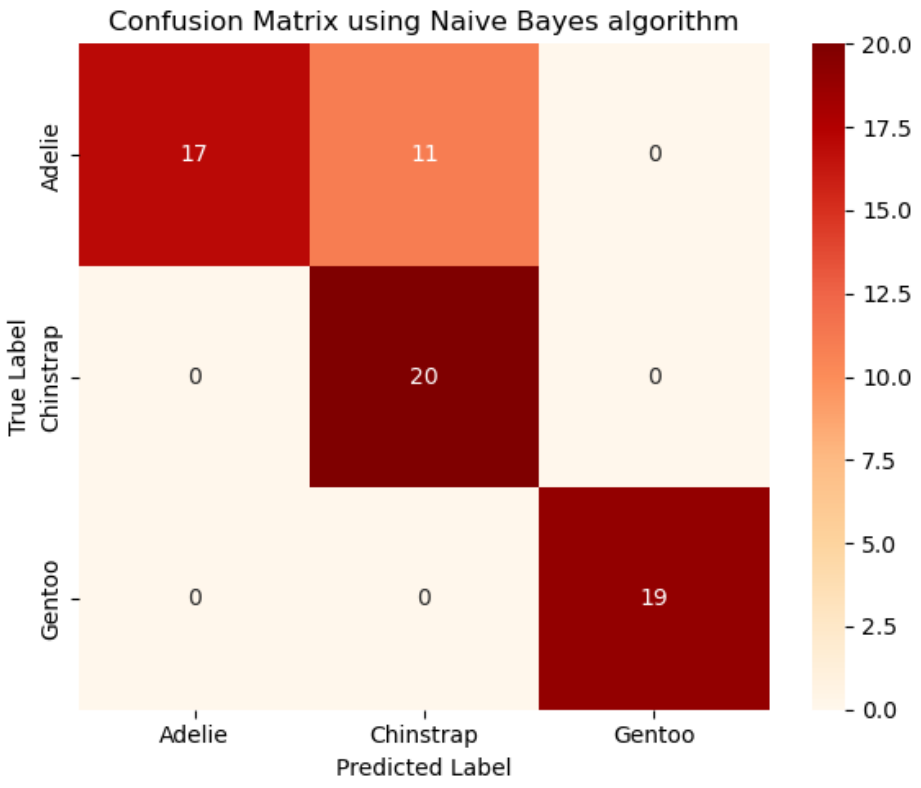
- Logistic Regression is a **discriminative** model that directly estimates the probability of a data point belonging to a particular class by fitting a linear combination of features. In the context of the Penguins dataset, Logistic Regression uses features such as bill length and flipper length to compute a weighted sum, which is then transformed into probabilities for penguins species. The model assumes a linear relationship between the features and the log-odds of the classes and optimizes parameters using maximum likelihood estimation. This makes Logistic Regression sensitive to feature scaling and correlations. It is generally robust to noise and can tolerate moderately correlated features, but it may struggle with highly non-linear relationships unless additional feature engineering is applied.
- Naive Bayes, by contrast, is a **generative** model that applies Bayes' theorem to estimate the probability of a class given the input features, assuming conditional independence between features. For the Penguins dataset, it estimates the likelihood of features (*e.g.*, bill depth) for each species and combines these with prior probabilities to predict the most likely species. The "naive" independence assumption often does not hold in practice (*e.g.*, bill length and depth may be correlated), but it simplifies computation and allows Naive Bayes to be highly efficient, especially for high-dimensional data. It is less sensitive to irrelevant features and does not require feature scaling. However, it can underperform when feature dependencies are strong or when the data distribution deviates from the model's assumptions (*e.g.*, Gaussian for continuous features in Gaussian Naive Bayes). Zero probabilities must be carefully handled, typically via smoothing techniques.



To apply Naive Bayes, we use `GaussianNB` from `sklearn.naive_bayes`, which assumes that the features follow a Gaussian (normal) distribution — making it suitable for continuous numerical data such as bill length and body mass.

- Because Naive Bayes relies on probabilities, **feature scaling is not required**; however, **handling missing values and encoding categorical variables numerically remains necessary**.
- While Naive Bayes may not always match the performance of more complex models like Random Forests, it offers fast training, low memory requirements, and reliable performance for simpler classification tasks.

```
from sklearn.naive_bayes import GaussianNB

nb_model = GaussianNB()
nb_model.fit(X_train_scaled, y_train)

y_pred_nb = nb_model.predict(X_test_scaled)

score_nb = accuracy_score(y_test, y_pred_nb)
print("Accuracy for Naive Bayes:", score_nb)
print("\nClassification Report:\n", classification_report(y_test, y_pred_nb))

cm_nb = confusion_matrix(y_test, y_pred_nb)
plot_confusion_matrix(cm_nb, "Confusion Matrix using Naive Bayes algorithm", "4-
confusion-matrix-nb.png")
```

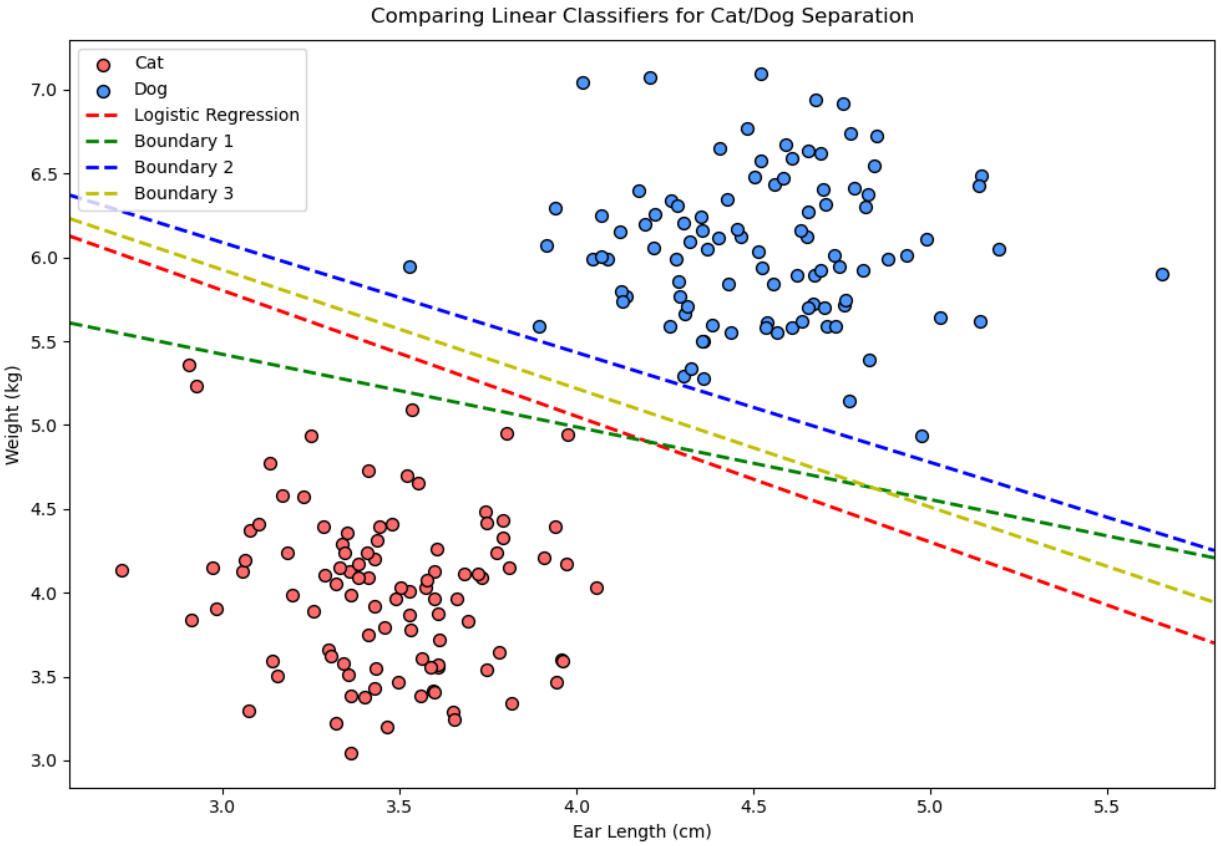


Support Vector Machine (SVM)

Previously, we presented an example using a Logistic Regression classifier, which produces a linear decision boundary to separate two classes based on their features. It works by fitting this linear boundary using the logistic function, making it particularly effective when the data is linearly separable. A notable characteristic of Logistic Regression is that the decision boundary typically lies in the region where the predicted probabilities of the two classes are closest — essentially where the model is most uncertain.

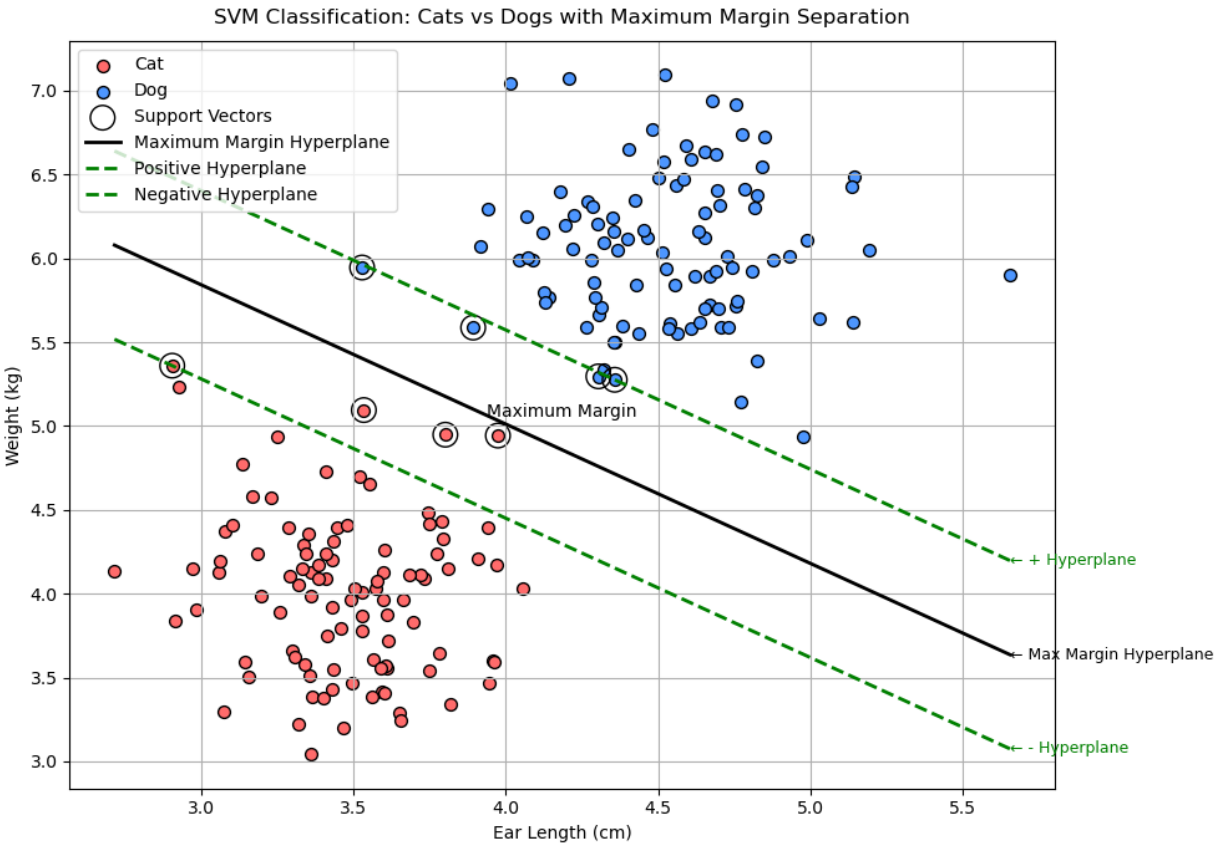
However, when there is a large gap between two well-separated classes — as can occur when distinguishing cats from dogs based on weight and size — Logistic Regression faces an inherent limitation: an infinite number of possible solutions. The algorithm has no built-in mechanism to select a single "optimal" boundary when multiple valid linear separators exist within the wide margin between classes. As a result, it may place the decision boundary somewhere in that gap, creating a broad, undefined region with little or no supporting data. While this may not affect accuracy on clearly separated data, it can reduce the model's robustness when new or noisy data points appear near that boundary.

Below is another example of separating cats from dogs based on ear length and weight. In addition to the linear decision boundary produced by the Logistic Regression classifier, we can identify three other linear boundaries that also achieve good separation between the two classes. The question then arises: which boundary is truly better, and how can we evaluate their performance on unseen data?



To better handle such situations, we can turn to the **Support Vector Machine (SVM)** algorithm. Unlike Logistic Regression, SVM focuses on maximizing the margin — the distance between the decision boundary and the closest data points from each class, known as support vectors (as illustrated in the figure below). When a large gap exists between two classes, SVM takes advantage of this space by positioning the boundary near the center of the gap while maintaining the maximum margin. This results in a more stable and robust classifier, especially when the classes are well-separated.

Unlike Logistic Regression, which considers all data points to estimate probabilities, SVM relies primarily on the most critical examples — those closest to the decision boundary — making it less sensitive to outliers and more precise in defining class separations.



The SVM classification boundary for distinguishing cats and dogs based on ear length and weight. The solid black line represents the maximum margin hyperplane (decision boundary), while the dashed green lines indicate the positive and negative hyperplanes that define the margin. The black circles highlight the support vectors — the critical data points that determine the width of the margin.

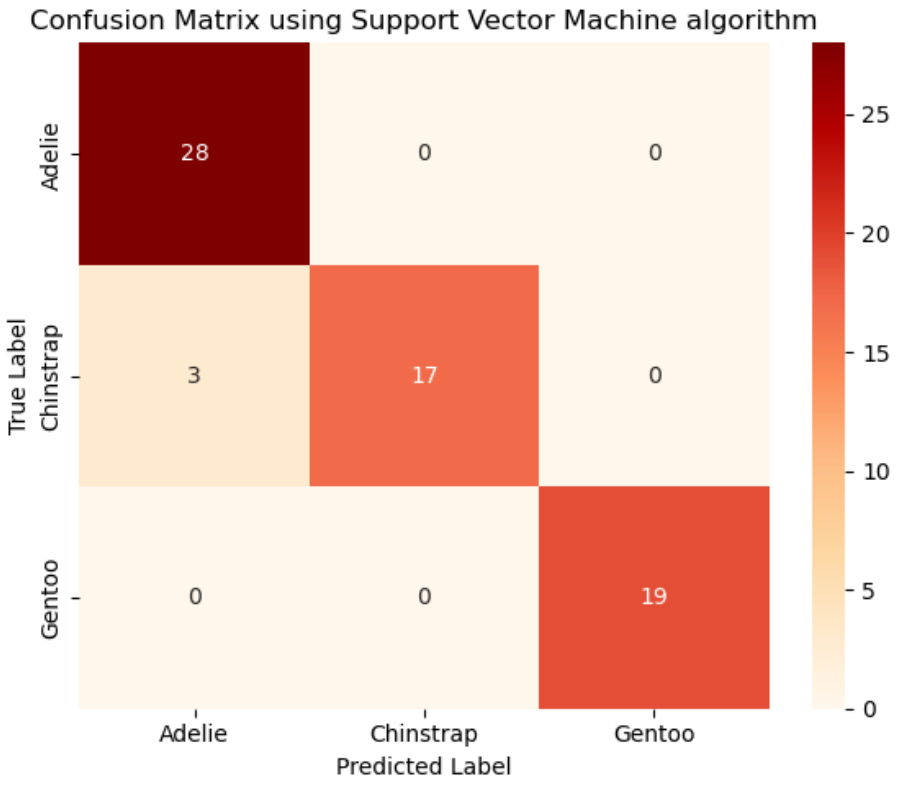
To apply SVM, we use `SVC` (Support Vector Classification) from `sklearn.svm`. By default, it assumes a nonlinear relationship between features, modeled using the `rbf` (Radial Basis Function) kernel. This kernel enables the model to learn complex decision boundaries by implicitly mapping input features into a higher-dimensional space.

Note

You can also experiment with other kernels, such as `linear`, `poly`, or `sigmoid`, to explore different types of decision boundaries.

By adjusting hyperparameters such as `C` (regularization strength) and `gamma` (kernel coefficient), we can control the trade-off between margin width and classification accuracy. Below is a code example demonstrating how to apply `SVC` with the `rbf` kernel to classify penguins.

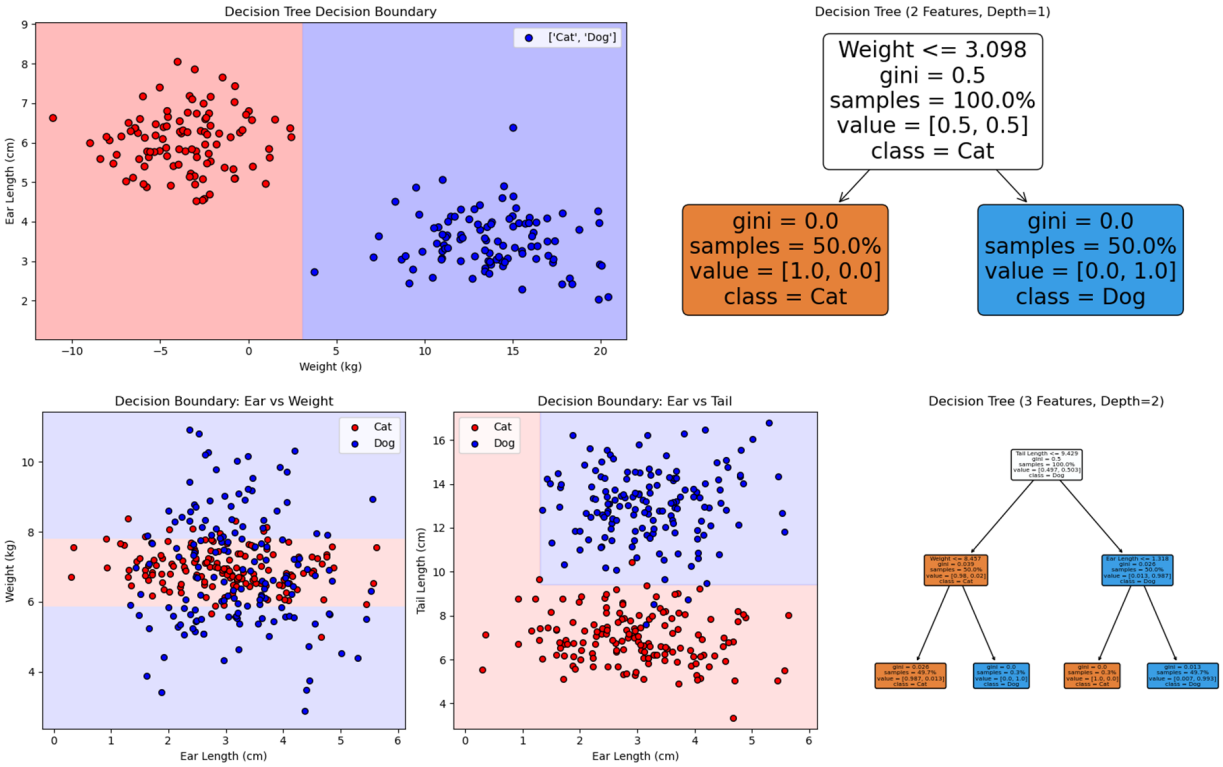
```
from sklearn.svm import SVC

svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=123)
svm_model.fit(X_train_scaled, y_train)

y_pred_svm = svm_model.predict(X_test_scaled)

score_svm = accuracy_score(y_test, y_pred_svm)
print("Accuracy for Support Vector Machine:", score_svm)
print("\nClassification Report:\n", classification_report(y_test, y_pred_svm))

cm_svm = confusion_matrix(y_test, y_pred_svm)

plot_confusion_matrix(cm_svm, "Confusion Matrix using Support Vector Machine
algorithm", "5-confusion-matrix-svm.png")
```



Decision Tree

The **Decision Tree** algorithm is a versatile and highly interpretable method for classification tasks. Its core idea is to recursively split the dataset into smaller subsets based on feature thresholds, creating a tree-like structure of decisions that maximizes the separation of target classes.

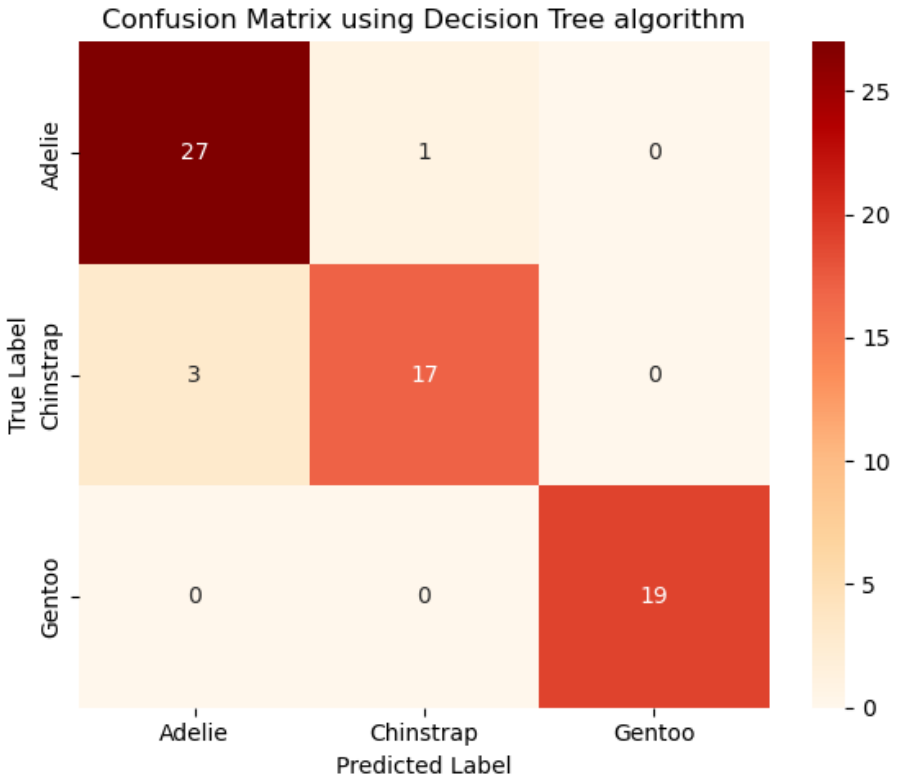
For example, a decision tree can be used to classify cats and dogs based on two or three features, illustrating how the algorithm partitions the feature space to distinguish between classes.



(Upper): Decision boundary separating cats and dogs based on two features (ear length and weight), along with the corresponding decision tree structure. (lower): Decision boundaries separating cats and dogs based on three features (ear length, weight, and tail length), and the corresponding decision tree structure.

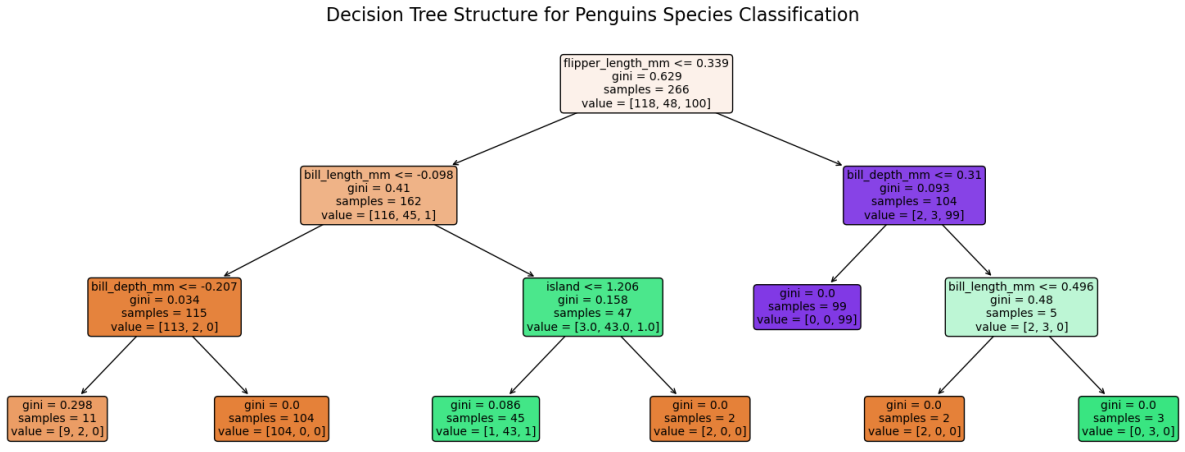
Below is a code example demonstrating the Decision Tree classifier applied to the penguins classification task.

```
from sklearn.tree import DecisionTreeClassifier

dt_model = DecisionTreeClassifier(max_depth=3, random_state = 123)
dt_model.fit(X_train_scaled, y_train)

y_pred_dt = dt_model.predict(X_test_scaled)

score_dt = accuracy_score(y_test, y_pred_dt)
print("Accuracy for Decision Tree:", score_dt )
print("\nClassification Report:\n", classification_report(y_test, y_pred_dt))

cm_dt = confusion_matrix(y_test, y_pred_dt)
plot_confusion_matrix(cm_dt, "Confusion Matrix using Decision Tree algorithm", "5-
confusion-matrix-dt.png")
```



We visualize the Decision Tree structure to better understand how penguins are classified based on their physical characteristics.

```
from sklearn.tree import plot_tree

plt.figure(figsize=(16, 6))
plot_tree(dt_model, feature_names=X.columns, filled=True, rounded=True, fontsize=10)

plt.title("Decision Tree Structure for Penguins Species Classification", fontsize=16)

plt.tight_layout()
plt.show()
```



(Optional) Random Forest

While Decision Trees are easy to interpret and visualize, they have some notable drawbacks. One primary issue is their tendency to overfit the training data, particularly when the tree is allowed to grow deep without constraints such as maximum depth or minimum samples per

split. Overfitting causes the model to capture noise in the training data, which can lead to poor generalization on unseen data — for example, misclassifying a Gentoo penguin as a Chinstrap due to overly specific splits. Additionally, decision trees are sensitive to small variations in the data; even slight changes, such as a few noisy measurements, can result in a significantly different tree structure, reducing the model's stability and reliability.

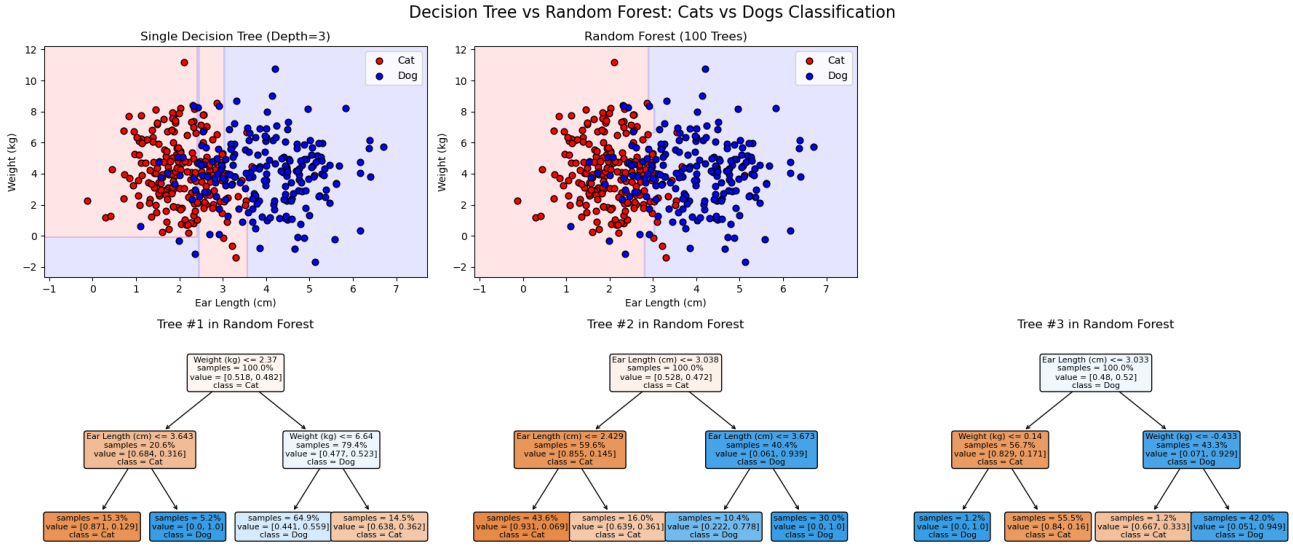
To address these limitations, we can use an ensemble learning technique called **Random Forest**. A Random Forest builds on the concept of decision trees by creating a large collection of them, each trained on a randomly selected subset of the data and features. By aggregating the predictions of multiple trees — typically through majority voting for classification — Random Forest reduces overfitting, improves generalization, and mitigates the inherent instability in individual decision trees.

Note

Ensemble learning is a ML approach that combines multiple individual models (often called base learners) to create a stronger, more accurate, and more robust overall model. The idea is that by aggregating the predictions of several models, the ensemble can reduce errors, improve generalization, and mitigate weaknesses of individual models. There are two main types of ensemble learning techniques:

- **Bagging** (**Bootstrap Aggregating**): Multiple models are trained independently on random subsets of the data, and their predictions are averaged (for regression) or voted on (for classification). Random Forest is a classic example of bagging applied to decision trees.
- **Boosting**: Models are trained sequentially, with each new model focusing on the errors made by previous models. Examples include AdaBoost, Gradient Boosting, and XGBoost.

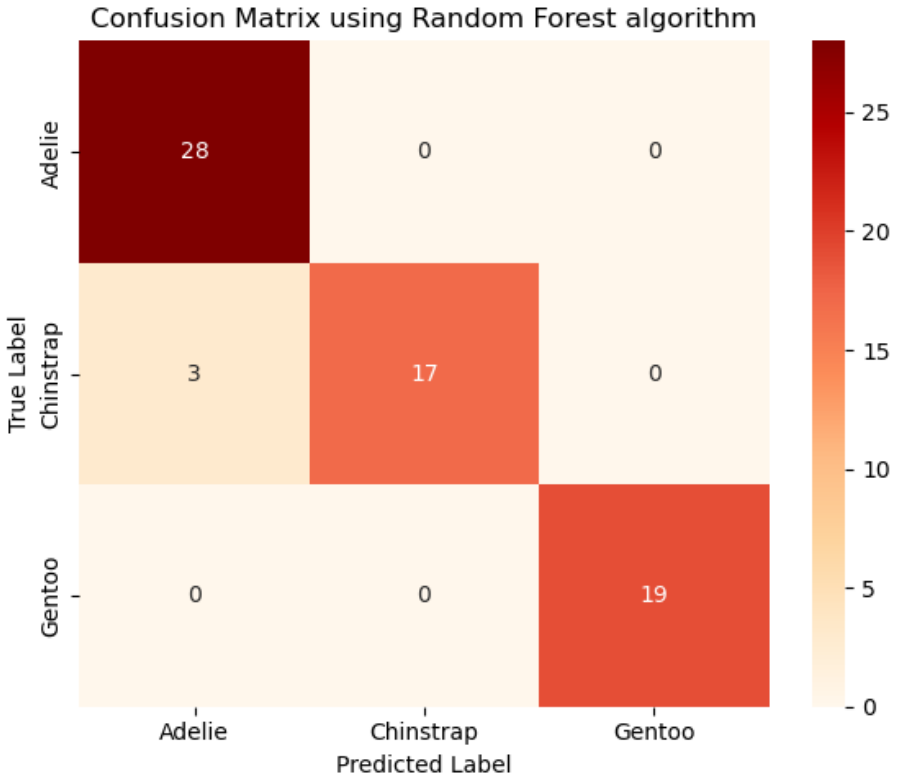
The figure below illustrates how a Random Forest improves upon a single Decision Tree when classifying cats and dogs based on synthetic measurements of ear length and weight.



Top row shows the classification boundaries for both models. On the left, a single Decision Tree creates rigid, rectangular decision regions that precisely follow axis-aligned splits in the training data. While this achieves a good separation of the training samples, the jagged boundaries suggest potential overfitting to noise. In contrast, the Random Forest (right) produces smoother, more nuanced decision boundaries through majority voting across 100 trees. The blended purple transition zones represent areas where individual trees disagree, demonstrating how the ensemble averages out erratic predictions from any single tree. Bottom row reveals why Random Forests are more robust by examining three constituent trees. Tree #1 prioritizes ear length for its initial split, Tree #2 begins with weight, and Tree #3 uses a completely different weight threshold.

Below is a code example demonstrating the application of the Random Forest classifier to the penguins classification task.

```
from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(n_estimators=100, random_state=123)
rf_model.fit(X_train_scaled, y_train)

y_pred_rf = rf_model.predict(X_test_scaled)

score_rf = accuracy_score(y_test, y_pred_rf)
print("Accuracy for Random Forest:", score_rf )
print("\nClassification Report:\n", classification_report(y_test, y_pred_rf))

cm_rf = confusion_matrix(y_test, y_pred_rf)

plot_confusion_matrix(cm_rf, "Confusion Matrix using Random Forest algorithm", "5-confusion-matrix-rf.png")
```



In addition to the confusion matrix, feature importance in a Random Forest (and also in a Decision Tree) model provides valuable insight into which input features contribute most to the model's predictions. Random Forest calculates feature importance by measuring how much each feature reduces impurity — such as Gini impurity or entropy — when used to split the data across all trees in the forest. Features that produce greater reductions in impurity are considered more important. These importance scores are then normalized to provide a relative ranking, helping to identify which features most strongly influence the model's predictions. This information is particularly useful for interpreting model behavior, selecting meaningful features, and understanding the underlying structure of the data.

Note

In Decision Tree and Random Forest, impurity measures how "mixed" the classes are in a given node. A pure node contains only instances of a single class, while an impure node contains a mixture of classes. Impurity metrics help the tree decide which feature and threshold to use when splitting the data to create nodes that are as pure as possible.

Gini impurity and entropy are metrics used to measure impurity of a dataset or a node.

During training, the algorithm evaluates all possible splits for a feature. It chooses the split that maximizes purity, *i.e.*, **minimizes Gini impurity** or maximizes information gain (**reduction in entropy**).

The greater the total reduction in impurity attributed to a feature, the more important it is considered. These importance scores are then normalized to provide a relative ranking, helping identify which features have the most influence on predicting the output class. This information is particularly useful for interpreting model behavior, selecting meaningful features, and understanding the underlying structure of the data.

Below is a code example showing how to plot feature importance using a Random Forest model to classify penguins into three categories.

```
importances = rf_model.feature_importances_
features = X.columns

plt.figure(figsize=(9, 6))
plt.barh(features, importances, color="tab:orange", alpha=0.75)
plt.xlabel("Feature Importance")
plt.ylabel("Features")
plt.title("Random Forest Feature Importance")
plt.tight_layout()
plt.show()
```

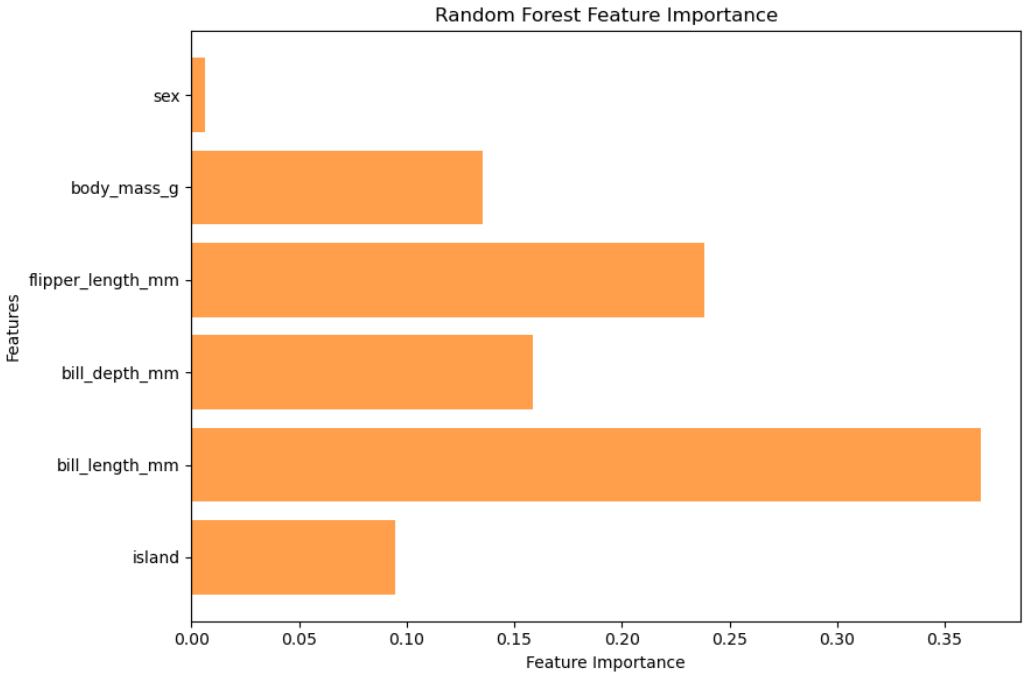


Illustration of feature importance for penguin classification. Longer bars indicate features with greater influence on the model's decisions, showing that the Random Forest relies more heavily on these measurements to identify species.

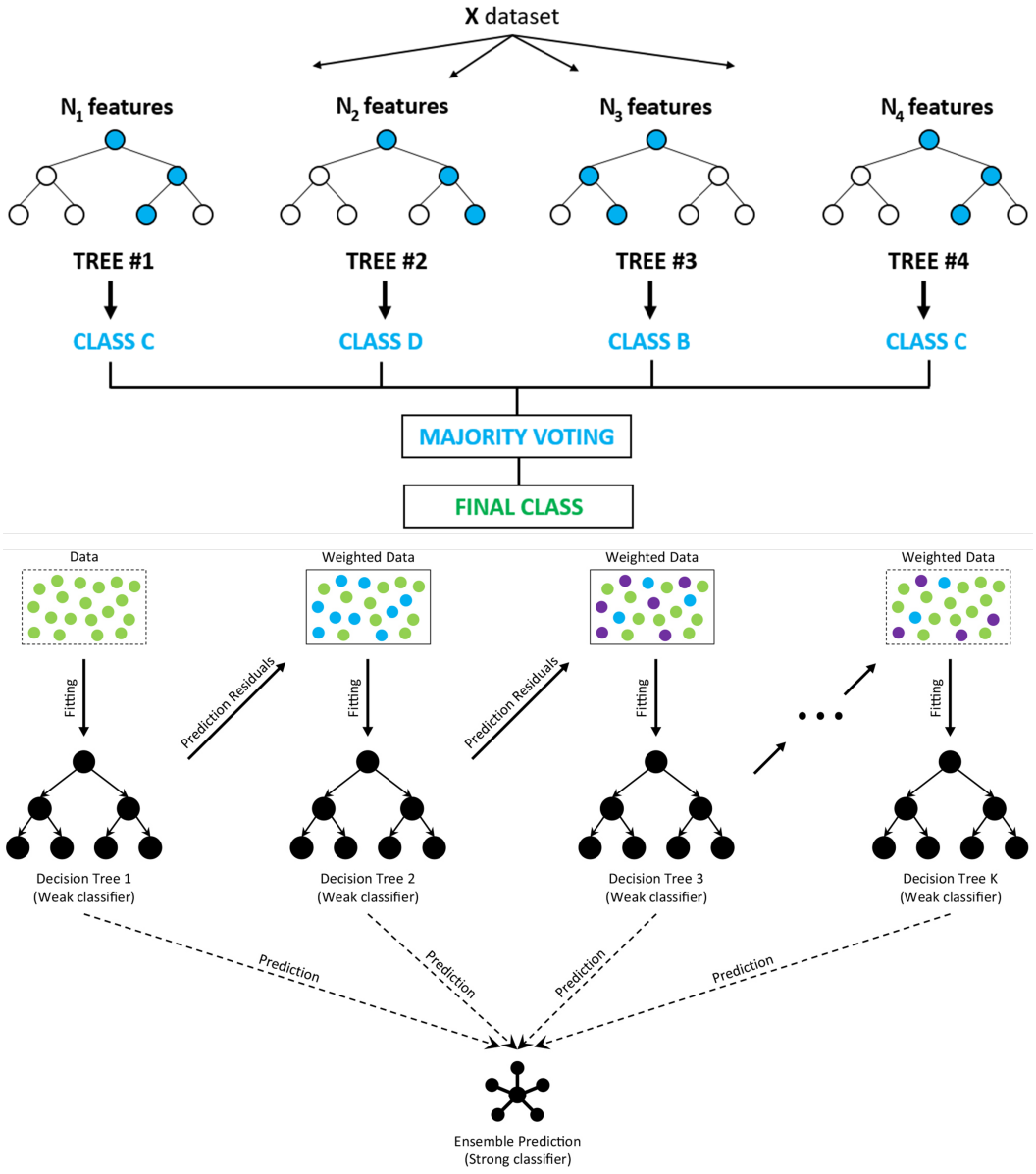
(Optional) Gradient Boosting

We have trained the model using a Decision Tree classifier, providing an intuitive starting point for classifying penguin species based on physical measurements. However, this classifier is sensitive to small fluctuations in the dataset, which can often lead to overfitting, especially when the tree grows deep.

To address the limitations of a single decision tree, we turned to Random Forest, an ensemble method that builds multiple decision trees on different random subsets of the data and features. By averaging the predictions of all trees or taking a majority vote in classification, Random Forest reduces overfitting and improves generalization. This approach balances model complexity with predictive performance and provides a reliable estimate of feature importance, helping identify which physical attributes are most influential in distinguishing penguin species.

While Random Forest provides robustness and improved accuracy over individual trees, we can further enhance performance using **Gradient Boosting**.

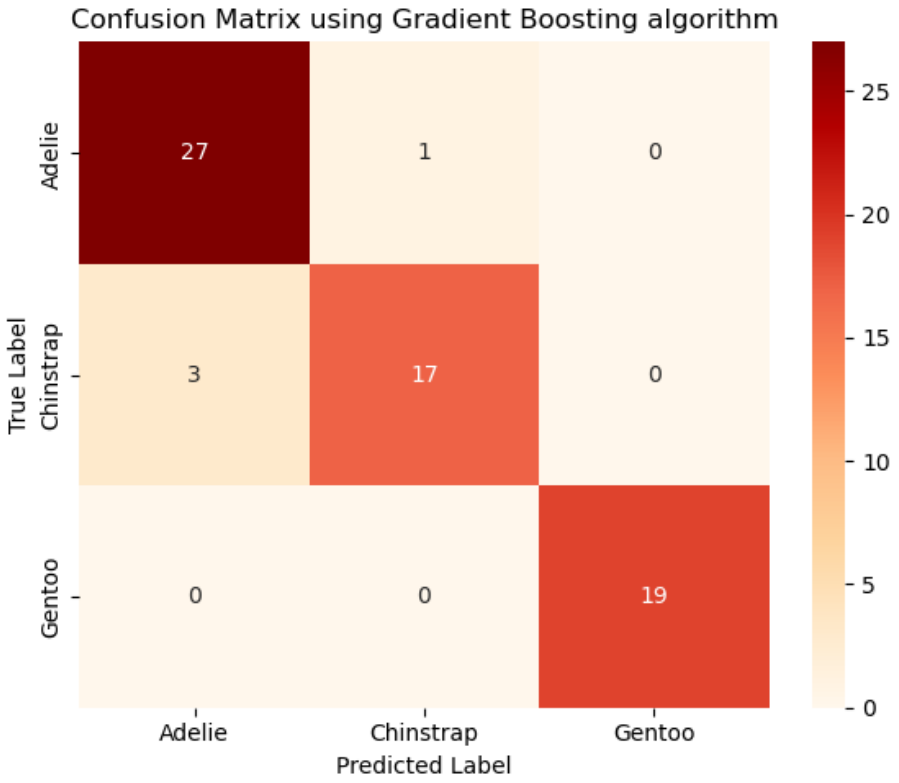
- Like Random Forest, Gradient Boosting is an ensemble learning technique, but it builds a strong classifier by combining many weak learners (typically shallow decision trees) in a sequential manner.
- Unlike Random Forest, which grows multiple trees independently and in parallel using random subsets of the training data, Gradient Boosting constructs trees one at a time, with each new tree trained to correct the errors of its predecessors.



lillustration of the Random Forest and Gradient Boosting algorithms.

In this code example below, we apply Gradient Boosting algorithm to classify penguin species. We use `GradientBoostingClassifier` from scikit-learn due to its simplicity and strong baseline performance.

```
from sklearn.ensemble import GradientBoostingClassifier

gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3,
random_state=123)
gb_model.fit(X_train_scaled, y_train)

y_pred_gb = gb_model.predict(X_test_scaled)

score_gb = accuracy_score(y_test, y_pred_gb)
print("Accuracy for Gradient Boosting:", score_gb)
print("\nClassification Report:\n", classification_report(y_test, y_pred_gb))

cm_gb = confusion_matrix(y_test, y_pred_gb)
plot_confusion_matrix(cm_gb, "Confusion Matrix using Gradient Boosting algorithm", "5-
confusion-matrix-gb.png")
```



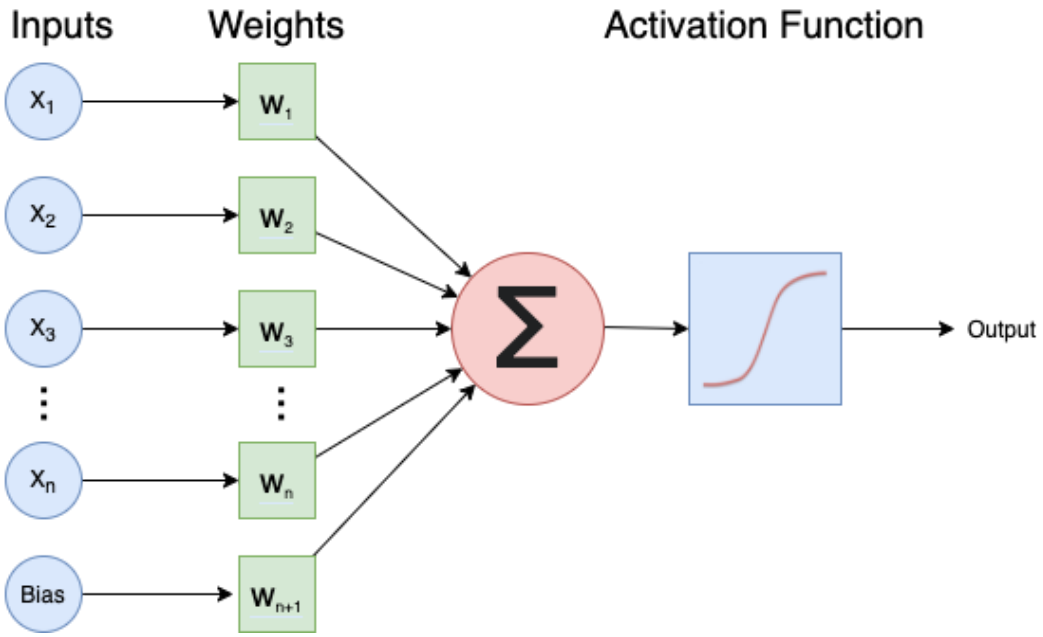
Note

This progression — from the simplicity of a single Decision Tree, to the robustness of Random Forest, and finally to the precision of Gradient Boosting — mirrors the evolution of **tree-based methods** in modern ML. While Random Forest remains excellent for baseline performance, Gradient Boosting often achieves state-of-the-art results on structured data, such as ecological measurements, provided the learning rate and tree depth are carefully tuned.

Multi-Layer Perceptron

A **Multilayer Perceptron** (MLP) is a type of artificial neural network consisting of multiple layers of interconnected perceptrons (or neurons) designed to mimic certain aspects of human brain function. Each neuron (illustrated in the figure below) has the following characteristics:

- Input: one or more inputs (`x_1` , `x_2` , ...), *e.g.*, features from the input data expressed as floating-point numbers.
- Operations: Typically, each neuron conducts three main operations:
  - Compute the weighted sum of the inputs where (`w_1` , `w_2` , ...) are the corresponding weights.
  - Add a bias term to the weighted sum.
  - Apply an activation function to the result.
- Output: The neuron produces a single output value.



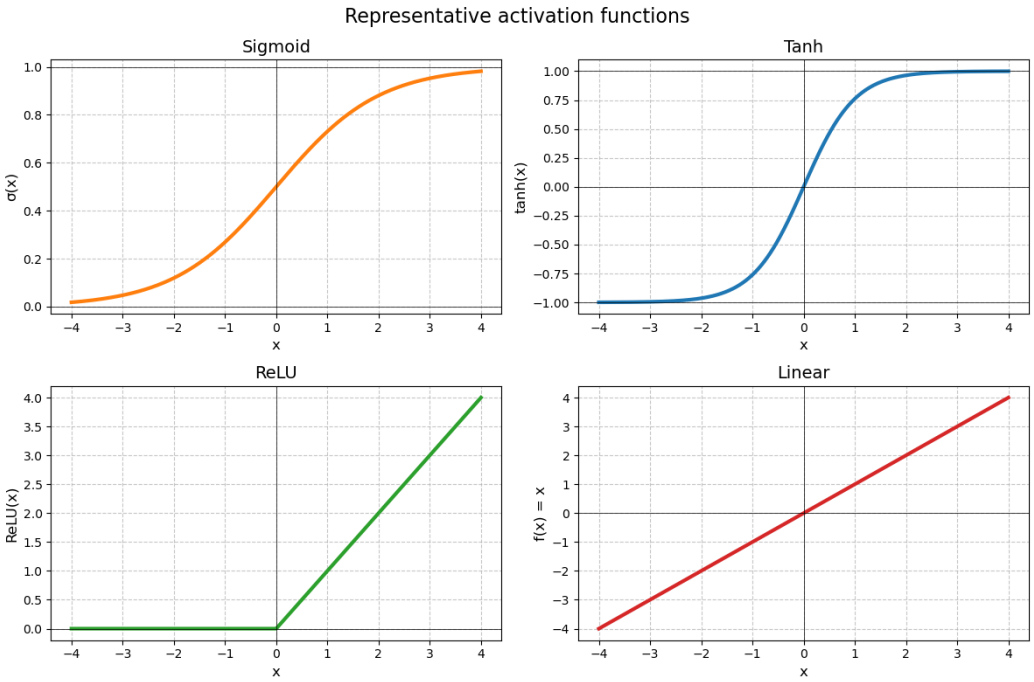
A common equation for the output of a neuron is

\[output = Activation(\sum_i (x_i * w_i) + bias).\]

An **activation function** is a mathematical transformation that converts the weighted sum of a neuron's inputs into its output signal. By introducing non-linearity into the network, activation functions enable neural networks to learn complex patterns and make sophisticated decisions based on the weighted inputs.

Below are some commonly used activation functions in neural networks and DL models. Each plays a crucial role in introducing non-linearities, allowing the network to capture intricate patterns and relationships in data.

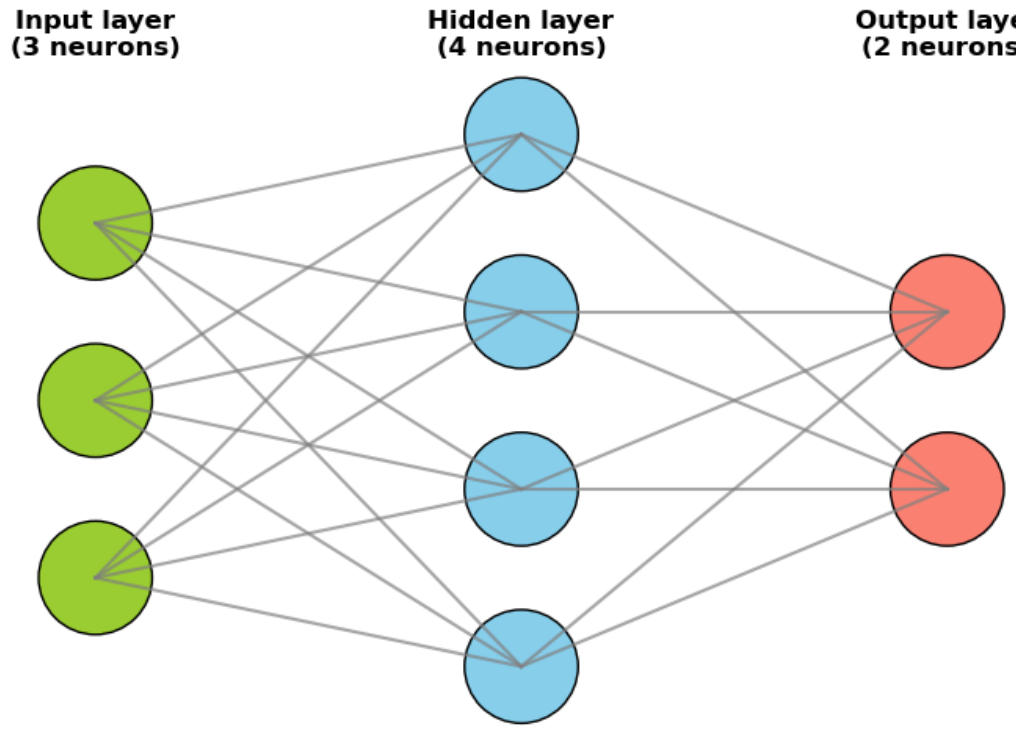
- **Sigmoid**: With its characteristic S-shaped curve, the sigmoid function maps inputs to a smooth 0-1 range, making it historically popular for binary classification tasks.
- **Hyperbolic tangent** (tanh): Similar to sigmoid but ranging from -1 to 1, tanh often provides stronger gradients during training.
- **Rectified Linear Unit** (ReLU): Outputs zero for negative inputs and the identity for positive inputs. ReLU has become the default choice for many architectures due to its computational efficiency and its ability to mitigate the vanishing gradient problem.
- **Linear**: This identity function serves as a reference, showing network behavior without any non-linear transformation.



A single neuron (perceptron) can learn simple patterns but is limited in modeling complex relationships. By combining multiple neurons into layers and connecting them into a network, we create a powerful computational framework capable of approximating highly non-linear functions. In a Multilayer Perceptron (MLP), neurons are organized into an input layer, one or more hidden layers, and an output layer.

The image below illustrates a three-layer perceptron network with 3, 4, and 2 neurons in the input, hidden, and output layers, respectively.

- The input layer receives raw data, such as pixel values or measurements, and passes it to the hidden layer.
- The hidden layer contains multiple neurons that process the information and progressively extract higher-level features. Each neuron in the hidden layer is fully connected to neurons in adjacent layers, forming a dense network of weighted connections.
- The output layer produces the network's predictions, whether it's a classification, regression output, or some other task.



In the penguin classification task, we build a three-layer perceptron using scikit-learn's `MLPClassifier` from `sklearn.neural_network`.

```
from sklearn.neural_network import MLPClassifier

mlp_model = MLPClassifier(hidden_layer_sizes=(16), activation='relu', solver='adam',
                  alpha=0, batch_size=8, learning_rate='constant',
                  learning_rate_init=0.001, max_iter=1000,
                  random_state=123, n_iter_no_change=10)
mlp_model.fit(X_train_scaled, y_train)
```

The model is configured with:

- an input layer matching the number of features (6 per penguin),
- a hidden layer (*e.g.*, 16 neurons) to capture non-linear relationships, and
- an output layer with three nodes (one per penguin class), using `relu` activation for the hidden layer.

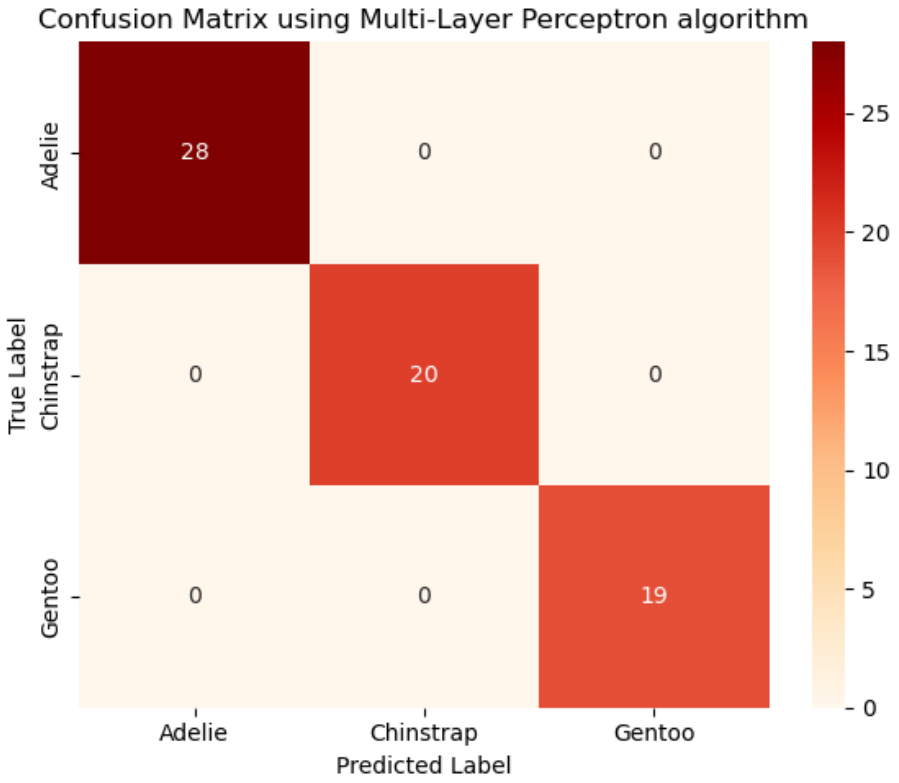
The hyperparameters used to construct this MLP are listed below:

- `adam`, the optimization algorithm used to update weight parameters.
- `alpha`, the L2 regularization term (penalty). Setting this to 0 disables regularization, meaning the model won't penalize large weights. This may cause overfitting if the dataset is small or noisy.
- `batch_size`, the number of samples per mini-batch during training. Smaller batches lead to more frequent updates (finer learning) but can increase noise and training time.
- `learning_rate`, specifies the learning rate schedule. "constant" means that the learning rate keeps fixed throughout training. Other options like "invscaling" or "adaptive" would adjust the learning rate during training.

- `learning_rate_init=0.001`, the initial learning rate (fixed here). A smaller value means slower learning, which may require more iterations but offers more stability.
- `max_iter`, the maximum number of training iterations (epochs).
- `random_state=123`, controls the random number generation for weight initialization and data shuffling, ensuring reproducible results.
- `n_iter_no_change=10`, if the validation score does not improve for 10 consecutive iterations, training will stop early. This is a form of early stopping to prevent overfitting or unnecessary computation.

After training the model, we evaluate its accuracy on the testing set and visualize the results by computing and plotting the confusion matrix.

```
y_pred_mlp = mlp_model.predict(X_test_scaled)

score_mlp = accuracy_score(y_test, y_pred_mlp)
print("Accuracy for Neural Network:", score_mlp)
print("\nClassification Report:\n", classification_report(y_test, y_pred_mlp))

cm_mlp = confusion_matrix(y_test, y_pred_mlp)
plot_confusion_matrix(cm_mlp, "Confusion Matrix using Multi-Layer Perceptron
algorithm", "5-confusion-matrix-mlp.png")
```



(Optional) Deep Neural Networks

MLP is a foundational neural network architecture, consisting of an input layer, one or more hidden layers, and an output layer. While MLP excels at learning complex patterns from tabular data, its shallow depth (typically 1-2 hidden layers) limits its ability to handle very high-dimensional or abstract data such as raw images, audio, or text.

To overcome these limitations, Deep Neural Network (DNN) extends the MLP framework by adding multiple hidden layers. These additional layers allow the model to learn highly abstract features through deep hierarchical representations: early layers might capture basic features (like edges or shapes), while deeper layers recognize complex objects or semantic patterns. This depth enables DNN to outperform traditional MLP in complex tasks requiring high-level feature extraction, such as computer vision and natural language processing.

DNN architectures

DNNs have specialized architectures designed to handle different types of data (*e.g.*, spatial, temporal, and sequential data) and tasks more effectively.

- A standard feedforward deep neural network consists of stacked fully connected layers
- **Convolutional neural networks** (CNNs) are particularly well-suited for image data. They use convolutional layers to automatically extract local features like edges, textures, and shapes, significantly reducing the number of parameters and improving generalization on visual tasks.
- **Recurrent neural network** (RNN) is designed for sequential data such as time series, speech, or natural language. RNNs include loops that allow information to persist across time steps, enabling the model to learn dependencies over sequences. More advanced versions, like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), address the limitations of basic RNNs by managing long-term dependencies more effectively.
- In addition to CNNs and RNNs, the **Transformer** architecture has emerged as the state-of-the-art in many language and vision tasks. Transformers rely entirely on attention mechanisms rather than recurrence or convolutions, enabling them to model global relationships in data more efficiently. This flexibility has made them the foundation of powerful models like BERT, GPT, and Vision Transformers (ViTs). These specialized DL architectures illustrate how tailoring the network design to the structure of the data can lead to significant performance gains and more efficient learning.

Here, we use the Keras API to construct a small DNN and apply it to the penguin classification task, demonstrating how even a compact architecture can effectively distinguish between penguin species (Adelie, Chinstrap, and Gentoo).

In this example, we exclude the categorical features `island` and `sex` from both the training and testing datasets. The target label `species` is then encoded using the `pd.get_dummies()` function in Pandas. Afterward, we split the data into training and testing sets and standardize the feature values to ensure consistent scaling during model training.

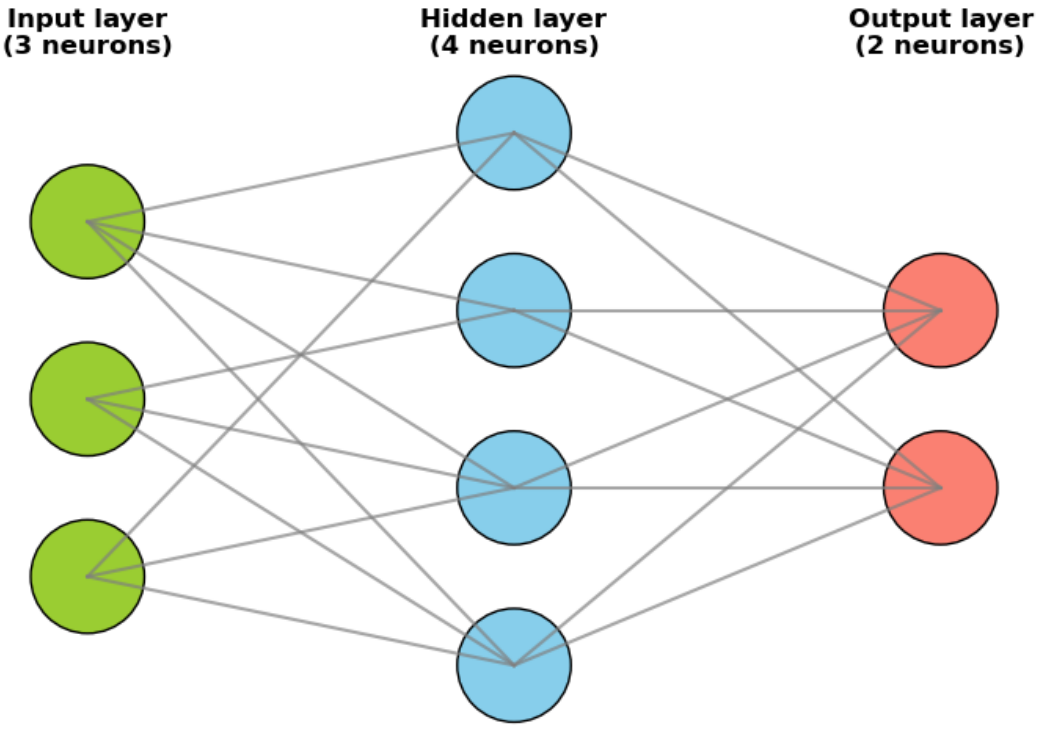
```
from tensorflow import keras

X = penguins_classification.drop(['species','island', 'sex'], axis=1)
y = penguins_classification['species'].astype('int')
y = pd.get_dummies(penguins_classification['species']).astype(np.int8)
y.columns = ['Adelie', 'Chinstrap', 'Gentoo']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)
print(f"Number of examples for training is {len(X_train)} and test is {len(X_test)}")

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

When building a DNN with Keras, there are two common approaches: using the `Sequential()` API step by step, or defining all layers at once within the `Sequential()` constructor. Here, we adopt the first approach, whereas the second approach is used in the Jupyter notebook to construct the same DNN.

- We start by creating an empty model with `keras.Sequential()`, which initializes a linear container for stacking sequential layers.
- Next, we define each layer separately using the `Dense` class, specifying the number of neurons and activation function for each layer.
- Finally, we add all the layers using `keras.Model()` to the sequential container, resulting in a trainable model.

```
from tensorflow.keras.layers import Dense, Dropout

dnn_model = Sequential()

input_layer = keras.Input(shape=(X_train_scaled.shape[1],)) # 4 input features

hidden_layer1 = Dense(32, activation="relu")(input_layer)
hidden_layer1 = Dropout(0.2)(hidden_layer1)

hidden_layer2 = Dense(16, activation="relu")(hidden_layer1)
#hidden_layer2 = Dropout(0.0)(hidden_layer2)

hidden_layer3 = Dense(8, activation="relu")(hidden_layer2)

output_layer = Dense(3, activation="softmax")(hidden_layer3) # 3 classes

dnn_model = keras.Model(inputs=input_layer, outputs=output_layer)
```

The `keras.layers.Dropout()` is a regularization technique in Keras used to reduce overfitting by randomly setting a fraction of input units to zero during training. For example, `Dropout(0.2)` means that 20% of the outputs of a specific layer will be randomly set to zero in each training step.



We can use `dnn_model.summary()` to print a concise summary of a DNN's architecture. It provides provides an overview of the model's layers, their output shapes, and the number of trainable parameters, making it easier to understand and debug the network.

```
dnn_model.summary()
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 4)	0
dense (Dense)	(None, 32)	160
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dense_2 (Dense)	(None, 8)	136
dense_3 (Dense)	(None, 3)	27

```
Total params: 851 (3.32 KB)
Trainable params: 851 (3.32 KB)
Non-trainable params: 0 (0.00 B)
```

Now that we have designed a DNN that, in theory, should be capable of classifying penguins, we need to specify two critical components before training: (1) a loss function to quantify prediction errors, and (2) an optimizer to adjust the model's weights during training.

- **Loss function**: For multi-class classification, we select categorical cross-entropy, which penalizes incorrect probabilistic predictions. In Keras, this is implemented via the `keras.losses.CategoricalCrossentropy` class. This loss function works naturally with the `softmax` activation function we applied in the output layer. For a full list of available loss functions in Keras, see the documentation.
- **Optimizer**: The optimizer determines how efficiently the model converges during training. Keras provides many options, each with its advantages, but here we use the widely adopted `Adam` (adaptive moment estimation) optimizer. Adam has several parameters, and the default values generally perform well, so we will use it with its defaults.

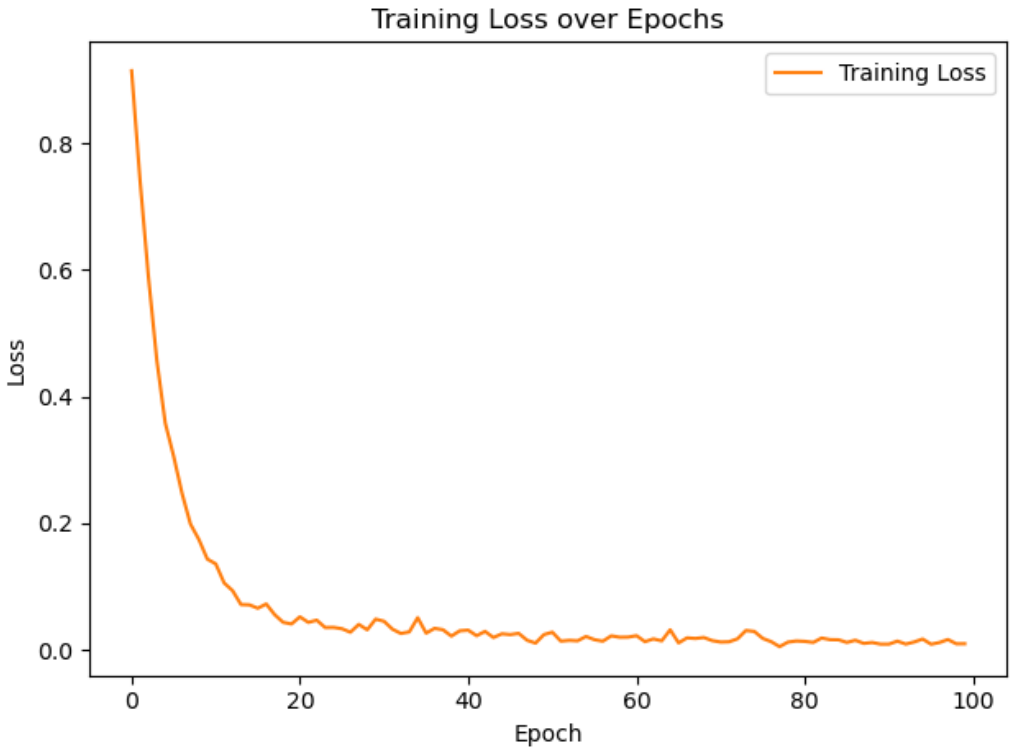
We use `model.compile()` to combine the chosen loss function and optimier before starting training.

```
from keras.optimizers import Adam

dnn_model.compile(optimizer='adam', loss=keras.losses.CategoricalCrossentropy())
```

Now we are ready to train the DNN model. Here, we vary only the number of `epochs`. One training epoch means that every sample in the training data has been shown to the neural network once and used to update its parameters. During training, we set `batch_size=16` to balance memory efficiency with gradient stability, and `verbose=1` to display a progress bar showing the loss and metrics for each epoch in real time.

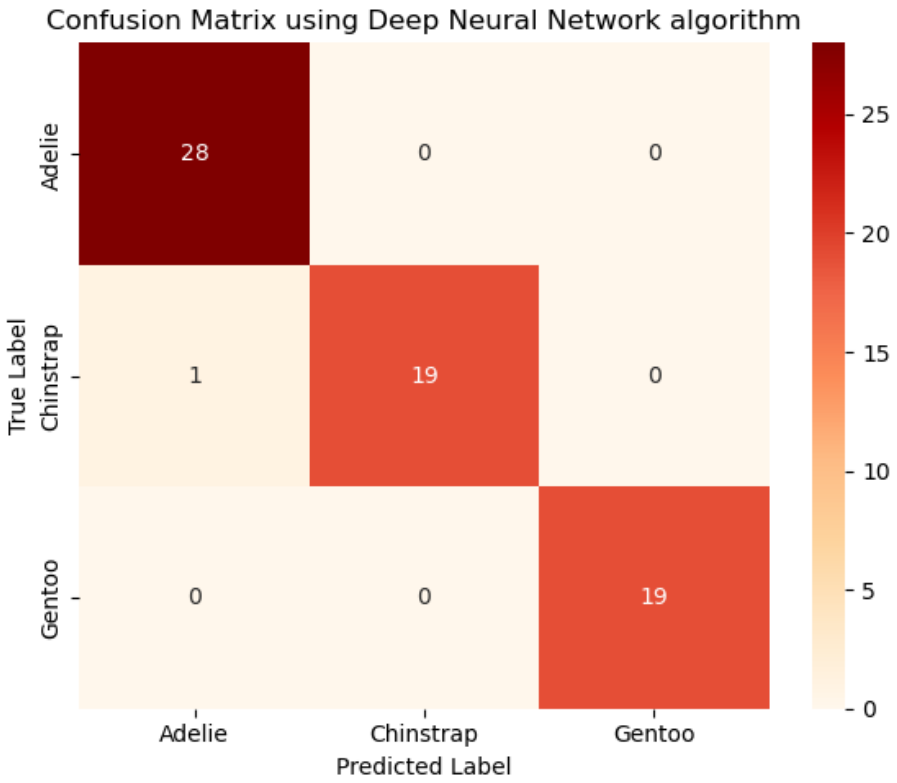
```
history = dnn_model.fit(X_train_scaled, y_train, batch_size=16, epochs=100, verbose=1)
```

The `.fit()` method returns a history object, which contains a history attribute holding the training loss and other metrics for each epoch. Plotting the training loss can provide valuable insight into how learning progresses. For example, we can use Seaborn to plot the training loss with epochs `sns.lineplot(x=history.epoch, y=history.history['loss'], c="tab:orange", label='Training Loss')`.



Finally, we evaluate the model's performance on the testing set by computing its accuracy and visualizing the results with a confusion matrix.

```
# predict class probabilities
y_pred_dnn_probs = dnn_model.predict(X_test_scaled)

# convert probabilities to class labels
y_pred_dnn = np.argmax(y_pred_dnn_probs, axis=1)
y_true = np.argmax(y_test, axis=1)

score_dnn = accuracy_score(y_true, y_pred_dnn)
print("Accuracy for Deep Neutron Network:", score_dnn)
print("\nClassification Report:\n", classification_report(y_true, y_pred_dnn))

cm_dnn = confusion_matrix(y_true, y_pred_dnn)
plot_confusion_matrix(cm_dnn, "Confusion Matrix using DNN algorithm", "5-confusion-
matrix-dnn.png")
```



Comparison of Trained Models

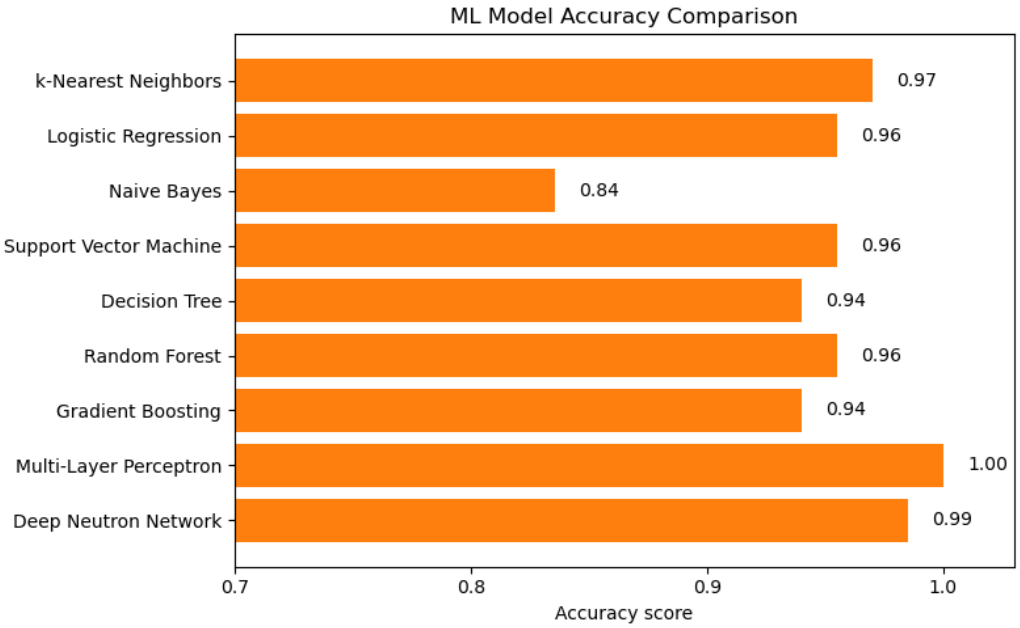
To evaluate the performance of different algorithms in classifying penguin species, we compare their accuracy scores and confusion matrices. The algorithms we the adopted in the previous sections include:

- Instance-based: k-Nearest Neighbors (KNN).
- Probability-based: Logistic Regression, and Naive Bayes.
- Hyperplane-based: Support Vector Machine (SVM).
- Tree-based methods: Decision Tree, Random Forest, and Gradient Boosting.
- Network-based models: Multi-Layer Perceptron (MLP) and Deep Neural Networks (DNN).

Each model was trained on the same training set and evaluated on a common testing set, with consistent preprocessing applied across all methods.

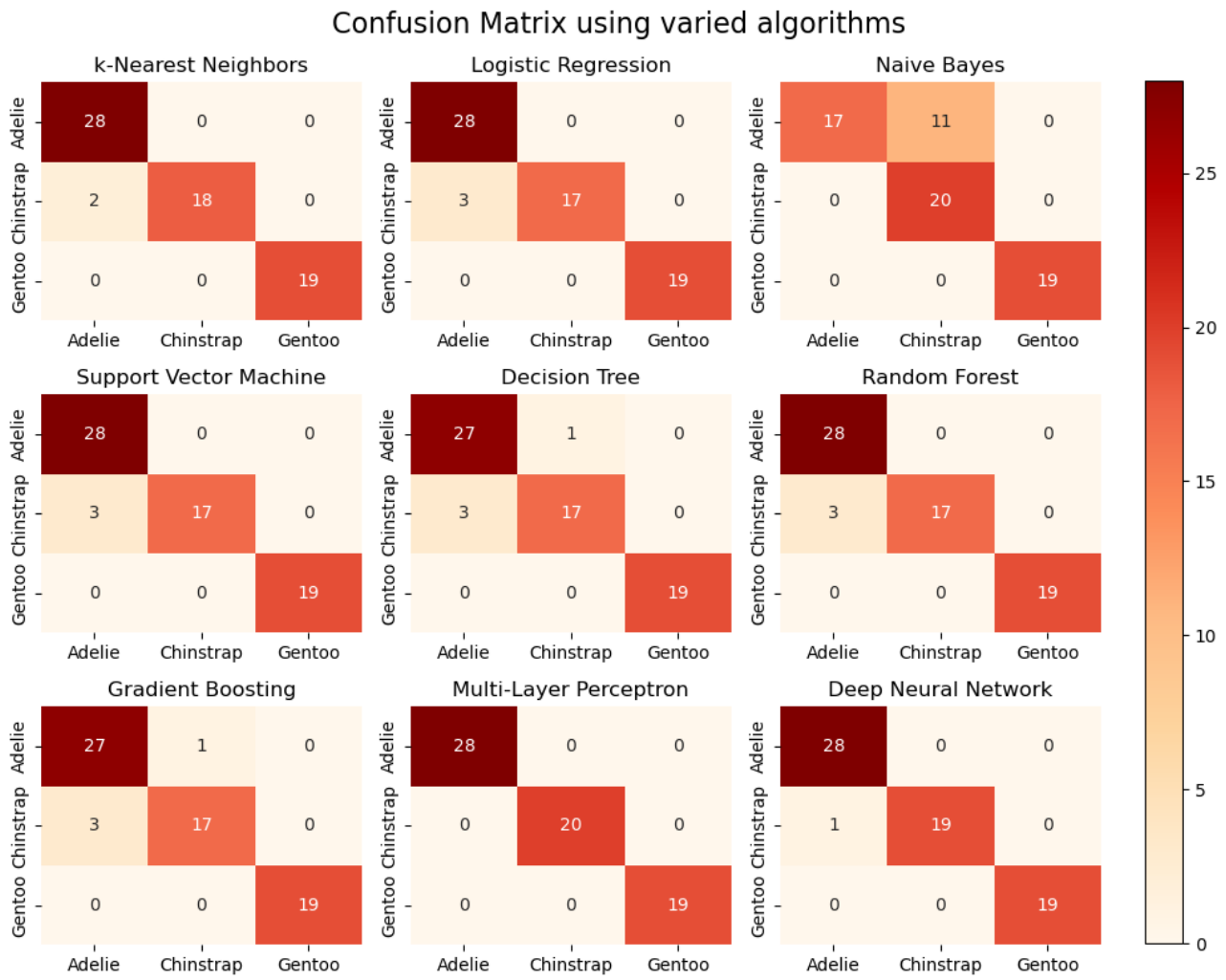
Performance under current training settings:

- MLP achieved the highest accuracy, demonstrating its effectiveness in capturing complex patterns and feature interactions in the Penguins dataset.
- Naive Bayes showed slightly lower accuracy, likely due to its strong independence assumption between features, which does not fully hold in this dataset.
- The other algorithms provided moderate performance.



The confusion matrices provided deeper insight into class-level prediction performance:

- MLP demonstrated well-balanced performance across all three penguin species.
- Naive Bayes, in contrast, confused Adelie and Chinstrap penguins, likely due to overlapping feature distributions between these species.
- other algorithms had a limited number of misclassifications, primarily between Adelie and Chinstrap.



➡ See also

- Introduction to Deep Learning

Keypoints

- Provided a fundamental introducton to classification tasks, covering basic concepts.
- Demonstrated essential steps for data preparation and processing using the Penguins dataset.
- Applied a range of classification algorithms — instance-based, probability-based, margin-based, tree-based, and neural network-based — to classify penguin species.
- Evaluated and compared model performance using metrics such as accuracy scores and confusion matrices.

Supervised Learning (II): Regression

Objectives

- Understand the fundamental concept of regression (overfitting, cross-validation, gradient search, ...)
- Distinguish between types of regression: simple vs. multiple regression, linear vs. non-linear regression, and other specialized regression methods.

- Perform regression tasks using representative algorithms (*e.g.*, k-NN, Linear Regression, Polynomial Regression, Support Vector Regression, Decision Tree, and Multi-Layer Perceptron)
- Evaluate model performance with metrics such as Root Mean Squared Error (RMSE) and the R-squared (R^2) score, and visualize predictive curves.

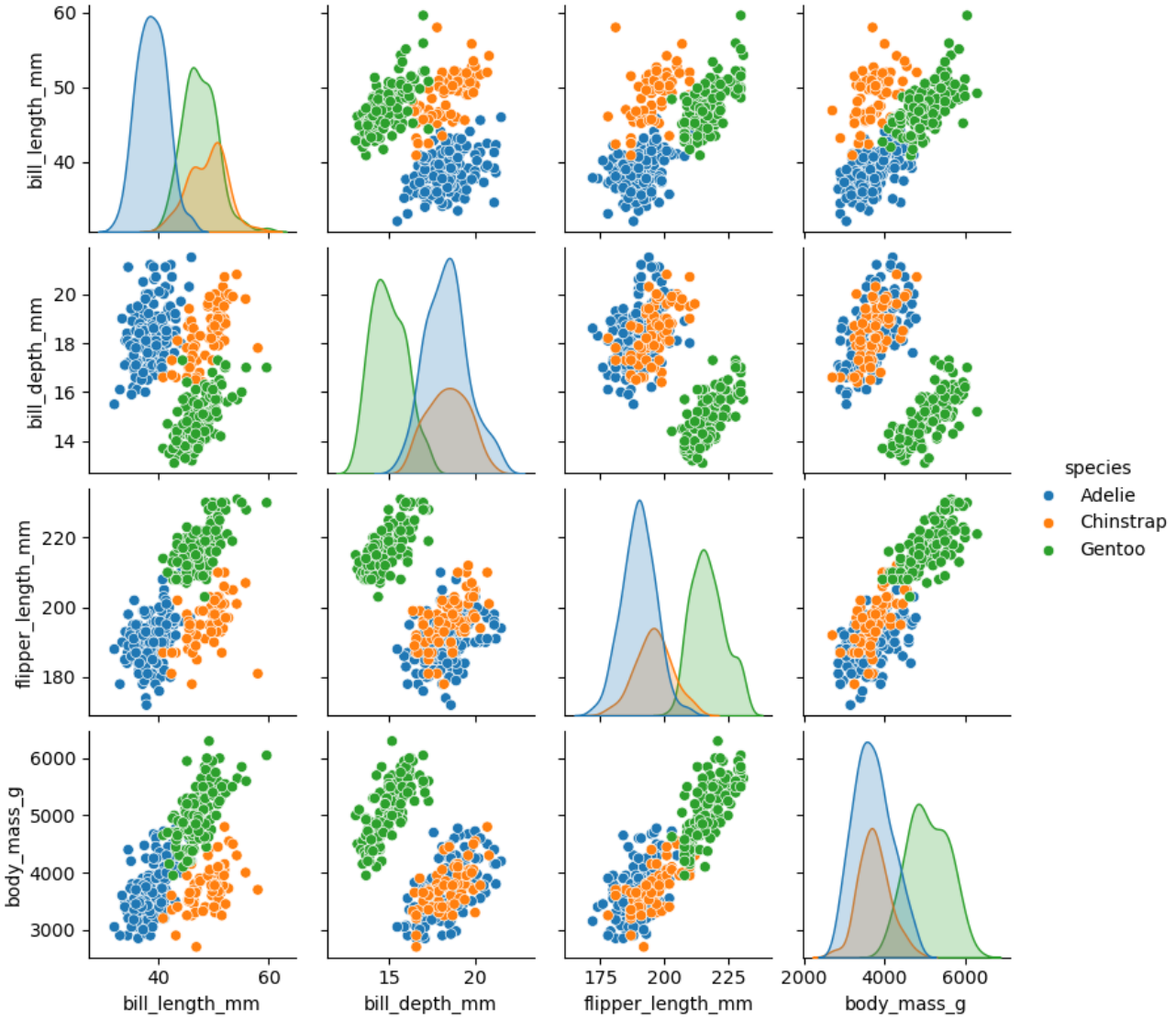
Instructor note

- 40 min teaching/demonstration
- 40 min exercises

Regression

Regression is a type of supervised machine learning task where the goal is to predict a continuous numerical value based on input features. Unlike classification, which assigns outputs to discrete categories, regression models produce real-valued predictions.

Although the Penguins dataset is most commonly used for classification tasks, it can also be applied to regression problems by choosing a continuous target variable. From the pairplot, we can observe a strong visual relationship between body mass and flipper length, indicating a clear positive correlation. Consequently, we select these two features for the regression task, aiming to estimate body mass based on flipper length.



Depending on the model construction approach, in this episode we explore a variety of regression algorithms to predict penguin body mass based on flipper length. These models are selected to represent different categories of machine learning approaches, ranging from simple, interpretable methods to more complex, flexible ones.

- **KNN Regression**: Predictions are made based on the average of the closest training samples. This non-parametric, instance-based model captures local patterns in the data effectively.
- **Linear Models**: Standard Linear Regression and Regularized Regression assume a straight-line relationship between flipper length and body mass. These models are interpretable and efficient, providing a solid baseline for comparison.
- **Non-linear Models**: To account for possible non-linear trends, we include Polynomial Regression with higher-degree terms and Support Vector Regression (SVR) with `rbf` kernels, which can model more complex relationships.
- **Tree-based Models**: Decision Trees, Random Forests, and Gradient Boosting offer robust alternatives by recursively partitioning the feature space or combining ensembles to improve accuracy and handle non-linearities effectively.
- **Neural Networks**: These serve as universal function approximators, capable of learning intricate patterns in the data, but typically require larger datasets and more computational resources.

Each model's performance is rigorously assessed using cross-validated metrics such as Root Mean Squared Error (RMSE) and R^2. The resulting predictive curves illustrate how well each model captures the biological relationship between flipper length and body mass.

Data Preparation

Similar to the procedures adopted in previous episodes, we follow the same preprocessing steps for the Penguins dataset, including handling missing values and detecting outliers. For the regression task, categorical features are not needed, so encoding them is unnecessary.

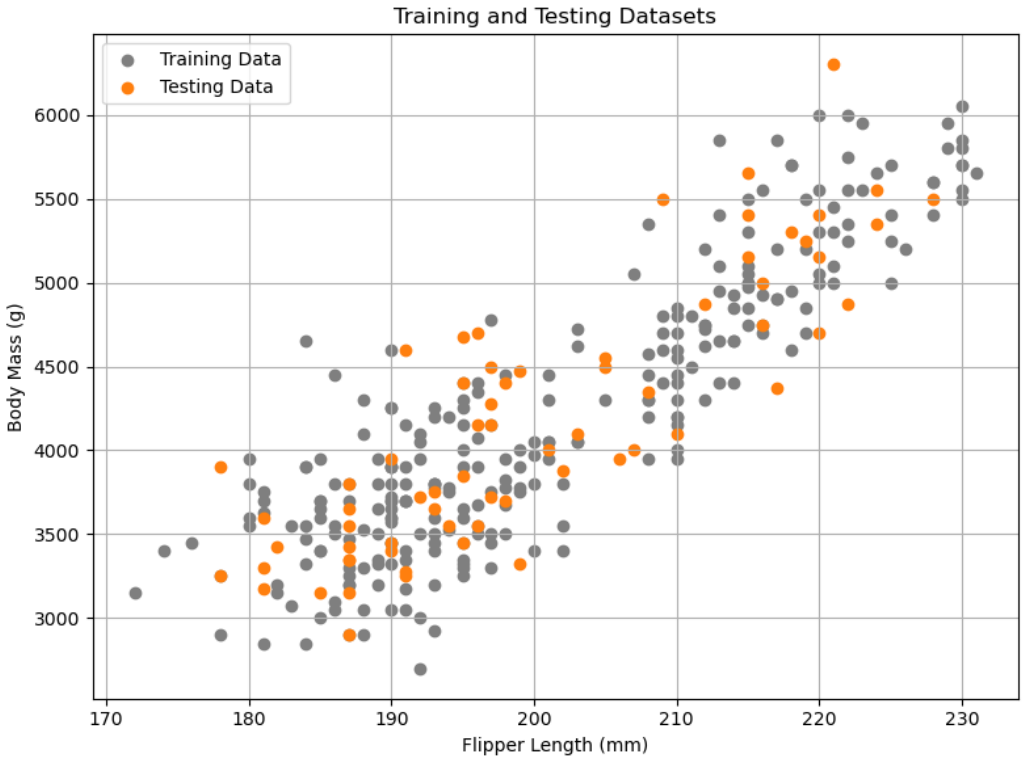
Data Processing

Below is the code script to extract `flipper_length_mm` and `body_mass_g` features from the main dataset.

```
X = penguins_regression[["flipper_length_mm"]].values
y = penguins_regression["body_mass_g"].values
```

In this episode, we first perform feature scaling, followed by splitting the data into training and testing sets. The `inverse_transform()` method reverts transformed data back to its original scale or format.

```
from sklearn.preprocessing import StandardScaler

# standardize feature and target
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).ravel()

X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled =
train_test_split(X_scaled, y_scaled, test_size=0.2, random_state=123)

X_train_orig = scaler_X.inverse_transform(X_train_scaled).ravel()
y_train_orig = scaler_y.inverse_transform(y_train_scaled.reshape(-1, 1)).ravel()
...
```



Training Model & Evaluating Model Performance

k-Nearest Neighbors (KNN)

We begin by applying the KNN algorithm to the penguin regression task, as illustrated in the code example below.

```
from sklearn.neighbors import KNeighborsRegressor

knn_model = KNeighborsRegressor(n_neighbors=5)
knn_model.fit(X_train_scaled, y_train_scaled)

# predict on test data
y_pred_knn_scaled = knn_model.predict(X_test_scaled)
y_pred_knn = scaler_y.inverse_transform(y_pred_knn_scaled.reshape(-1, 1)).ravel()
```

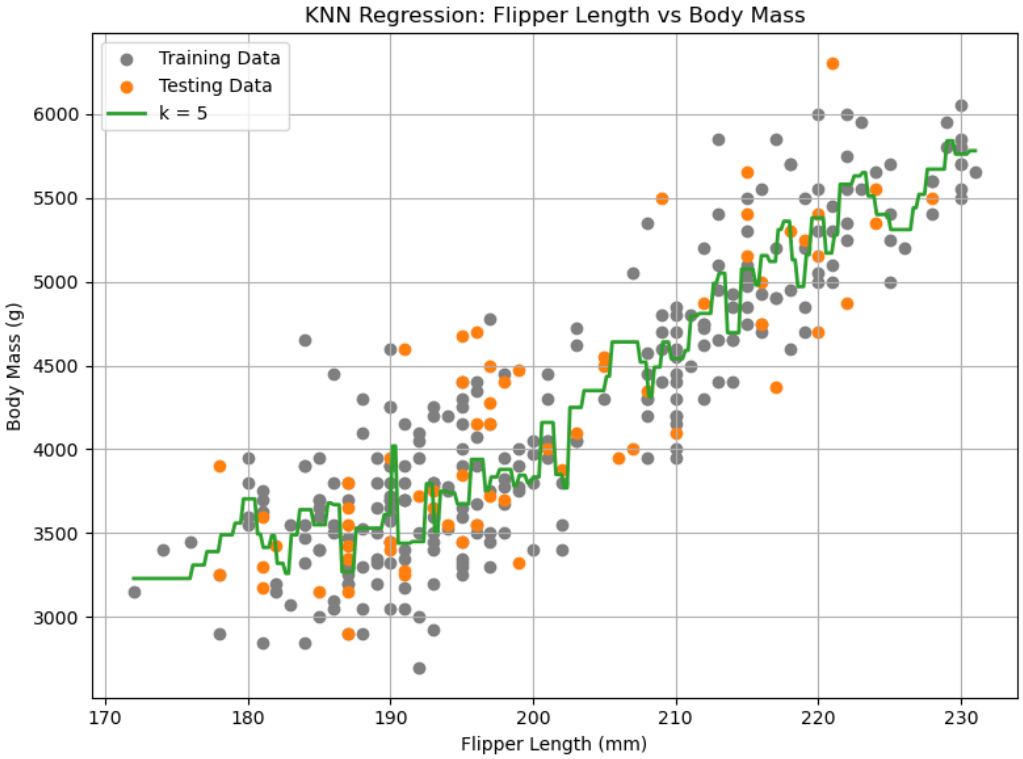
For the regression task, we use Root Mean Squared Error (RMSE) and R^2 score as evaluation metrics.

- RMSE measures the average magnitude of prediction errors, providing insight into how closely the model's predictions match the actual values
- R^2 score indicates the proportion of variance in the target variable that is explained by the model, reflecting its overall goodness of fit.

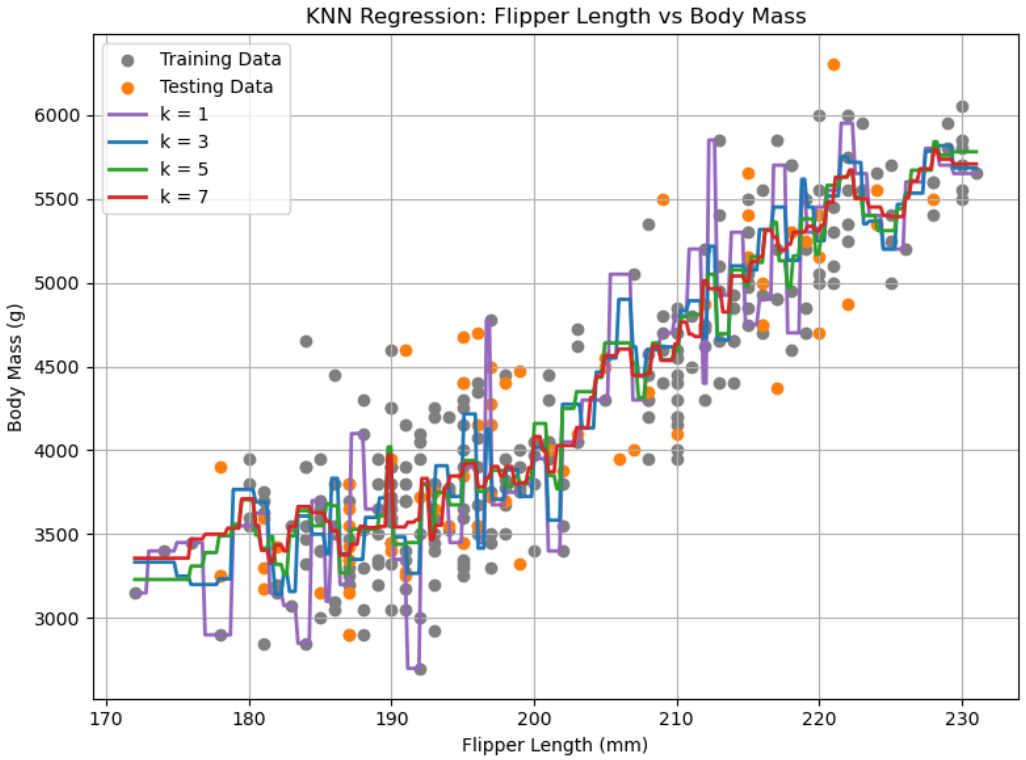
```
# evaluate model performance
from sklearn.metrics import root_mean_squared_error, r2_score

rmse_knn = root_mean_squared_error(y_test_orig, y_pred_knn)
r2_value_knn = r2_score(y_test_orig, y_pred_knn)
print(f"K-Nearest Neighbors RMSE: {rmse_knn:.2f}, R²: {r2_value_knn:.2f}")
```

To visualize the KNN algorithm for the regression task, we plot the **predictive curve**, which maps input values to predicted outputs. This curve illustrates how KNN responds to changes in a single feature. Since KNN is a non-parametric, instance-based method, it does not learn a fixed equation during training. Instead, predictions are made by averaging the target values of the k nearest training examples for each input.

The resulting predictive curve is typically piecewise-smooth, adapting to local patterns in the data. That is, the curve may bend or flatten depending on regions where data points are dense or sparse.



This makes the predictive curve an especially useful tool for assessing whether KNN is underfitting (*e.g.*, when k is large) or overfitting (e.g., when k is small). By adjusting k and observing changes in the curve's shape, we can intuitively tune the model's **bias-variance tradeoff**.



The bias-variance tradeoff

The bias-variance tradeoff is a fundamental concept in machine learning that describes the balance between model simplicity and model flexibility when trying to make accurate predictions.

- Bias measures how much a model's predictions systematically differ from the true values. High bias means the model is too simple and cannot capture the underlying patterns in the data, which leads to underfitting.
- Variance measures how much a model's predictions change when trained on different datasets. High variance means the model is too sensitive to small fluctuations in the training data, which leads to overfitting.

Linear Regression

Having explored a KNN regressor to predict penguin body mass from flipper length, we now turn to a fundamental and interpretable alternative: the Linear Regression model. While KNN makes predictions based on the average mass of the most similar observations, linear regression aims to identify a single, global linear relationship between the two variables. This approach fits a straight line through the data that minimizes the overall prediction error, producing a model that is typically less computationally intensive and offers immediate insight into the underlying trend.

The core concept of this linear model is a simple equation:

$$body\_mass = \beta_0 + \beta_1 \times flipper\_length$$

- the coefficient, β_1, represents the model's estimate of how much a penguin's body mass increases for each additional millimeter of flipper length

- the intercept, β_0, indicates the theoretical body mass for a penguin with a flipper length of zero. While this value is not biologically meaningful, it is necessary to position the line correctly.

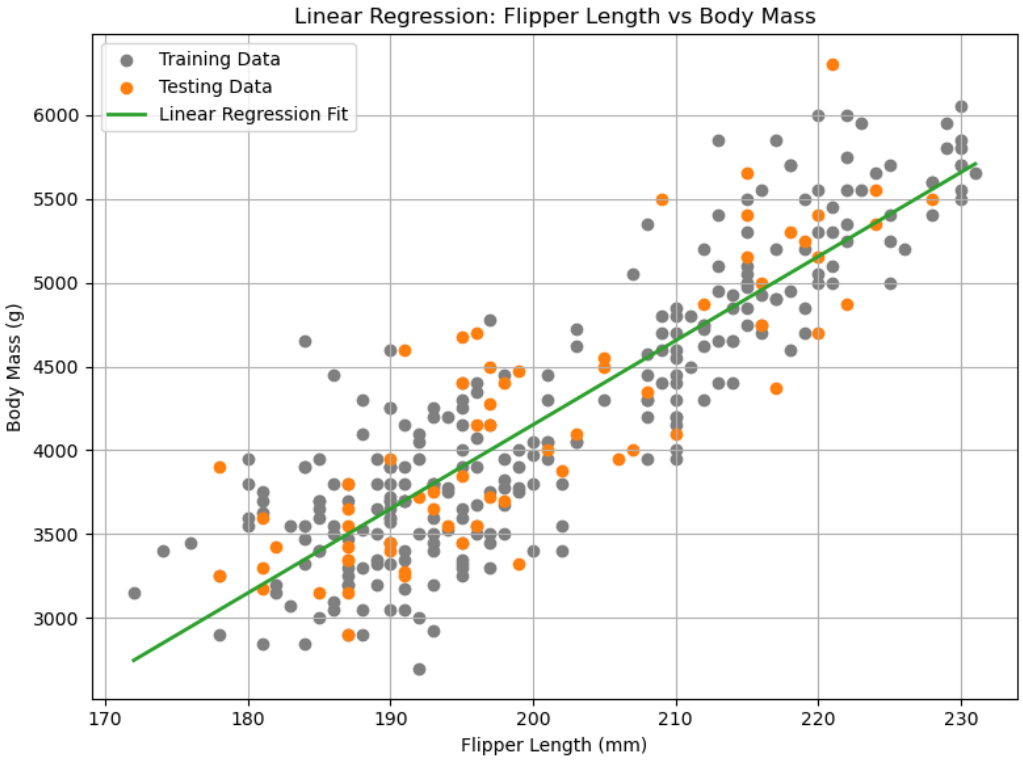
The fitted values of β_1 and β_0 can be accessed via `model.coef_` and `model.intercept_`, respectively. This equation provides a direct and interpretable rule: for any given flipper length, we can calculate a precise predicted body mass with given β_1 and β_0.

```
from sklearn.linear_model import LinearRegression

linear_model = LinearRegression()
linear_model.fit(X_train_scaled, y_train_scaled)
print(linear_model.coef_, linear_model.intercept_)

y_pred_linear_scaled = linear_model.predict(X_test_scaled)
y_pred_linear = scaler_y.inverse_transform(y_pred_linear_scaled.reshape(-1, 1)).ravel()
```

Once trained, we evaluate the linear regression model's predictive performance on the testing set using the same metrics: RMSE and R^2 score. In the Penguins dataset, a high R^2 indicates that flipper length is a strong predictor of body mass, while a low RMSE reflects precise predictions. These metrics also allow for direct comparison with KNN and other models, such as Polynomial Regression and tree-based methods that will be discussed below, highlighting situations where the simple linear assumption is sufficient and where it may fall short.

```
rmse_linear = root_mean_squared_error(y_test_orig, y_pred_linear)
r2_value_linear = r2_score(y_test_orig, y_pred_linear)
print(f"Linear Regression RMSE: {rmse_linear:.2f}, R²: {r2_value_linear:.2f}")
```

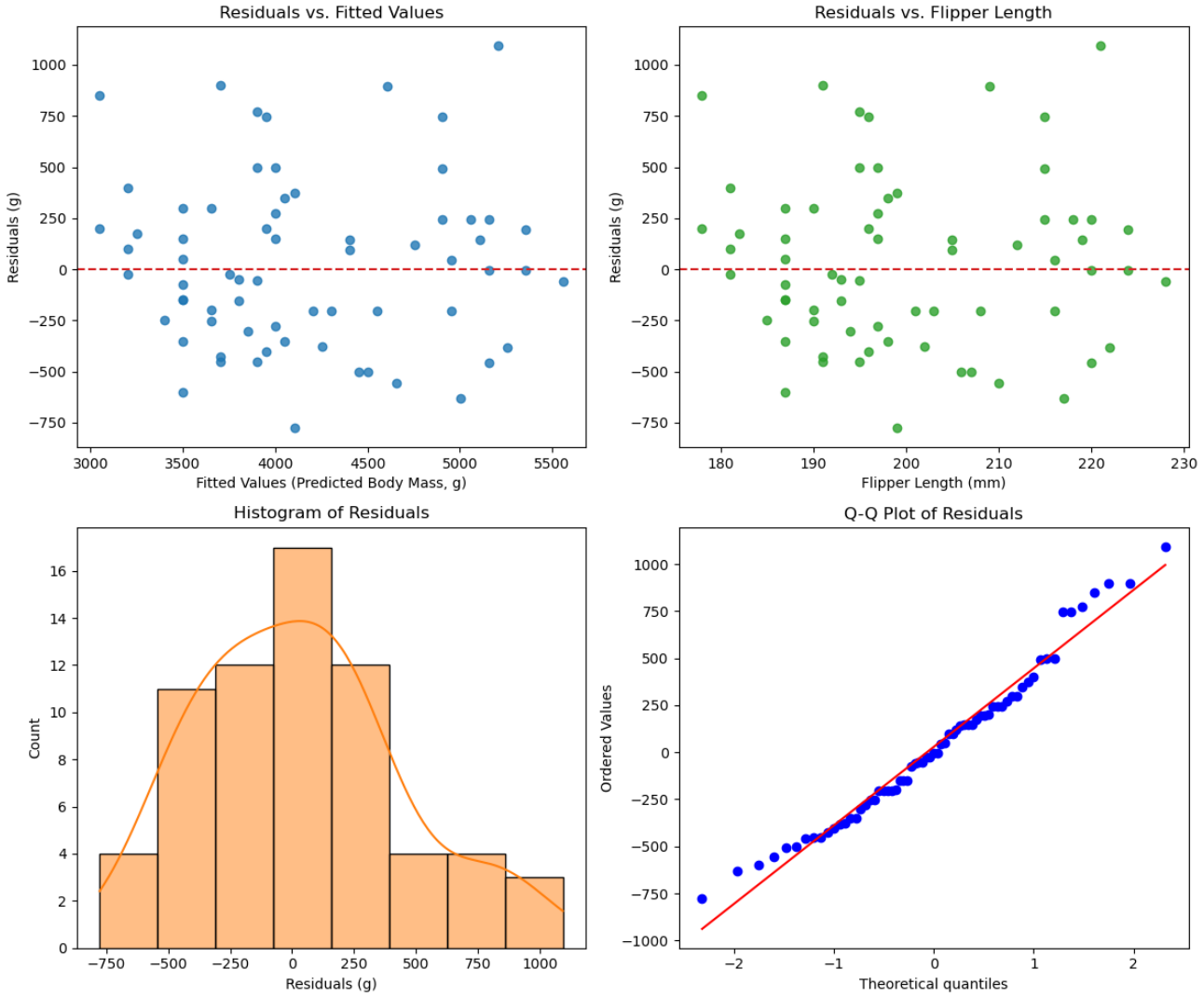
The resulting predictive curve is shown below.



Residual analysis

While metrics like RMSE and R-squared scores provide a high-level summary of model performance, **residual analysis** allows us to examine the model more deeply and verify the key assumptions of linear regression, ensuring that its conclusions are valid and reliable. Residuals are the differences between the observed body mass values and the values predicted by the model.

From the figure below, we can see that the residuals are randomly scattered around zero, with no apparent systematic patterns. This indicates that the linear model is largely unbiased and effectively captures the main trend between flipper length and body mass.



Note

If we notice certain patterns, *i.e.*, residuals that consistently increase or decrease with larger flipper lengths, it suggests that the relationship between body mass and flipper length might not be purely linear. Similarly, if the residuals fan out, showing greater spread at higher predicted values, it indicates heteroscedasticity, meaning the model errors are not consistent across the range of predictions. Such patterns imply that a simple linear regression model may not fully capture the variability in body mass.

Another key aspect of residual analysis involves assessing normality, as linear regression assumes normally distributed residuals for reliable inference. For the Penguins dataset, this can be evaluated using a histogram or a Q-Q (quantile-quantile) plot of the residuals.

The histogram of residuals illustrates the distribution of prediction errors across the dataset. In the Penguins dataset, these residuals should form a roughly symmetric, bell-shaped curve centered at zero. This indicates that the model is not systematically over-predicting or under-predicting body mass, and that most errors are relatively small, with fewer large deviations.

The Q-Q plot compares the distribution of the residuals to a theoretical normal distribution. On the plot, the x-axis represents the expected quantiles from a standard normal distribution, while the y-axis shows the quantiles of the observed residuals. If the residuals are normally distributed, the points should align closely with the diagonal reference line.

Overfitting and underfitting

In the previous section, we evaluated the Linear Regression model on the testing dataset and calculated metrics such as RMSE and R^2 to understand its predictive performance. While this gives a good indication of how well this model generalizes to unseen data, it only tells half the story.

To get a complete picture, it is important to also assess this model's performance on the training dataset and compare it with the testing results. This comparison is the primary diagnostic tool for identifying a model's fundamental flaw: whether it is learning the underlying signal or merely memorizing the data.

By calculating performance metrics like RMSE and R-squared for both training and testing datasets, we can identify potential issues such as overfitting and underfitting.

- **Overfitting** occurs when the model performs extremely well on the training data but poorly on the testing data. This indicates that the model has memorized the training patterns, including noise, rather than capturing the true underlying relationship.
- **Underfitting** happens when the model performs poorly on both training and testing datasets, suggesting that it is too simple to capture the relevant trends in the data.

```
# --- Training data predictions ---
y_pred_train_scaled = linear_model.predict(X_train_scaled)
y_pred_train = scaler_y.inverse_transform(y_pred_train_scaled.reshape(-1, 1)).ravel()

rmse_linear_train = root_mean_squared_error(y_train_orig, y_pred_train)
r2_linear_train = r2_score(y_train_orig, y_pred_train)

print(f"Linear Regression (Train) RMSE: {rmse_linear_train:.2f}, R²:
{r2_linear_train:.2f}")

# --- Testing data predictions ---
y_pred_test_scaled = linear_model.predict(X_test_scaled)
y_pred_test = scaler_y.inverse_transform(y_pred_test_scaled.reshape(-1, 1)).ravel()

rmse_linear_test = root_mean_squared_error(y_test_orig, y_pred_test)
r2_linear_test = r2_score(y_test_orig, y_pred_test)

print(f"Linear Regression (Test)  RMSE: {rmse_linear_test:.2f}, R²:
{r2_linear_test:.2f}")

# Linear Regression (Train) RMSE: 387.10, R²: 0.77
# Linear Regression (Test)  RMSE: 411.85, R²: 0.72
```

The trained Linear Regression model for predicting penguin body mass based on flipper length in the penguins dataset achieves comparable RMSE and R^2 scores on both the training and testing datasets, indicating a fairly good model.

✍️ Regularized Regression (Ridge and Lasso)

To address overfitting and underfitting, regularized regression methods, such as **Ridge** and **Lasso** regression, extend linear regression by adding a penalty term to the standard cost function. This penalty discourages the model from relying too heavily on any single feature or from becoming overly complex by forcing coefficient values to be small.

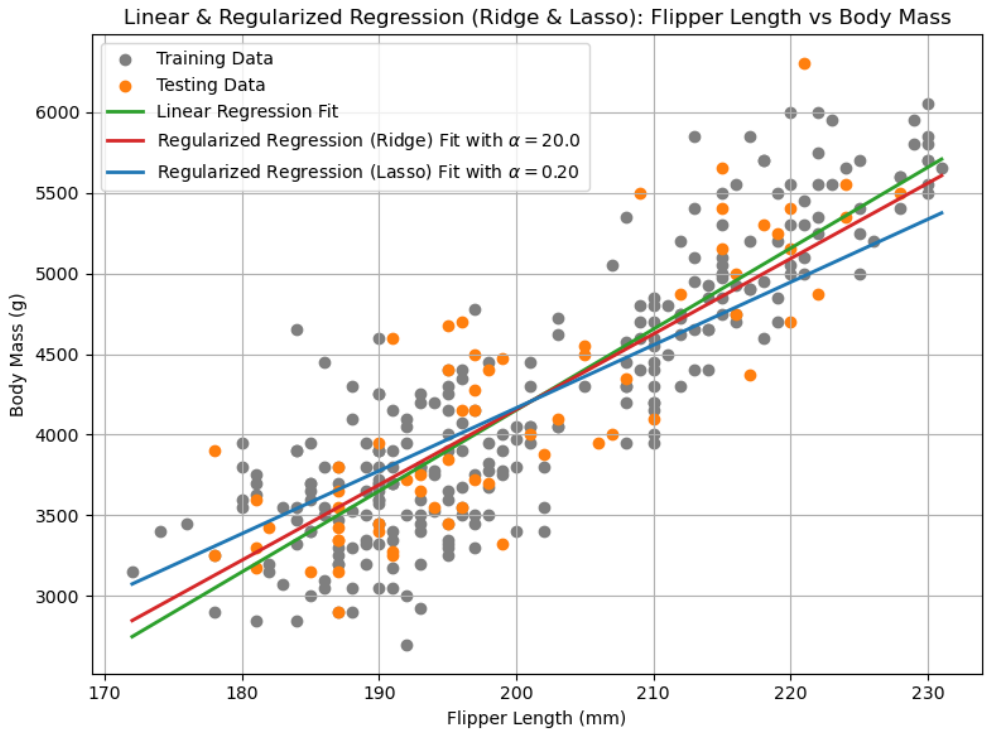
- **Ridge Regression** (L2 regularization) shrinks coefficients towards zero but never entirely eliminates them, which is highly effective for handling correlated features and improving stability. This is common in the penguins dataset when predictors like flipper length and bill length are correlated.
- **Lasso Regression** (L1 regularization) can drive some coefficients to exactly zero, effectively performing automatic feature selection and creating a simpler, more interpretable model. For instance, Lasso might retain flipper length while discarding less predictive features, improving generalization.

In this exercise (code examples are availalbe in the Jupyter Notebook), we will

- Train the Penguins dataset using Ridge and Lasso regression models, and compare their fitted parameters, RMSE, and R^2 scores.
- Conduct a residual analysis to evaluate whether the regularized regression models achieve better performance than the standard linear regression model.

Ridge Regression | Lasso Regression

```
from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=20.0)
ridge_model.fit(X_train_scaled, y_train_scaled)
print(ridge_model.coef_, ridge_model.intercept_)

y_pred_ridge_scaled = ridge_model.predict(X_test_scaled)
y_pred_ridge = scaler_y.inverse_transform(y_pred_ridge_scaled.reshape(-1,
1)).ravel()

rmse_ridge = root_mean_squared_error(y_test_orig, y_pred_ridge)
r2_value_ridge = r2_score(y_test_orig, y_pred_ridge)
print(f"Regularized Regression (Ridge) RMSE: {rmse_ridge:.2f}, R²:
{r2_value_ridge:.2f}")
```



Polynomial Regression

In the previous section, we assumed that penguin body mass is linearly proportional to flipper length, and after training, we have verified that this assumption holds reasonably well. However, for other applications, if two variables are explicitly not linearly related, and a simple linear model may fail to capture the underlying patterns. In such cases, we can resort to polynomial regression to capture non-linear relationship by including higher-degree terms of the predictor variable.

In the context of the Penguins dataset, polynomial regression extends linear regression by modeling body mass as a polynomial function of flipper length with the formula as

$$body\_mass = \beta_0 + \beta_1 \times flipper\_length + \beta_2 \times flipper\_length^2 + \beta_3 \times flipper\_length^3 + ...$$

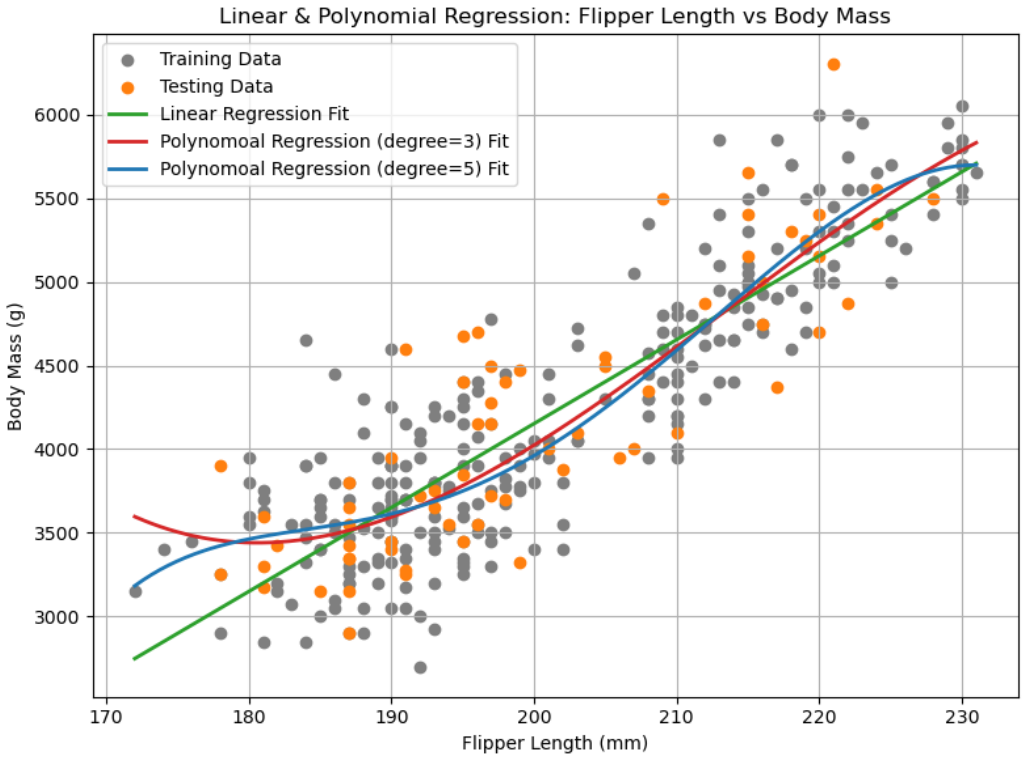
This approach allows the model to fit a curved relationship, which might be relevant if, for example, body mass increases more rapidly with flipper length for larger penguins, as seen in species like Gentoo.

The process of training a Polynomial Regression model is similar to Linear Regression. We first transform the original feature (flipper length) by adding polynomial terms (*e.g.*, $flipper_length^2$ and higher-degree terms), creating a feature matrix that the Polynomial Regression model uses to fit a non-linear curve while still employing Linear Regression techniques on the transformed features.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

degree3=3
poly3_model = make_pipeline(PolynomialFeatures(degree3), LinearRegression())
poly3_model.fit(X_train_scaled, y_train_scaled)
print(poly3_model.named_steps['linearregression'].coef_,
poly3_model.named_steps['linearregression'].intercept_)

y_pred_poly3_scaled = poly3_model.predict(X_test_scaled)
y_pred_poly3 = scaler_y.inverse_transform(y_pred_poly3_scaled.reshape(-1, 1)).ravel()

rmse_poly3 = root_mean_squared_error(y_test_orig, y_pred_poly3)
r2_value_poly3 = r2_score(y_test_orig, y_pred_poly3)
print(f"Polynomial Regression (degree={degree3}) RMSE: {rmse_poly3:.2f}, R²:
{r2_value_poly3:.2f}")
```

The Polynomial Regression model is trained by minimizing the sum of squared and cubic residuals, and performance is evaluated using metrics like RMSE and R^2. Compared with Linear Regression, we see that a third-degree polynomial regression model provides a marginally better fit than the simple linear model.

Below we present the predictive curves for Polynomial Regression models with degrees 3 and 5, alongside the curve for Linear Regression. In addition, we report the evaluation metrics (RMSE and R^2) on the testing dataset to provide a quantitative comparison.



Performance metrics (RMSE and R^2) on the testing dataset

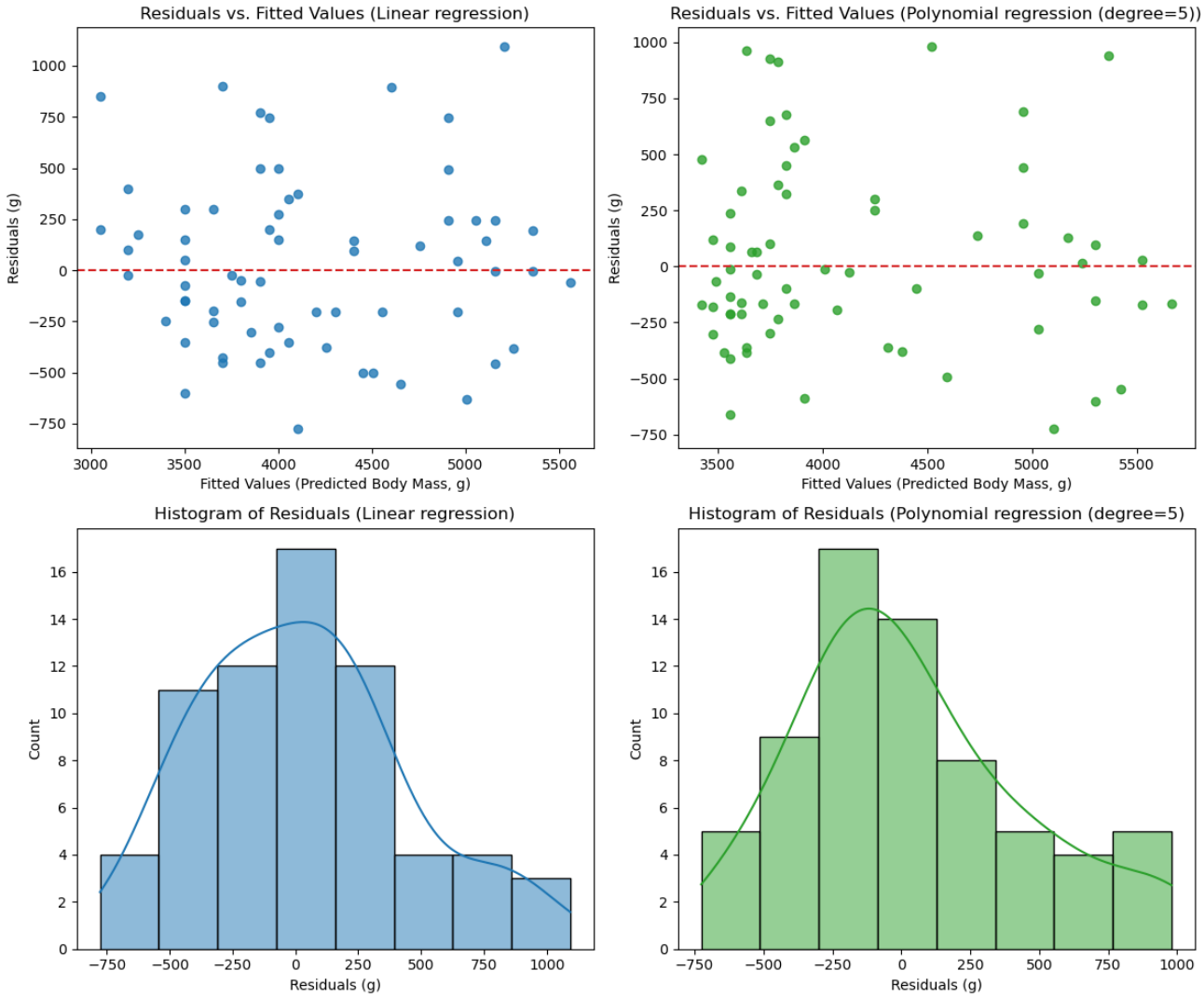
Model	RMSE	R^2
Linear Regression	411.85	0.72

Model	RMSE	R^2
Ridge Regression	414.18	0.72
Lasso Regression	437.19	0.69
Polynomial Regression (degree=3)	407.47	0.73
Polynomial Regression (degree=5)	415.55	0.72
Support Vector Regression	424.24	0.70

Caution

It is crucial to approach this added complexity with caution. While higher-degree polynomials can achieve very close fits to the training data, they are also highly prone to overfitting. For example, a model with a very high degree (*e.g.*, degree = 10) may contort itself to pass through nearly every training point, capturing random noise rather than the true underlying biological relationship. As a result, such a model would likely perform poorly on unseen test data, sacrificing generalizability for apparent short-term accuracy.

Additionally, residual analysis can provide further information on the model performance.



Compared with Linear Regression, the Polynomial Regression model with degree = 5 shows signs of overfitting, as evidenced by systematic deviations in the residuals and the asymmetric distribution of prediction errors across the dataset.

Overall, Polynomial Regression serves as a simple yet powerful extension of Linear Regression. It enables us to capture non-linear relationships while still benefiting from the interpretability and computational efficiency of a linear framework applied to transformed features. The key challenge lies in selecting the appropriate polynomial degree. Too low a degree may underfit the data, missing important trends, while too high a degree risks memorizing noise and overfitting. To strike the right balance, rigorous evaluation techniques — such as cross-validation on the training set — are typically used to identify the optimal degree, followed by a final assessment on the testing set. By carefully tuning complexity, polynomial regression can deliver genuine improvements in predictive accuracy and provide a more faithful representation of the often non-linear patterns found in the natural world.

Support Vector Machine

In the previous episode, we introduced the SVM model, which is widely recognized for its effectiveness in classification tasks by finding an optimal hyperplane that maximizes the margin between classes. Here, we adapt the same principles to regression through **Support Vector Regression** (SVR). Unlike Linear Regression or Polynomial Regression, which minimize squared errors, SVR builds on the concepts of margins and support vectors, and aims to find a tube (or a channel) (ε-insensitive zone) of a specified width that captures as many data points as possible, while only the points lying outside this tube (the support vector) affect the model's predictions.

The core challenge SVR faces is that, by its fundamental nature, it seeks a linear relationship (a flat hyperplane). In many real-world problems, such as predicting a penguin's body mass from its flipper length, the underlying relationship, while roughly linear, may contain subtle non-linear patterns that a straight line cannot fully capture.

Rather than manually generating polynomial features, which can be computationally expensive and impractical in high-dimensional spaces, kernel functions are used to capture non-linear relationships by implicitly projecting (rather than explicitly transforming) the input data into higher-dimensional feature spaces.

Note

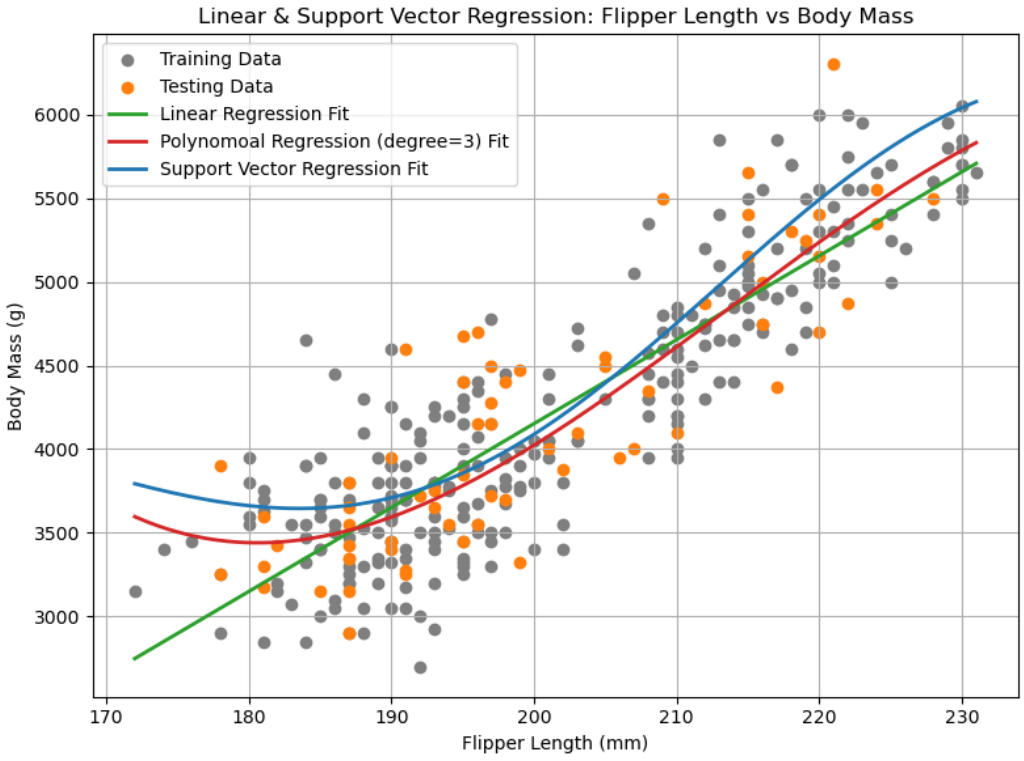
Several kernel types are commonly used in SVR, each imparting different characteristics to the model:

- **Linear Kernel**: The simplest kernel, which does not perform any transformation and assumes a linear relationship between features and the target variable. It is fast and interpretable but lacks flexibility for modeling complex patterns.

- **Polynomial Kernel**: This kernel enables the model to fit polynomial curves of a specified degree d with contronable flexibility. While more adaptable than a linear kernel, it can be sensitive to the chosen degree and may perform poorly when extrapolating beyond the training range.
- **Radial Basis Function (RBF) Kernel**: The most widely used kernel for non-linear problems, capable of generating highly flexible and smooth curves. It is versatile and effective for capturing complex relationships in the data.

For the penguin regression task, we use the RBF kernel in the SVR model to capture potential non-linear relationships between flipper length and body mass that a simple linear model might not be able to detect.

```
from sklearn.svm import SVR

svr_model = SVR(kernel='rbf', gamma=0.1, C=100.0, epsilon=1.0)
svr_model.fit(X_train_scaled, y_train_scaled)

y_pred_svr_scaled = svr_model.predict(X_test_scaled)
y_pred_svr = scaler_y.inverse_transform(y_pred_svr_scaled.reshape(-1, 1)).ravel()

rmse_svr = root_mean_squared_error(y_test_orig, y_pred_svr)
r2_value_svr = r2_score(y_test_orig, y_pred_svr)
print(f"Support Vector Regression RMSE: {rmse_svr:.2f}, R²: {r2_value_svr:.2f}")
```



For the Penguins dataset, SVR can potentially outperform Linear Regression if the relationship between flipper length and body mass is non-linear, as it can flexibly adapt to complex patterns without requiring explicit polynomial features. However, in this regression task, SVR with a (non-linear) RBF kernel underperforms compared to the Linear Regression model. There are two main reasons for this:

- The relationship between flipper length and body mass is fundamentally linear or only mildly non-linear, so the flexibility of SVR is not necessary.
- The hyperparameters (`gamma` , `C` , `epsilon` in `svr_model = SVR(kernel='rbf', gamma=0.1, C=100.0, epsilon=1.0)`) used for the SVR model may not be optimal. In this case, tuning the hyperparameters using techniques like grid search or cross-validation could improve performance.

✍️ Tuning hyperparameter

In this exercise (code examples are availalbe in the Jupyter Notebook, we will use **grid search** combined with **cross-validation** to find the optimal hyperparameters for the SVR model (code example is available in the Jupyter Notebook). We will:

- Compare RMSE and R^2 values to evaluate predictive performance.
- Plot predictive curves to visually assess how well the model fits the data.

❗ Grid search and Cross validation

Grid search is a method used to find the best combination of hyperparameters for a ML model. It will search all possible combinations of the hyperparameter values you specify, trains the model for each combination, and evaluates it using a validation set. After testing all combinations, it picks the set of hyperparameters that gives the best performance.

Cross-validation is a method to check how well a model will perform on unseen data.

- Instead of just splitting the data once into training and testing sets, cross-validation splits the data into several parts (folds).
- The model is trained on some folds and tested on the remaining fold. This process is repeated so that every fold gets a turn as the test set.
- The performance scores from all folds are averaged, giving a more reliable estimate of how the model will do in real situations.
- Here is a one example: The dataset is split into 5 parts. The model trains on 4 parts and tests on 1 part. This is repeated 5 times, each time with a different test part.

Decision Tree

In addition to instance-based models such as KNN and margin-based models like SVR, we can also apply tree-based methods for the regression task to predict a penguin's body mass from its flipper length. One of the most intuitive and interpretable approaches in this family is the **Decision Tree Regressor**.

A Decision Tree Regressor is a non-linear model that partitions the feature space (flipper length) into distinct regions based on feature thresholds and assigns a constant value (the average body mass) to each region. For the penguin regression task, the Decision Tree

Regressor recursively splits the dataset into groups of penguins with similar flipper lengths. At each split, the model selects the threshold that minimizes the variance of body mass within the resulting groups. This recursive process continues until stopping criteria are met, such as reaching a maximum tree depth or a minimum number of samples per leaf.

Below is a code example demonstrating the Decision Tree Regressor (`max_depth = 3`) applied to the penguins regression task.

```
from sklearn.tree import DecisionTreeRegressor

dt3_model = DecisionTreeRegressor(max_depth=3, random_state=123)
dt3_model.fit(X_train_scaled, y_train_scaled)

y_pred_dt3_scaled = dt3_model.predict(X_test_scaled)
y_pred_dt3 = scaler_y.inverse_transform(y_pred_dt3_scaled.reshape(-1, 1)).ravel()

rmse_dt3 = root_mean_squared_error(y_test_orig, y_pred_dt3)
r2_value_dt3 = r2_score(y_test_orig, y_pred_dt3)
print(f"Decision Tree (depth=3) RMSE: {rmse_dt3:.2f}, R²: {r2_value_dt3:.2f}")
```

Predictions for a new penguin are straightforward. Once the tree is built, the model follows the decision path down the tree according to the penguin's feature values until it reaches a leaf node. The predicted value is then the mean (or median) body mass of the training samples in that leaf. For example, if a leaf node contains 15 penguins with an average body mass of 3850 grams, any new penguin whose features lead it to this leaf will be predicted to have a mass of 3850 g.

When applying a Decision Tree Regressor to the penguin regression tesk, the tree depth plays a crucial role in shaping the fitted curve. With a relatively shallow tree, such as depth = 3, the model makes only a few splits on flipper length, resulting in broad, step-like regions where

body mass is predicted as the average within each group. This provides a coarse approximation of the relationship, capturing the general trend but missing finer variations.

Increasing the tree depth to 5 allows for more splits, creating narrower regions and a fitted curve that follows the data more closely. While this improves flexibility and reduces bias, it also increases the risk of capturing noise in the training set, leading to overfitting. Comparing fitted curves at different depths illustrates **the classic trade-off in decision trees: shallow trees may underfit, while deeper trees may fit the training data too closely**.

✍ Random Forest and Gradient Boosting Regressions

We have discussed the limitations of Decision Tree algorithm, which can be mitigated using powerful ensemble methods such as Random Forest and Gradient Boosting. In this exercise (code examples are availalbe in the Jupyter Notebook), we will:

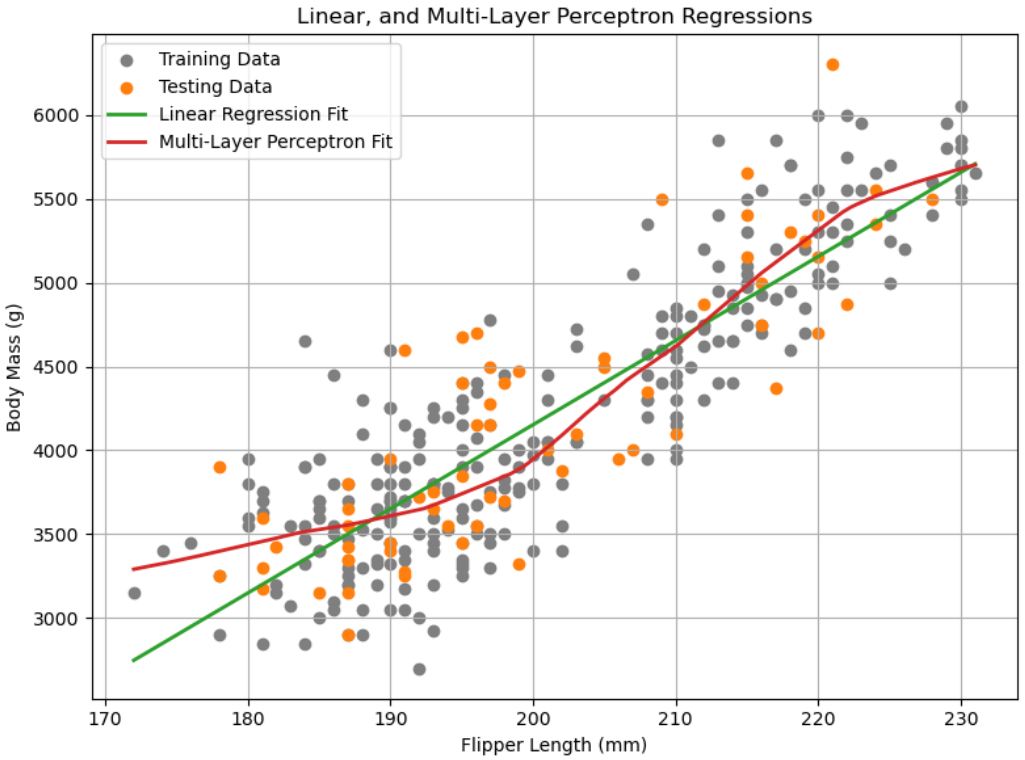
- Apply Random Forest and Gradient Boosting Regressors to the penguin regression task using initial (arbitrary) hyperparameters.
- Optimize hyperparameters via grid search and cross-validation to improve predictive performance.
- Plot predictive curves to visually evaluate how well each model fits the data.

Multi-Layer Perceptron

For this penguins task, we will explore implementations using three popular frameworks: the user-friendly scikit-learn, the high-level deep learning library Keras, and the more granular, research-oriented PyTorch.

In Scikit-learn, the `MLPRegressor` class offers a convenient interface for training small- to medium-sized neural networks, requiring minimal configuration while still providing flexibility for most regression tasks.

```
from sklearn.neural_network import MLPRegressor

mlp_model = MLPRegressor(hidden_layer_sizes=(32, 16, 8), activation='relu',
                         solver='adam', max_iter=5000, random_state=123)
mlp_model.fit(X_train_scaled, y_train_scaled)

y_pred_mlp_scaled = mlp_model.predict(X_test_scaled)
y_pred_mlp = scaler_y.inverse_transform(y_pred_mlp_scaled.reshape(-1, 1)).ravel()

rmse_mlp = root_mean_squared_error(y_test_orig, y_pred_mlp)
r2_value_mlp = r2_score(y_test_orig, y_pred_mlp)
print(f"Multi-Layer Perceptron RMSE: {rmse_mlp:.2f}, R²: {r2_value_mlp:.2f}")
```



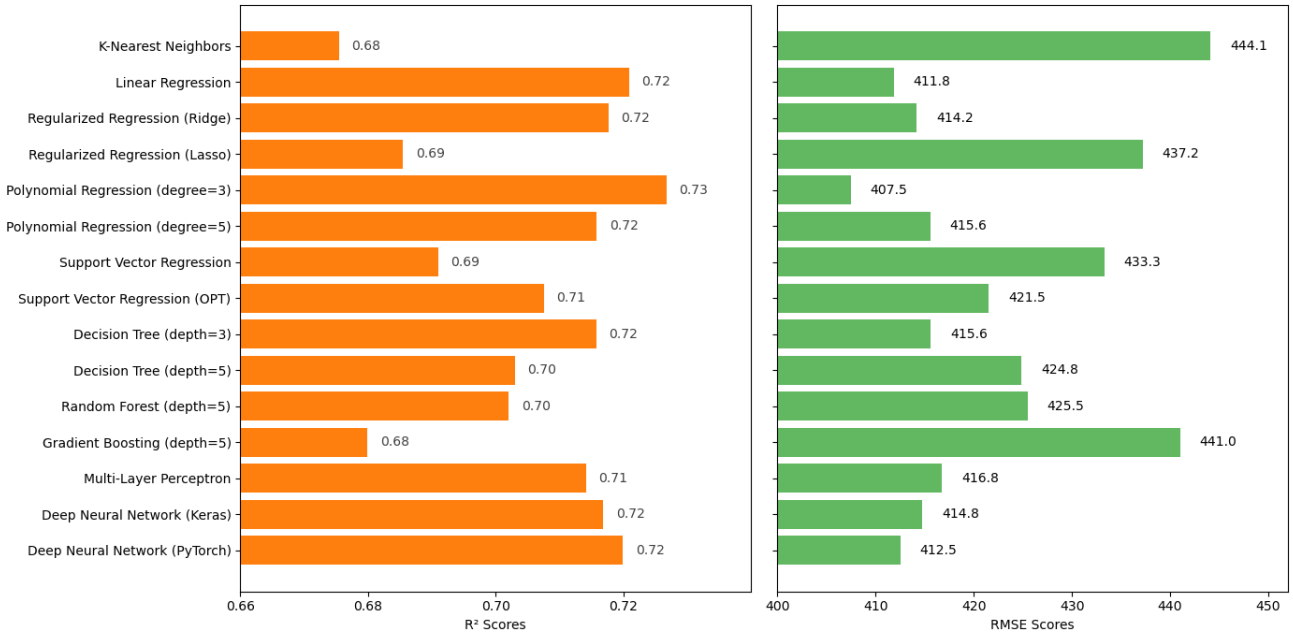
For greater control and scalability, frameworks like Keras (built on TensorFlow) and PyTorch allow us to design custom neural network architectures. We can specify the number of hidden layers, the number of neurons per layer, activation functions (*e.g.*, ReLU or tanh), and optimization algorithms (*e.g.*, stochastic gradient descent or Adam). These frameworks also offer tools for monitoring training, adjusting learning rates, and preventing overfitting through techniques such as regularization or dropout.

✍️ DNNs using Keras (TensorFlow) and PyTorch

Code examples are availalbe in the Jupyter Notebook:

- Construct DNNs using Keras and PyTorch.
- Apply DNNs to the penguin regression task using given hyperparameters.
- Optimize hyperparameters listed below to improve predictive performance
  - architecture hyperparameters
    - number of layers
    - number of neurons per layer
    - activation functions (*e.g.*, `ReLU`, `tanh`, `sigmoid`)
  - training hyperparameters
    - optimizers (*e.g.*, SGD, Adam, RMSprop)
    - learning rate
    - batch size
    - number of epochs
  - regularization hyperparameters
    - dropout rate
    - early stopping parameters
- Plot predictive curves for visualization

Comparison of Trained Models



Summary of regression models

- Best performance: Polynomial Regression (degree=3) gave the lowest RMSE (407.47) and the highest R^2 (0.73), slightly outperforming Linear Regression.
- Strong performers: Linear Regression, Ridge, Polynomial Regression (degree=5), Decision Tree (depth=3), MLP, and Deep Neural Networks (Keras, PyTorch) showed similar results (RMSE ≈ 412–415, R^2 ≈ 0.71–0.72).
- Moderate performers: Random Forest (depth=5), Decision Tree (depth=5), and SVR (optimized) performed decently (R^2 ≈ 0.70–0.72) but not better than simpler models.
- Weaker performers: Plain SVR, Lasso, Gradient Boosting, and KNN trailed behind with higher RMSEs (>430) and lower R^2 values (≈0.68–0.69).

➡ See also

- Hyperparameter optimization
- Grid search
- Introduction to Cross-Validation in Machine Learning

❗ Keypoints

- We explored regression as a supervised learning task for predicting penguins body mass from their flipper length.
- Starting with simple models like Linear Regression, we gradually introduced more advanced approaches, including Polynomial Regression, Support Vector Regression, tree-based models, and neural networks.
- All models were evaluated using matrics including RMSE and R^2 scores, and visualized with predictive curves.

- Simple models (Linear and Polynomial Regression) performed as well as or better than complex models (SVR, trees, ensembles, neural network-based models).
- This indicates that the relationship between flipper length and body mass is mostly linear, with mild non-linear patterns.

Unsupervised Learning (I): Clustering

Objectives

- Explain what unsupervised learning is and how clustering fits into it.
- Understand main ideas behind centroid-based (K-Means), hierarchical, density-based (DBSCAN), model-based (GMM), and graph-based (Spectral Clustering) methods.
- Apply representative clustering algorithms in practice.
- Understand how to evaluate clustering quality using confusion matrices, silhouette scores, or visualizations.

Instructor note

- 40 min teaching/demonstration
- 40 min exercises

Unsupervised Learning

In Episode 5 and Episode 6, we have explored supervised learning, where each training example includes both input features and the corresponding output. This setup enables the models to learn a direct mapping from inputs to targets. For the Penguins dataset, both classification and regression models, such as logistic/linear regression, decision trees, and neural networks, were applied to either classify penguin species or predict body mass based on flipper length.

It is important to emphasize that **supervised learning** depends heavily on labeled data, which may not always be available in real-world scenarios. Collecting labeled data can be expensive, time-consuming, or even impossible. In such cases, we turn to **unsupervised learning** to uncover patterns and structure in the data.

In unsupervised learning, the dataset contains only the input features without associated labels. The goal is to discover hidden patterns, structures within the data, and derive insights without explicit supervision. Unsupervised learning is essential for analyzing the vast amounts of raw data generated in real-world applications, from scientific research to business intelligence. Its significance can be seen across several key areas:

- **Exploratory Data Analysis** (EDA): Techniques such as clustering and dimensionality reduction are fundamental for understanding the structure of complex datasets. They can reveal natural groupings, trends, and correlations that might otherwise remain hidden, providing a crucial first step in any data-driven investigation.

- **Anomaly Detection**: Unsupervised learning is vital for maintaining security and operational integrity. By modeling “normal” behavior, algorithms can identify unusual patterns, such as fraudulent financial transactions, network intrusions, or rare mechanical failures, without needing labeled examples of every type of anomaly.
- **Feature Engineering** and **Representation Learning**: Methods like **Principal Component Analysis** (PCA) can compress data into its most informative components, reducing noise and improving the efficiency and performance of downstream supervised models.

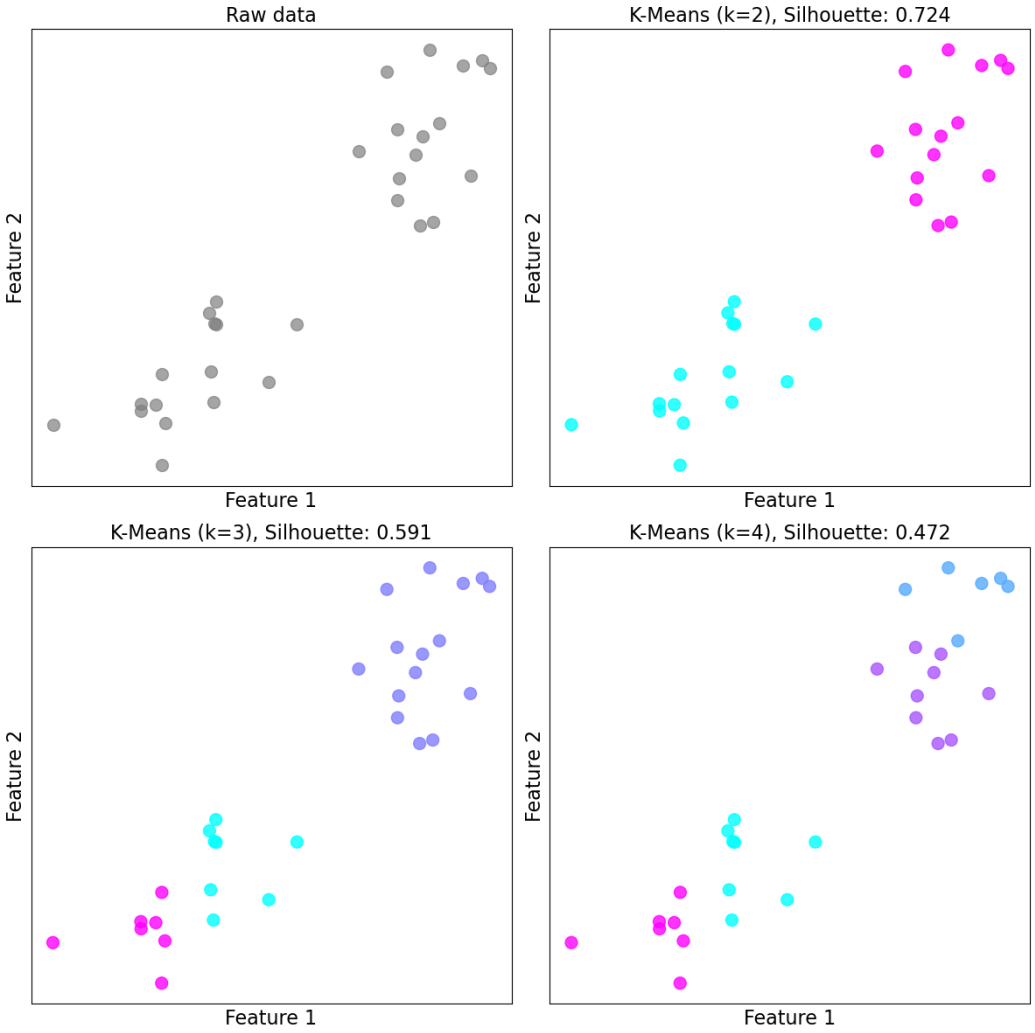
In this and the following episodes, we will apply **Clustering** and **Dimensionality Reduction** methods on the Penguins dataset to explore its underlying structure and uncover hidden patterns without the guidance of pre-existing labels. By employing clustering methods like K-means, we aim to identify species-specific clusters or other biologically meaningful subgroups among Adelie, Gentoo, and Chinstrap penguins. Additionally, dimensionality reduction techniques like PCA will simplify the dataset’s feature space, enabling visualization of complex relationships and enhancing subsequent analyses. These approaches will deepen our understanding of penguin characteristics, reveal outliers, and complement supervised methods by providing a robust framework for exploratory data analysis.

Clustering

Clustering is one of the most widely used techniques in unsupervised learning, where the goal is to group similar data points together without using predefined labels. For example, if we cluster penguins based on their physical characteristics such as flipper length, body mass, bill length, and bill depth, we may be able to separate them into natural groups that correspond to their species – even without explicitly providing species labels.

Clustering, however, presents several fundamental challenges. One major issue is determining the optimal number of clusters (k), which is often non-trivial. Many algorithms, such as k-means, require specifying the number of clusters in advance, which may not be immediately obvious from the data.

As illustrated in the figure below, the data could be grouped into two, three, or four clusters, depending on the level of granularity chosen. Selecting too few clusters may oversimplify the structure and miss important patterns, while choosing too many clusters can lead to overfitting, where random noise is mistakenly treated as meaningful groups.



In addition, the quality and scale of features also have a significant impact on clustering results. Features with larger scales can dominate distance computations, making preprocessing steps such as standardization essential.

It should be emphasized that the interpretation of clustering results is subjective. While an algorithm can identify groups, it is up to the analyst to determine whether those groups are meaningful or merely artifacts of the algorithm.

Clustering outcomes are also highly sensitive to the choice of algorithm and distance metric. For instance, K-Means tends to find spherical clusters, whereas DBSCAN (Density-Based Spatial Clustering of Applications with Noise) can detect arbitrarily shaped clusters and identify outliers, may leading to very different conclusions using the same dataset.

In this episode, we will apply multiple clustering algorithms to evaluate their performance on the Penguins dataset.

Data Preparation

Following the procedures used in previous episodes, we apply the same preprocessing steps to the Penguins dataset, including handling missing values and detecting outliers. For the clustering task, categorical features are not required, so encoding them is unnecessary.

Data Processing

The data processing is straightforward for the clustering task: we simply extract the numerical variables and apply standardization.

```
penguins = sns.load_dataset('penguins')
penguins_clustering = penguins.dropna()

X = penguins_clustering[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
'body_mass_g']]

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

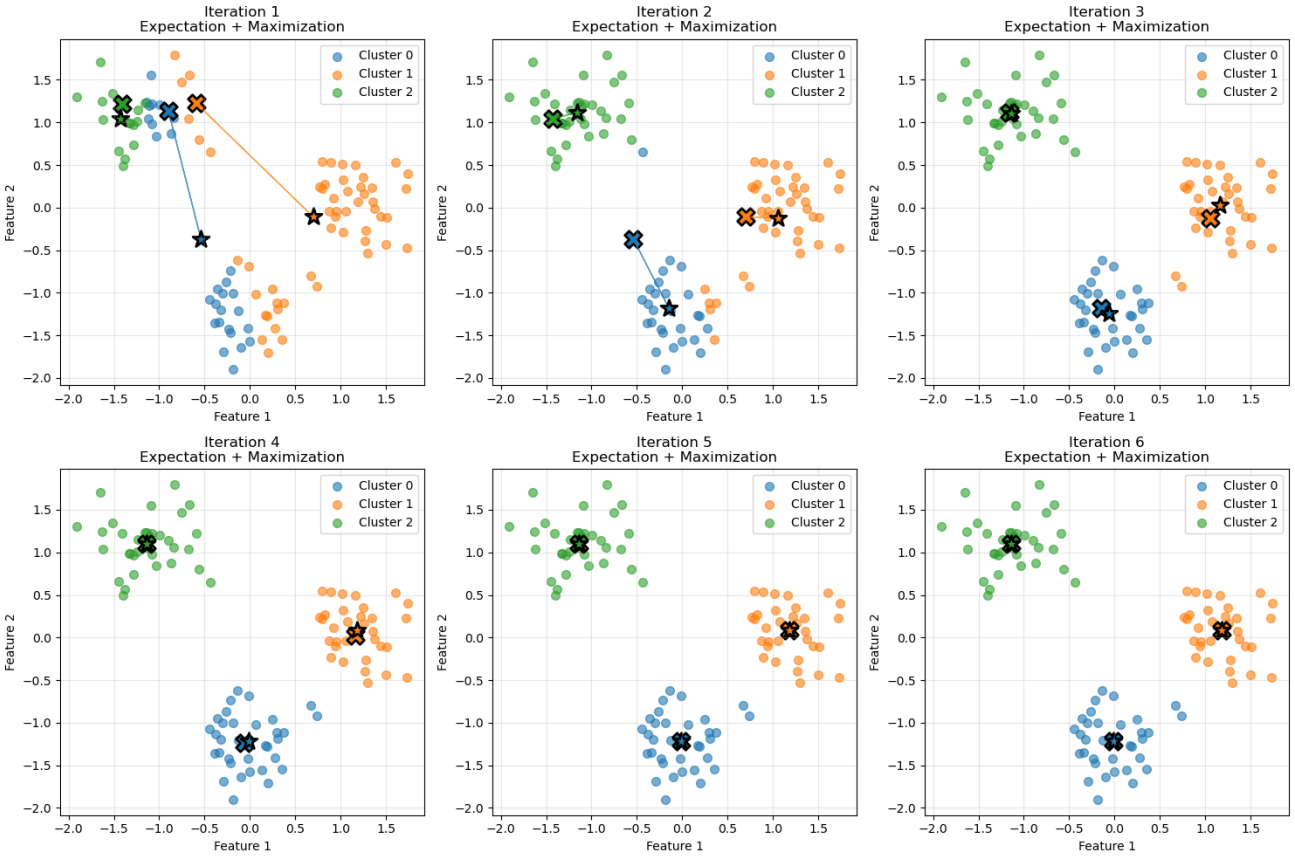
Training Model & Evaluating Model Performance

K-Means Clustering

K-Means clustering, a centroid-based partitioning method, is a widely used unsupervised learning algorithm that divides data into k distinct, non-overlapping clusters. K-Means operates on a simple yet powerful principle: each cluster is represented by its centroid, which is the mean position of all points within the cluster.

The algorithm begins by randomly initializing k centroids in the feature space and then iteratively refines their positions through a two-step **expectation-maximization** process:

- In the expectation step, each data point is assigned to its nearest centroid based on Euclidean distance, forming preliminary clusters.
- In the maximization step, the centroids are recalculated as the mean of all points assigned to each cluster. This process repeats until convergence, typically when centroid positions stabilize or cluster assignments no longer change significantly.



align: center width: 100%

We build a `kmeans` model using the `KMeans` class from `sklearn.cluster` with specified parameters. By fitting the constructed `kmeans` model, we can obtain the cluster ID to which each point belongs.

- `n_clusters=k`, number of clusters to find from the dataset
- `n_init=10`, number of times KMeans runs with different centroid seeds (default 10 or more)
- `random_state`, ensures reproducibility

```
from sklearn.cluster import KMeans

k = 3  # we know there are 3 species
kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
clusters = kmeans.fit_predict(X_scaled)
```

Evaluating clustering efficiency can be tricky because, unlike supervised learning, we don't always have "true labels". Depending on whether we have ground truth or not, there are two main evaluation approaches:

- If we don't know the actual labels, we can measure how well each data point fits within its assigned cluster compared to other clusters, such as the **Silhouette Score**.
- If we know the actual labels (*e.g.*, penguin species), we can measure how well clustering recovers them, such as the **Adjusted Rand Index** (ARI).

Here we adopte these two matrics to evaluate model performance.

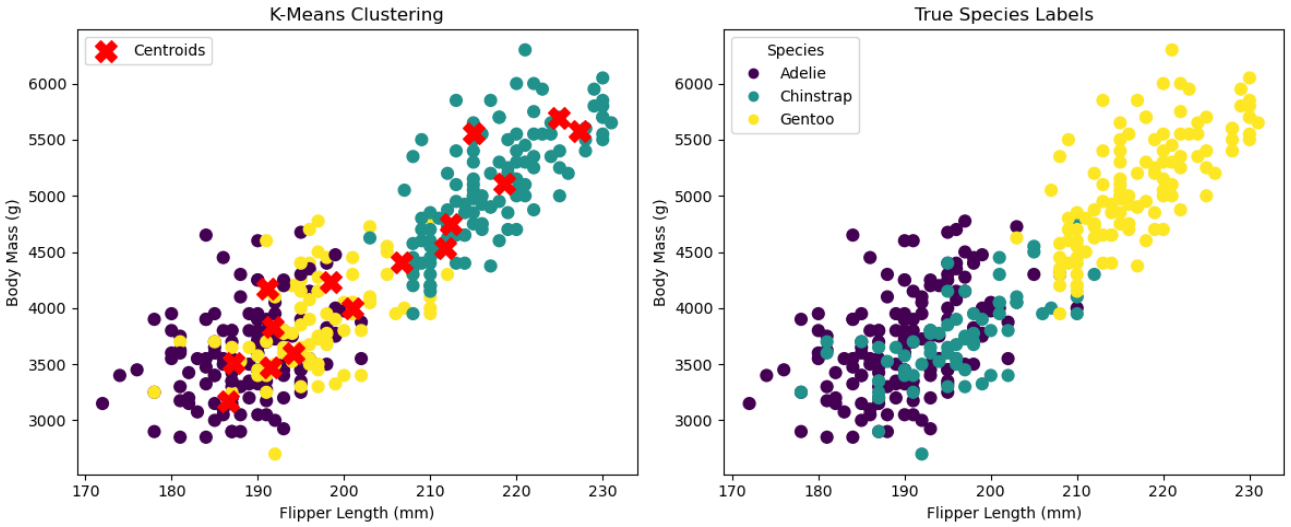
```
penguins_cluster = penguins.dropna().copy()
penguins_cluster['cluster'] = clusters

from sklearn.metrics import silhouette_score, adjusted_rand_score

sil_score = silhouette_score(X_scaled, clusters)
ari_score = adjusted_rand_score(penguins_cluster['species'], clusters)

print(f"Silhouette Score: {sil_score:.3f}")
print(f"Adjusted Rand Index (vs true species): {ari_score:.3f}")
```

Higher ARI values indicate that the clustering results align closely with the true groupings. An ARI of 1.0 represents perfect agreement, and of 0 corresponds to random clustering. Negative ARI values suggest clustering performance worse than random chance.

The Silhouette Score ranges between -1 to +1.

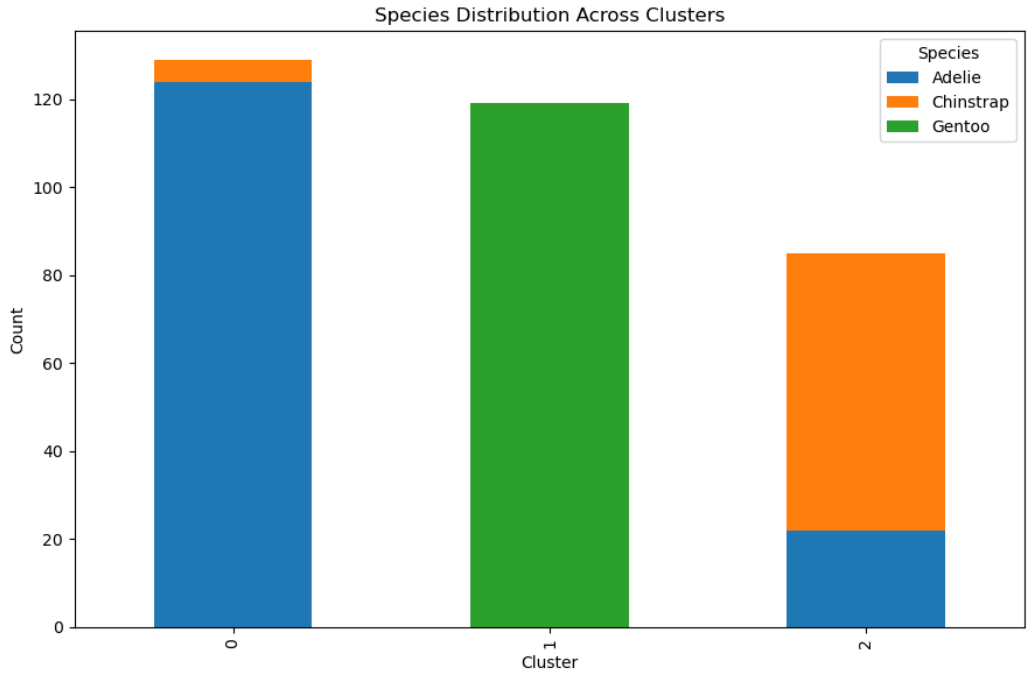
- A score of +1 indicates that samples in the dataset are well-matched to their own cluster and clearly separated from other clusters.
- A score of 0 suggests that samples lie on the boundary between clusters.
- A score of -1 implies that samples may have been incorrectly assigned to clusters.

We take a further step to visualize the distributions of penguins by comparing their true labels with the clusters determined by the `kmeans` model.



We have 333 penguins, and from the plots above it is difficult to determine how many penguins belong to each cluster and what their species are. This can be clarified by examining the distribution of penguin species across clusters and computing a cross-tabulation of two categorical variables using the `.crosstab()` method in Pandas.

```
cross_tab = pd.crosstab(penguins_cluster['cluster'], penguins_cluster['species'])

cross_tab.plot(kind='bar', stacked=True, figsize=(9, 6))
```



Determination of optimal number of clusters

At the beginning of this section, we set `n_clusters = 3`, because we already knew that the Penguins dataset contains three species. However, in many real-world applications, the true number of groups or clusters is not known in advance. In such cases, it becomes essential to estimate the appropriate number of clusters before performing the actual clustering task.

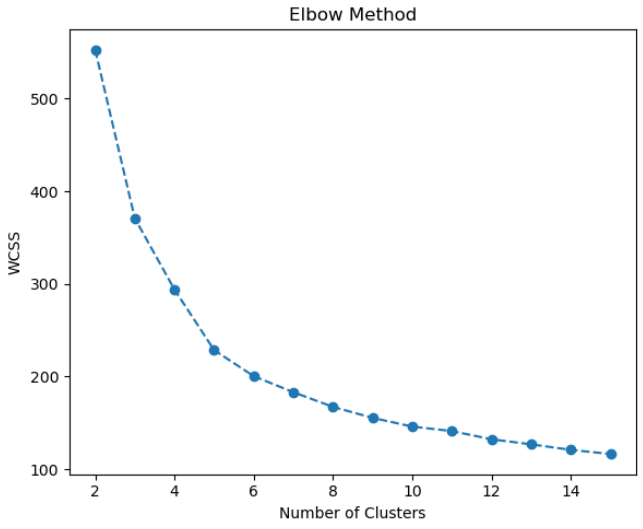
To address this, we employ two widely-used heuristic methods, the **Elbow Method** and the **Silhouette Score analysis**, to determine the optimal cluster number *k*.

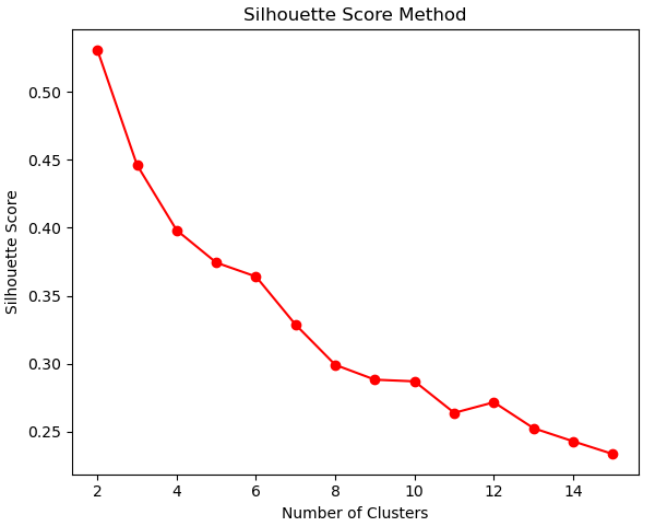
- The Elbow Method quantifies the quality of the clustering using **Within-Cluster Sum of Squares** (WCSS), which measures how tightly the data points in each cluster are grouped around their centroid.
  - Intuitively, we want clusters that are tight and cohesive, which corresponds to a low WCSS.
  - As we increase the number of clusters *k*, the WCSS will always decrease because the clusters become smaller and tighter. Beyond a certain point, the improvement of k becomes marginal contribution to WCSS.
  - By plotting the WCSS against the number of clusters, we look for the **elbow** point in the curve, which represents a good balance between model complexity and cluster compactness.
- The Silhouette Score Method evaluates the quality of clustering by measuring how similar each data point is to its own cluster compared to other clusters.

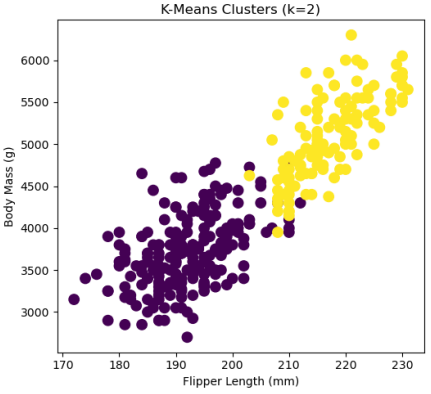
- For a single data point, the silhouette coefficient compares the average distance to points in its own cluster (cohesion) to the average distance to points in the nearest neighboring cluster (separation).

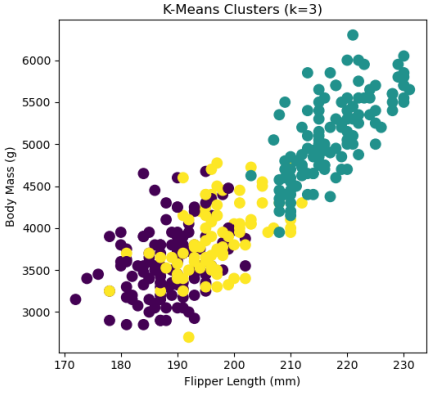
We rerun the computation using the K-Means algorithm across a range of cluster values. For each tested number of clusters, we compute both the WCSS and the Silhouette Score. By plotting these metrics against the number of clusters *k*, we can visually assess the trade-offs and identify the most suitable cluster count. The code example and corresponding output are shown below.

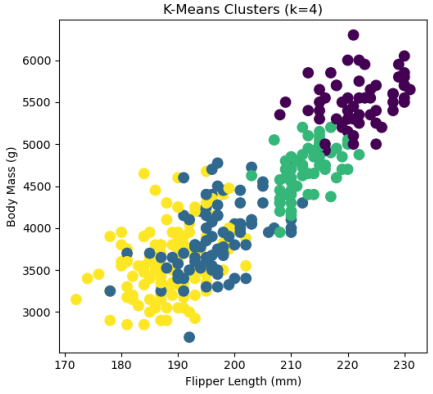
```
max_clusters = 15
wcss = []
silhouette_scores = []

for i in range(2, max_clusters + 1):
    kmeans = KMeans(n_clusters=i, random_state=42, n_init=10)
    kmeans.fit(X_scaled)

    wcss.append(kmeans.inertia_)

    silhouette_avg = silhouette_score(X_scaled, kmeans.labels_)
    silhouette_scores.append(silhouette_avg)

    print(f"Clusters: {i}, WCSS: {kmeans.inertia_:.2f}, Silhouette:
{silhouette_avg:.3f}")
```











💬 Why does K-Means suggest grouping the penguins into 2 clusters?

K-Means suggesting 2 clusters instead of 3 in the Penguins dataset is actually a common outcome, and it happens for several reasons:

- feature overlap between species:
  - Some penguin species, like Adelie and Chinstrap, have very similar measurements for features such as bill length, bill depth, flipper length, and body mass.
  - K-Means uses Euclidean distance, so if two species' points are close in feature space, the algorithm may group them into a single cluster.
- data scaling or feature selection:
  - Features with larger scales or high correlation can dominate the distance calculation.
  - If preprocessing is not optimal, K-Means may prioritize grouping based on dominant features rather than species distinctions.
- K-Means assumes spherical clusters:
  - K-Means works best when clusters are roughly spherical and equally sized.
  - If clusters have different shapes, densities, or overlap, K-Means may merge two clusters to minimize WCSS, resulting in fewer clusters than the actual number of species.
- Elbow or Silhouette methods suggest 2:
  - When using the elbow method, the WCSS curve may show a clear "elbow" at k=2, indicating that adding a third cluster doesn't significantly reduce WCSS.
  - Similarly, the average Silhouette Score might be highest for k=2, because splitting the overlapping species into separate clusters reduces cohesion.

Hierarchical Clustering

Hierarchical clustering is an unsupervised learning method that builds a hierarchy of clusters by either divisive (top-down) or agglomerative (bottom-up) strategies. In the agglomerative approach, each data point starts as its own cluster, and the algorithm iteratively merges the closest clusters based on a distance metric. This continues until all points are merged into a single cluster. The result can be visualized as a **dendrogram**, a tree-like diagram showing the nested structure of clusters at different levels of granularity.

❗ Hierarchical Clustering *vs.* Decision Tree

Hierarchical clustering is conceptually similar to a decision tree in some ways, but it is not the same as a decision tree or random forest.

- Similarity is that both build a tree-like structure
- Key differences
  - Purpose of the tree
    - In hierarchical clustering, the tree (dendrogram) represents nested clusters and shows the order in which points/clusters are merged or split.
    - In decision trees, the tree represents decision rules to predict a target variable.

- Supervised *vs.* unsupervised algorithms
- With and without ensemble concept
  - Random forest is an ensemble of decision trees and focuses on improving prediction accuracy and reducing overfitting.
  - Hierarchical clustering has no ensemble concept or predictive objective; it is purely descriptive.

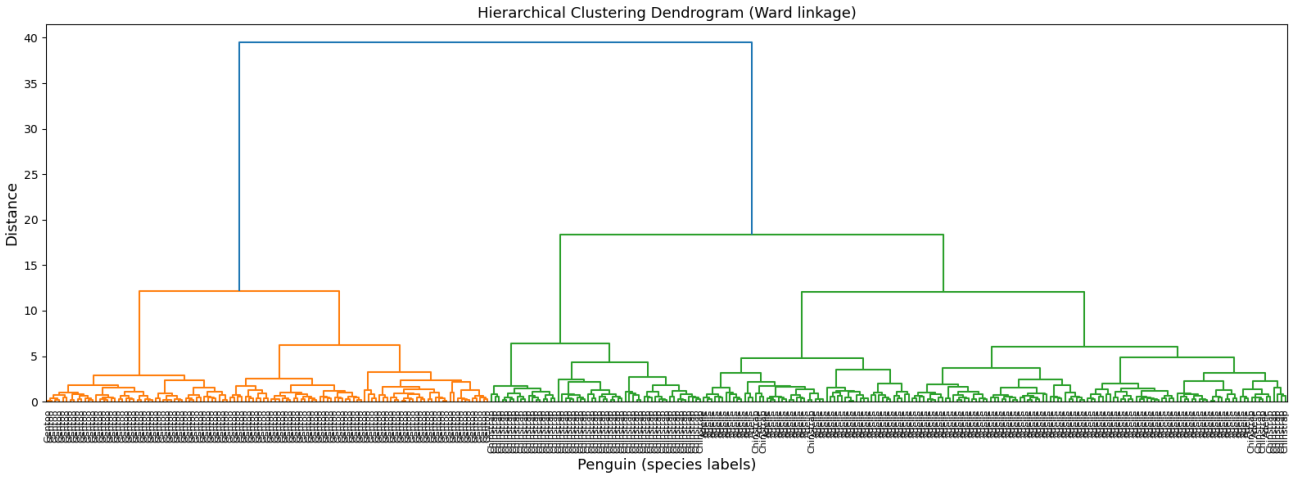
In short, **Hierarchical Clustering is structurally similar to a tree (like a dendrogram)**, but **it is unsupervised and descriptive, unlike Decision Trees or Random Forests, which are supervised and predictive**.

We first use SciPy and then Scikit-Learn packages for the clustering task, for the purpose of comparison. In SciPy, hierarchical clustering involves two steps: **computing the linkage matrix** (`linkage()`), and then **extracting clusters from it** (`fcluster()`). In the code listed below,

- `linkage()` computes the full hierarchical clustering tree (linkage matrix), storing all merges, distances, and cluster sizes.
- `fcluster()`, cuts the dendrogram at a specified threshold to produce a flat clustering, here forming 3 clusters (`t=3` , criterion='maxclust').

```
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster

# compute linkage matrix
linked = linkage(X_scaled, method='ward')

# assign 3 clusters based on dendrogram cut
labels_scipy = fcluster(linked, t=3, criterion='maxclust')

penguins_cluster = penguins.dropna().copy()
penguins_cluster['hier_cluster_scipy'] = labels_scipy
```

Next we plot the dendrogram to visualize clustering structure.

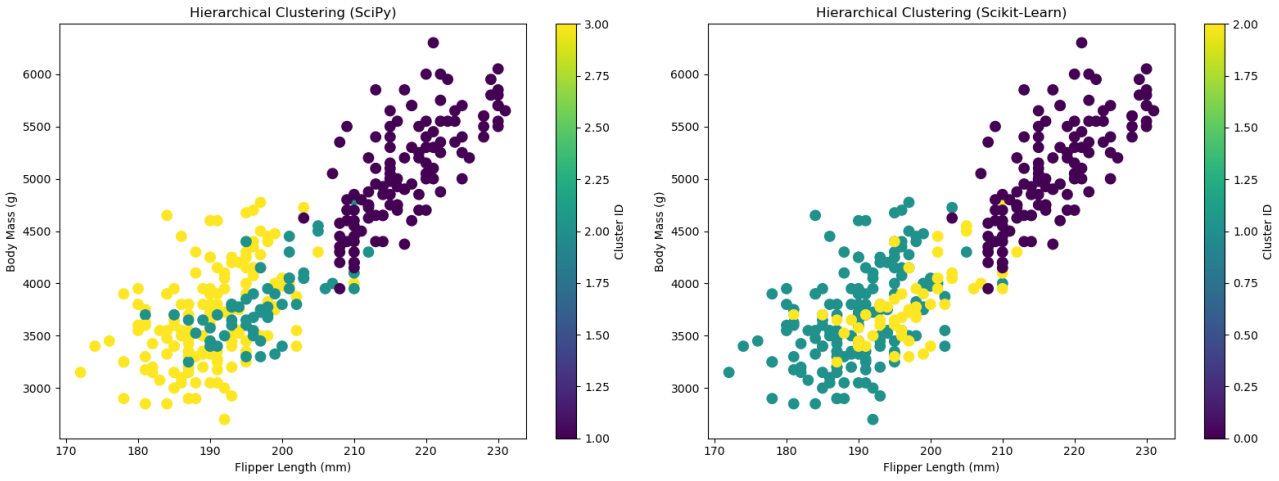


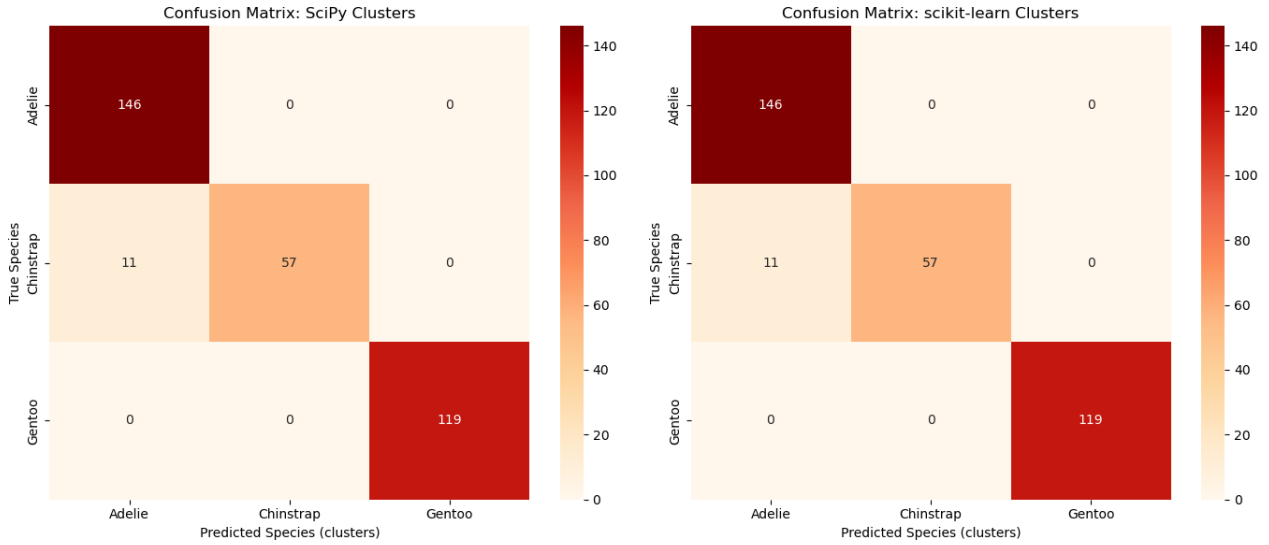
Here, we move to the Scikit-learn package and employ `AgglomerativeClustering` to construct the clustering model with hyperparameters.

- The parameter `linkage` determines how the distance between clusters is calculated. There are several options for this parameters:
  - `ward`, minimizes the variance of merged clusters (only works with Euclidean distance).
  - `complete`, maximum distance between points in clusters.
  - `average`, average distance between points in clusters.
  - `single`, minimum distance between points in clusters.
- The parameter `metric` is the distance metric used to compute the distance between points.
  - `euclidean` is the standard straight-line distance in feature space.
  - There are also other options like `manhattan`, `cosine`, *etc.*

```
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(n_clusters=3, metric='euclidean', linkage='ward')
labels = hc.fit_predict(X_scaled)

penguins_cluster['hier_cluster_aggl'] = labels
```

We can examine the number of penguins in each species within the clusters determined by the two models, and visualize their distributions using a confusion matrix.





DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together points that are closely packed while marking points in low-density regions as outliers. Unlike K-Means, DBSCAN does not require specifying the number of clusters in advance, making it particularly useful when the number of natural clusters is unknown. It is also capable of identifying clusters of arbitrary shapes, unlike centroid-based methods that assume roughly spherical clusters. This makes DBSCAN robust to clusters with irregular shapes or varying sizes.

We build a model using the `DBSCAN` class from `sklearn.cluster` with specified parameters. DBSCAN relies on two key parameters:

- `eps`, the radius that defines the neighborhood around a point.
- `min_samples`, the minimum number of points required within a point's eps neighborhood for it to be considered a core point.
- Below we use `eps=0.55` and `min_samples=5` in the code.
  - You can experiment with other `eps` values (*e.g.*, 0.50 and 0.80) while keeping `min_samples=5` to observe how the clustering results change.

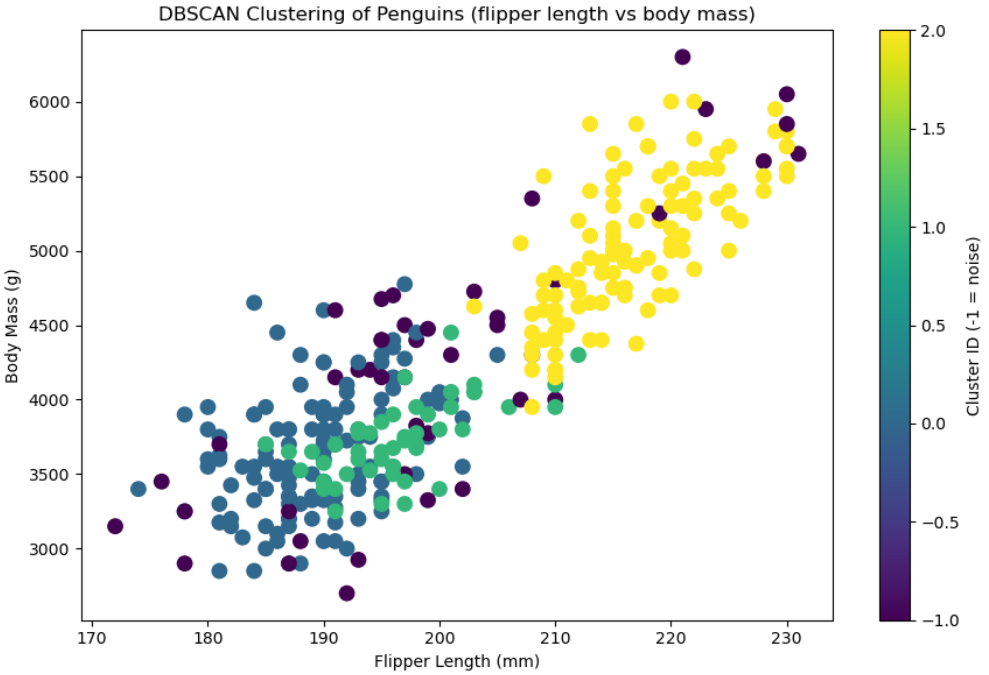
```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.55, min_samples=5)
labels = dbscan.fit_predict(X_scaled)

# evaluate clustering (only if at least 2 clusters found)
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
print(f"DBSCAN found {n_clusters} clusters (and {sum(labels==-1)} noise points).")

if n_clusters > 1:
    sil_score = silhouette_score(X_scaled, labels)  # FIXED
    ari_score = adjusted_rand_score(penguins_cluster['species'], labels)
    print(f"Silhouette Score: {sil_score:.3f}")
    print(f"Adjusted Rand Index (vs true species): {ari_score:.3f}")
```

Note

DBSCAN classifies samples into three types:

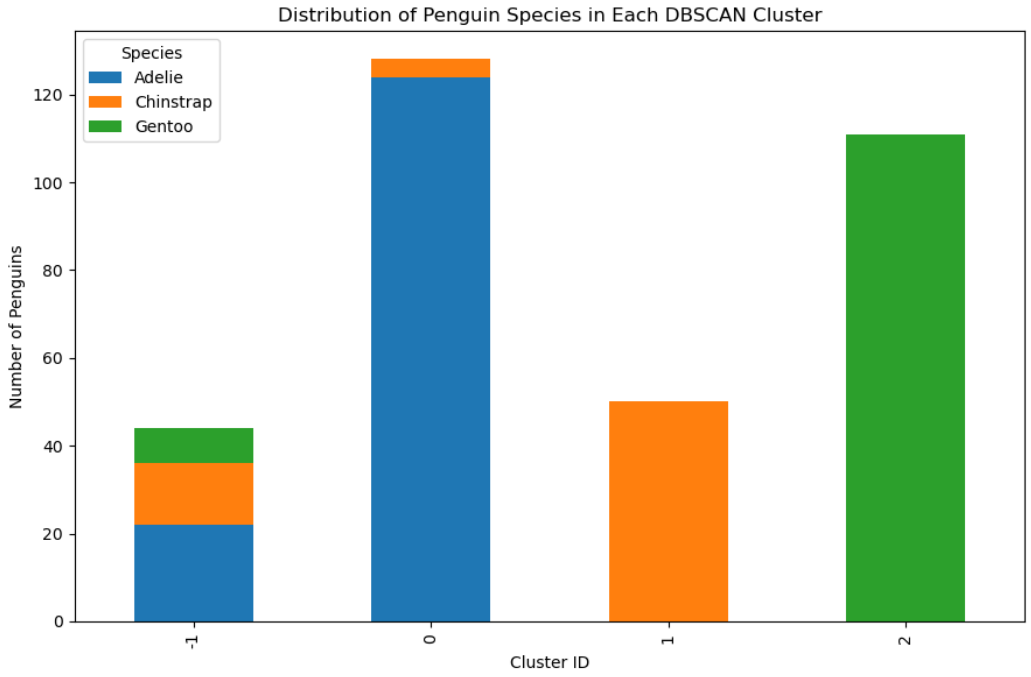
- **core points**: points with at least `min_samples` neighbors within `eps`.
- **border points**: points within `eps` of a core point but with fewer than `min_samples` neighbors themselves.
- **noise points** (outliers): points that are neither core nor border points.

Next, we visualize the distributions of penguins in each cluster, including any points identified as noise.



We further examine the distribution of penguin species across clusters using the cross-tabulation (`.crosstab()`) method in Pandas.

```
cross_tab = pd.crosstab(penguins_cluster['dbscan_cluster'],
penguins_cluster['species'])

cross_tab.plot(kind='bar', stacked=True, figsize=(9,6))
```



✍ Exercise

In this exercise (code examples are availalbe in the Jupyter Notebook, we will

- Experiment with `eps` values (0.50 and 0.80) while keeping `min_samples=5` to observe how the clustering results change.
- Computations with more combinations of `eps` and `min_samples`.
- Explore methods to find optimal hyperparameters (using grid search and cross-validation).

Gaussian Mixture Models

After exploring centroid-based methods like K-Means, hierarchical clustering models, and density-based approaches such as DBSCAN, we now turn our attention to model-based clustering algorithms. Unlike the previous methods that rely primarily on distance metrics or density thresholds, model-based clustering assumes that the data come from a mixture of underlying probability distributions. Each distribution corresponds to a cluster, and the algorithm tries to estimate both the parameters of the distributions and the clusters.

Since model-based clustering assumes that a dataset is generated from a mixture of underlying probability distributions, a variety of algorithms have been developed to handle different types of distributions. The choice of model depends on the nature of the data.

- When dealing with continuous numerical features that approximately follow a bell-shaped distribution, **Gaussian Mixture Models** (GMMs) are the most common choice.
- If the data consists of count values, mixture models based on Poisson distributions can be used.
- For categorical data, methods like Latent Class Analysis (LCA), which treats clusters as latent categorical variables, are often applied.
- In more flexible Bayesian frameworks, Dirichlet Process Mixture Models allow the number of clusters to be inferred directly from the data, avoiding the need to predefine it.

The GMM assumes that data points are generated from a mixture of Gaussian distributions, each representing a cluster. Instead of assigning points strictly to one cluster (like K-Means), GMM assigns each point a probability of belonging to each cluster, making it a soft clustering method.

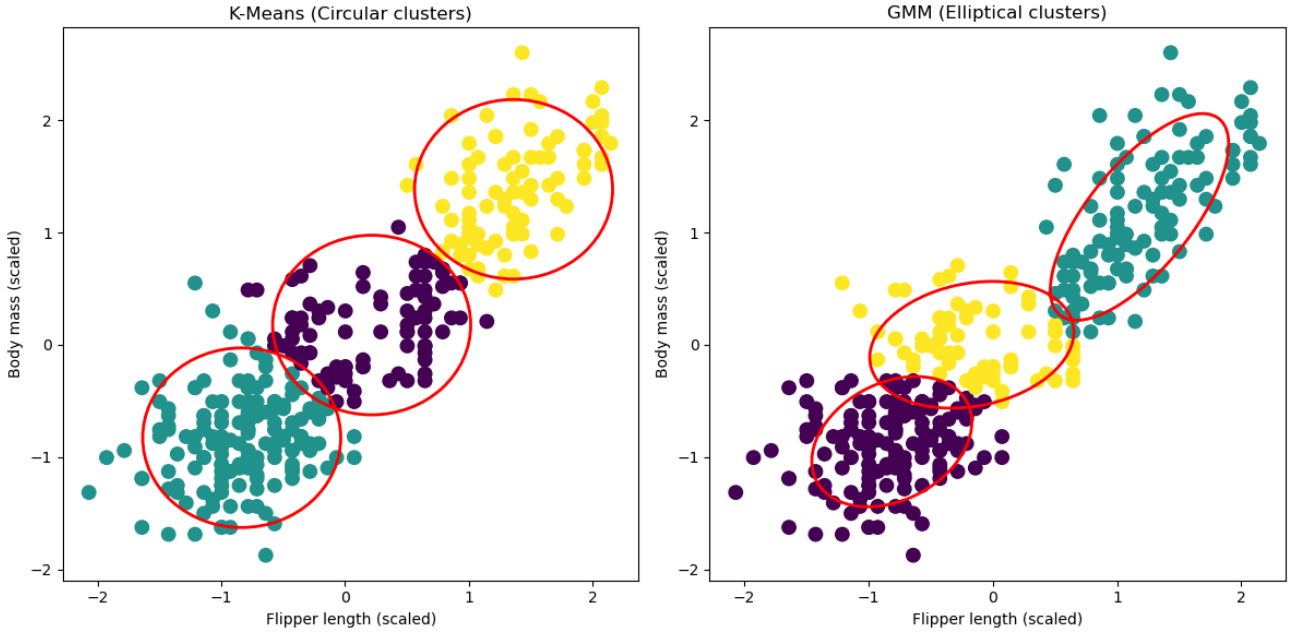
In the following example, we construct the GMM model with the specified hyperparameters.

- `n_components=3` means the number of Gaussian distributions (clusters) to fit
- `covariance_type` controls the form of the covariance matrix for each Gaussian distribution
  - `full` means that each cluster has its own general covariance matrix (most flexible, allows ellipsoidal shapes).
  - other options like `tied`, `diag`, and `spherical` corresponding to clusters with different shapes

```
from sklearn.mixture import GaussianMixture
from sklearn.metrics import adjusted_rand_score, silhouette_score

# build GMM model with 3 components (clusters)
gmm = GaussianMixture(n_components=3, covariance_type='full', random_state=42)
gmm.fit(X_scaled)
labels_gmm = gmm.predict(X_scaled)
```

Following the steps in the Jupyter Notebook, we can 1) examine the distribution of penguin species across clusters using the cross-tabulation method, 2) visualize the distribution of penguins within each cluster, and 3) illustrate their distributions with a confusion matrix.

Here, we specifically highlight the distributions penguins data points in clusters and the shapes clusters obtained from the KMeans and GMM models.



Spectral Clustering

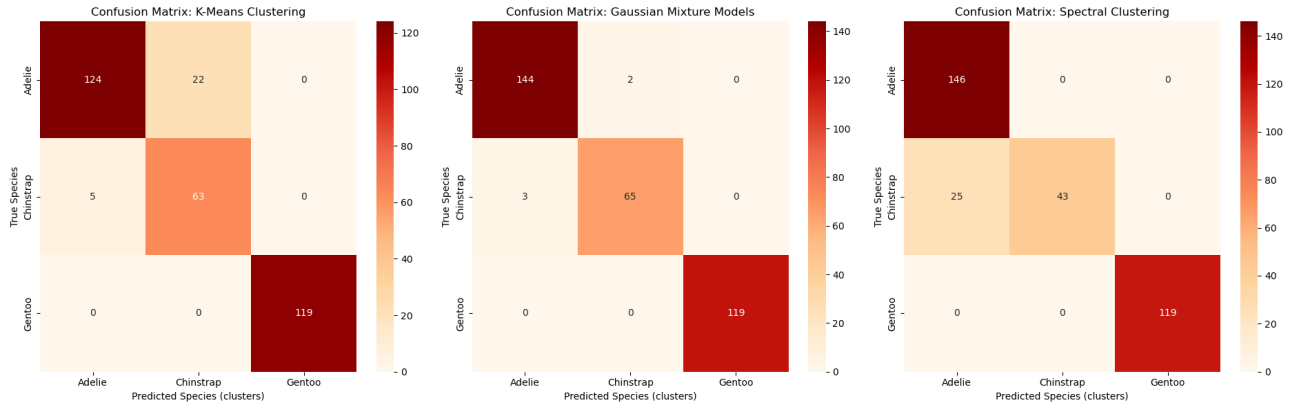
Following centroid-based, density-based, and model-based methods, we now turn our attention to **Spectral Clustering** algorithms.

Spectral Clustering represents a fundamentally different approach: rather than relying purely on distances between points or density, it leverages graph theory and the eigenstructure of similarity matrices to uncover clusters. That is, the main idea of this method is to represent the dataset as a graph where each node is a data point and edges encode the similarity between points (*e.g.*, using a Gaussian kernel). By computing the eigenvectors of the graph Laplacian, the algorithm transforms the original data into a lower-dimensional space where clusters become more distinguishable. Once in this space, standard clustering techniques, such as K-Means, are applied to assign cluster labels.

This method is especially powerful for datasets with complex, **non-convex** cluster shapes, where traditional algorithms like K-Means or Hierarchical Clustering may fail to capture the true underlying structure.

We adopted similar procedures to build the model, examine the distribution of penguin species across clusters using the cross-tabulation method, and visualize the distribution of penguins within each cluster.

```
from sklearn.cluster import SpectralClustering

# build model using Spectral Clustering (graph-based)
spectral = SpectralClustering(
    n_clusters=3,
    affinity='rbf',    # Gaussian kernel
    gamma=1.0,         # controls width of the Gaussian
    assign_labels='kmeans',
    random_state=42
)
labels = spectral.fit_predict(X_scaled)
penguins_cluster['spectral_cluster'] = labels

# evaluate clustering
ari = adjusted_rand_score(penguins_cluster['species'], labels)
sil = silhouette_score(X_scaled, labels)
print(f"Adjusted Rand Index (vs species): {ari:.3f}")
print(f"Silhouette Score: {sil:.3f}")
```



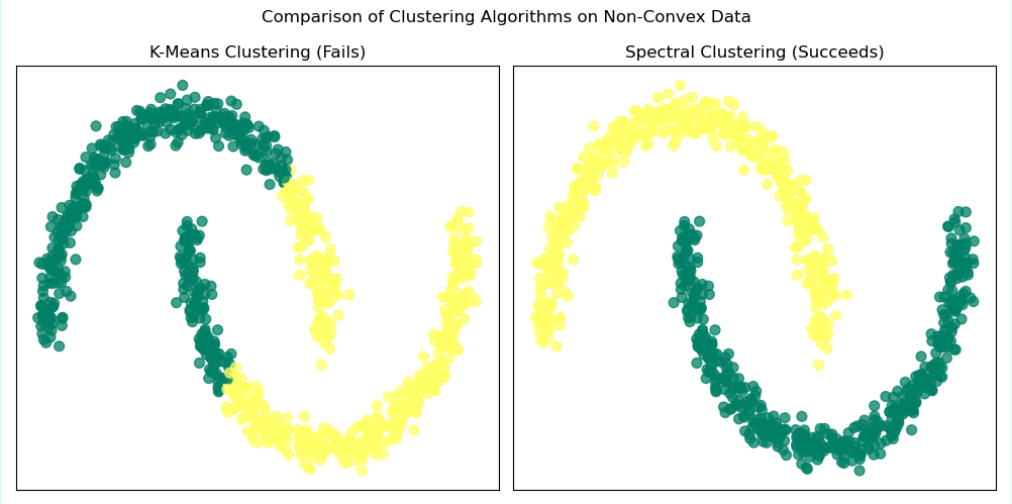
Spectral Clustering *vs.* Gaussian Mixture Models *vs.* K-Means

From the confusion matrix shown above, it seems that Spectral Clustering and K-Means models are less effective than the GMM model on the Penguins dataset. Main reasons may attribute to:

- Small dataset size:
  - The Penguins dataset has only 333 samples.
  - Spectral clustering computes the eigenvectors of the similarity matrix, which can be less stable with small datasets, leading to variability in cluster assignments.
- Small number of features/low dimensionality
  - The Penguins dataset typically uses only 4 numerical features. In such low-dimensional, fairly well-separated data, simpler methods like K-Means or Gaussian Mixture Models often perform just as well or better.
  - Spectral clustering shines when clusters are non-convex or complexly shaped in high-dimensional spaces.

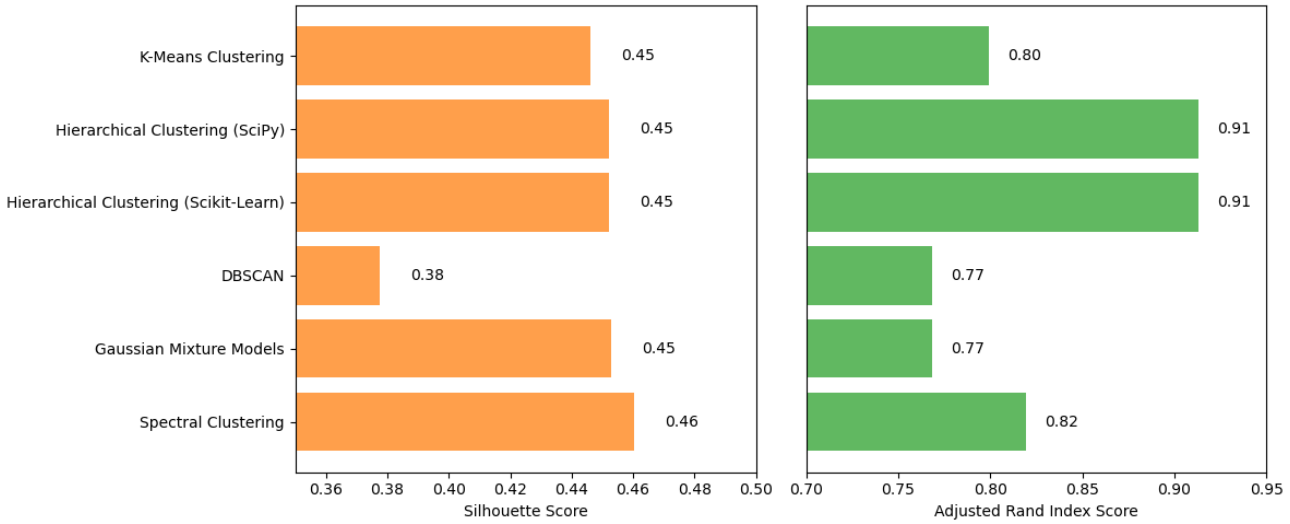
✍️ Exercise

Here, we apply these two models to the classic **two-moon dataset**, a well-known synthetic dataset with non-linearly separable clusters. This allows us to visually and quantitatively evaluate how each algorithm performs in capturing complex, non-convex structures and to compare their strengths and limitations in a controlled setting.



Spectral Clustering excels for datasets with complex, non-convex cluster shapes, where traditional algorithms like K-Meansmay fail to capture the true underlying structure.

Comparison of Clustering Models



Method	Type/Approach	Key Characteristics	Pros	Limitations
K-Means	Centroid-based	Partitions data into k clusters by minimizing within-cluster variance; clusters represented by centroids	Simple, fast, widely used; interpretable	Assumes spherical clusters; sensitive to initialization and outliers; requires specifying k
Hierarchical Clustering (SciPy)		Similar to Scikit-Learn, uses linkage matrix and fcluster to assign clusters	Flexible; supports different distance metrics and linkage methods	Requires careful selection of threshold to cut dendrogram; can be slow for large data
Hierarchical Clustering (Scikit-Learn)		Builds a hierarchy of clusters either bottom-up (agglomerative) or top-down (divisive); linkage criteria define merge/split decisions	Dendrogram visualization; no need to pre-specify number of clusters	Computationally expensive for large datasets; choice of linkage affects results
DBSCAN	Density-based	Groups points based on density; identifies core, border, and noise points; no need to specify number of clusters	Detects arbitrarily shaped clusters; robust to outliers; identifies noise	Sensitive to eps and min_samples; struggles with varying densities

Method	Type/Approach	Key Characteristics	Pros	Limitations
GMM	Model-based	Assumes data generated from a mixture of Gaussian distributions; each cluster has mean, covariance, and weight	Can model elliptical clusters; provides probabilities for cluster membership	Sensitive to initialization; may converge to local optima; assumes Gaussian distribution
Spectral Clustering	Graph-based	Uses graph Laplacian of similarity matrix; clusters derived from eigenvectors; can handle non-convex shapes	Captures complex structures; good for connected or non-spherical clusters	Computationally expensive for large datasets; sensitive to affinity and connectivity; may fail on disconnected graphs

Keypoints

- Clustering is about grouping data points that are similar to each other, without using labels.
- Representative algorithms for clustering tasks: K-Means (centroid-based), Hierarchical, DBSCAN (density-based), Gaussian Mixture Models (model-based), and Spectral Clustering (graph-based).
- Use tools like Silhouette score or visual plots to see how well clusters are separated.
- **No single algorithm is "best", and the right method depends on your data: size, type, shape, and what you want to achieve**.

Unsupervised Learning (II): Dimensionality Reduction

Objectives

- Understand the motivation for dimensionality reduction.
- Explain key dimensionality reduction techniques (PCA, t-SNE, UMAP).
- Describe the idea of PCA (linear method that keeps most variance) and apply PCA to correlate features with components.
- Interpret explained variance ratio and decide how many components to keep.
- Use PCA loadings and correlation circles to see how features contribute.
- Recognize t-SNE and UMAP as nonlinear methods for complex data visualization.
- Connect dimensionality reduction with clustering.

Instructor note

- 40 min teaching/demonstration
- 40 min exercises

From Clustering to Dimensionality Reduction: Simplifying Complexity

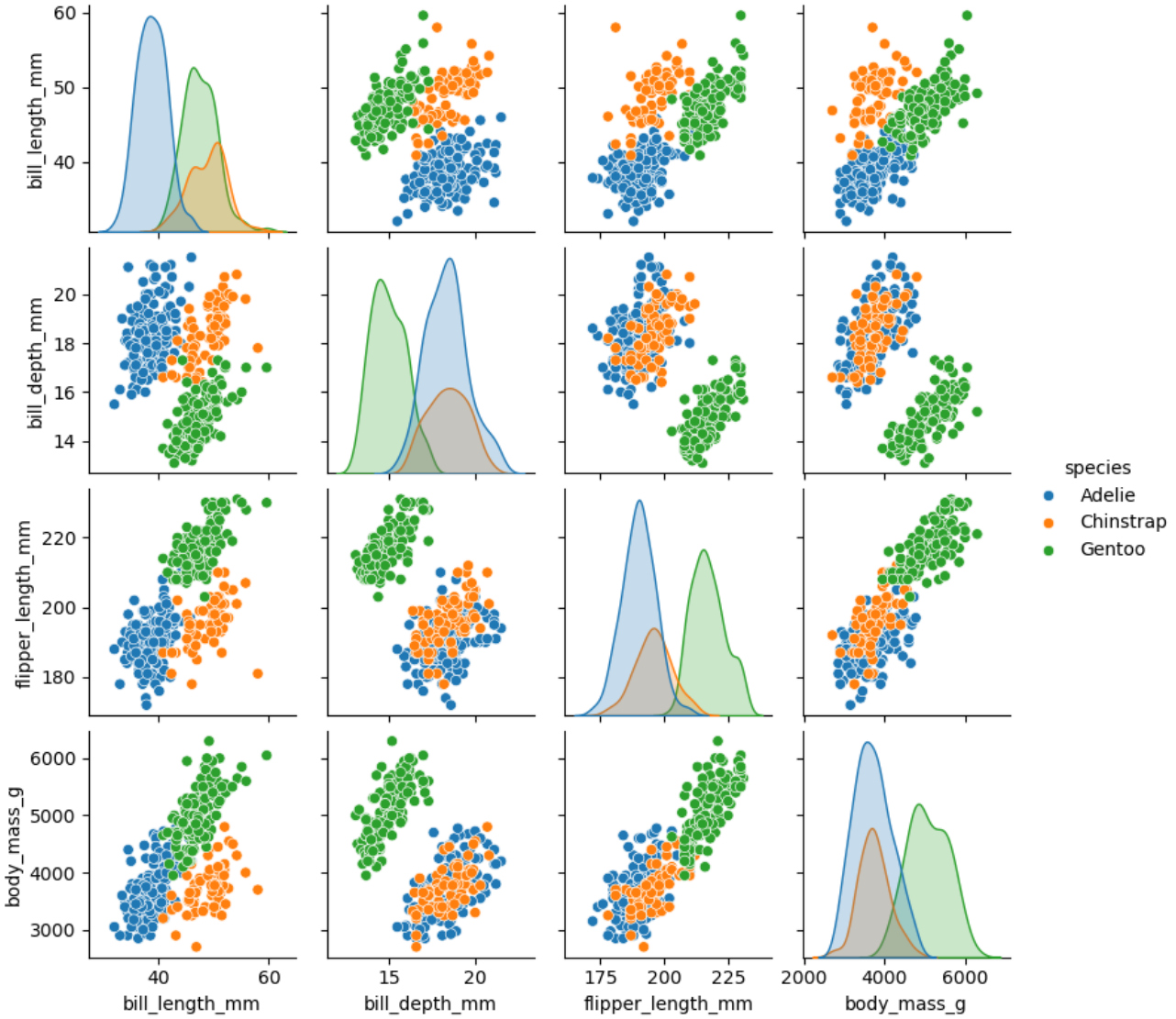
In the last episode, we talked about clustering, which is a core unsupervised ML technique that groups similar data points into clusters based on their features, without requiring labeled data. The fundamental value of clustering lies in its ability to reveal segments and patterns that are not immediately obvious, with applications ranging from customer segmentation in marketing to anomaly detection in network security.

Despite its usefulness, clustering comes with notable limitations. A major challenge is determining the appropriate number of clusters in advance, as in K-Means, where results can vary depending on initialization. Clustering results are also highly sensitive to the choice of distance metric and scaling of features, which can significantly alter outcomes. Furthermore, clustering often struggles with high-dimensional data due to the "curse of dimensionality", where distance measures lose their discriminative power, making it harder to identify meaningful groups. Given these challenges, especially around data sensitivity and interpretability, it is often crucial to preprocess data with another form of unsupervised learning before clustering: **Dimensionality Reduction**.

Where clustering seeks to group samples, dimensionality reduction focuses on simplifying the feature space itself. By transforming a high-dimensional dataset into a lower-dimensional subspace while preserving its most critical relationships, dimension reduction can mitigate noise, reduce computational cost, and reveal the most discriminative features that define the data's structure. This process not only addresses clustering's sensitivity to irrelevant features but also provides a powerful foundation for visualizing potential clusters in two or three dimensions, making the entire analytical process more robust and insightful.

Methods such as **PCA** (Principal Component Analysis), **t-SNE** (t-Distributed Stochastic Neighbor Embedding), and **UMAP** (Uniform Manifold Approximation and Projection) make data more manageable, reduce noise, and improve clustering performance.

Since the Penguins dataset includes multiple features (bill length, bill depth, flipper length, body mass, species labels, *etc.*), plotting it directly in two dimensions can make it difficult to capture all the relationships and structures hidden in the dataset. That is why we parepare the pairplots between all pair of features to achieve a clear visualization of their correlations.



In this episode, we will explore dimensionality reduction techniques and apply them to the Penguins dataset. Our goal is to take the dataset's multiple features, and project them into a simpler, lower-dimensional space for a better visualization. This process not only helps us better understand the data but also prepares it for downstream tasks, such as clustering or classification, by reducing noise and highlighting the most informative aspects of the dataset.

Data Preparation

Following the procedures used in previous episodes, we apply the same preprocessing steps to the Penguins dataset, including handling missing values and detecting outliers. For the clustering task, categorical features are not required, so encoding them is unnecessary.

Data Processing

The data processing is straightforward for the clustering task: we simply extract the numerical variables and apply standardization.

```
penguins = sns.load_dataset('penguins')
penguins_dimR = penguins.dropna()
penguins_dimR.duplicated().value_counts()

species = penguins_dimR["species"]

from sklearn.preprocessing import StandardScaler

X = penguins_dimR[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
'body_mass_g']]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Training Model & Evaluating Model Performance

Principal Component Analysis

We start with Principal Component Analysis (PCA), a powerful statistical technique used to reduce the dimensionality of data while preserving as much variability as possible. This is achieved by transforming the original variables into a new set of uncorrelated variables called **principal components**. Each principal component is a linear combination of the original variables, and they are ordered such that the first few retain most of the variation present in the original variables.

1. A simplified view: PCA with two components

We begin with a simple subspace consisting of two principal components. That is, we will transform the Penguins dataset, which originally contains four numerical features (bill length, bill depth, flipper length, and body mass), into a reduced subdataset represented by just two composite variables. These two new variables, called principal components, form a simplified version of the Penguins dataset and can essentially capture the most significant variance in the Penguins dataset while reducing complexity. This allows us to visualize the structure of the data in a two-dimensional space and better understand the relationships among the penguins.

We build a model using the `PCA` class from `sklearn.decomposition`, specifying that the `X_scaled` data should be reduced to two principal components."

```
from sklearn.decomposition import PCA

# construct a PCA model with 2 PCs
pca_2 = PCA(n_components=2)
X_pca_2 = pca_2.fit_transform(X_scaled)
explained_var_2 = pca_2.explained_variance_ratio_

print(f'''The explained variance of PC1    is {explained_var_2[0]:.2%}
The explained variance of PC2    is {explained_var_2[1]:.2%}
The explained variance (PC1+PC2) is {explained_var_2.sum():.2%}''')
```

The term `explained variance ratio` tells us how much of the total variability in the original Penguins dataset is captured by each principal component. The first principal component has an explained variance ratio of 0.72, it means 72% of the variability in the dataset can be represented by that single component.

For each penguine in the dataset, the contributions of its four original featuers to the new components are available in the `X_pca_2` dataset.

```
X_pca_2_species = penguins_dimR.join(pd.DataFrame(X_pca_2,
        index = penguins_dimR.index,
        columns = ['PC_1', 'PC_2']))
```

💬 What does the data in two principal components represent?

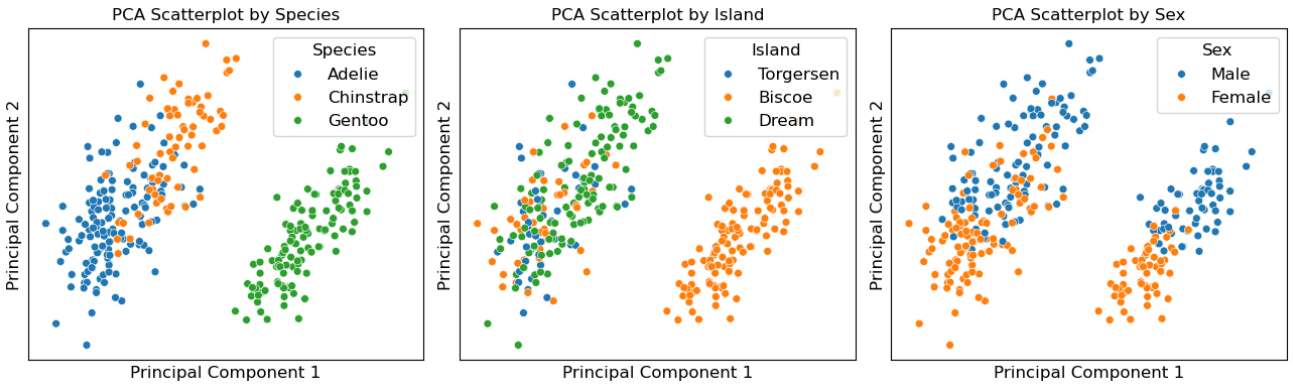
- When we have two components,
  - PC1 is the direction in feature space along which the data varies the most,
  - PC2 is the direction orthogonal to PC1 that captures the next largest variance.
- When we have three components, PC3 captures the next largest source of variance, orthogonal to both PC1 and PC2.
  - In such a situation, we have to use 3D visualization to visualize the three components.
- When we have four componnents, PC4 captures the next largest source of variance, orthogonal to both PC1, PC2, and PC3.
  - It id difficult for us to visualize the four dimensional space.

A short summary

- fewer components → lower-dimensional representation → some information is lost, but main patterns remain.
- more components → higher-dimensional representation → more information retained.

After reducing the dimensionality of the Penguins dataset to two principal components, we can now visualize the transformed data. Each penguin, originally described by four numerical features (bill length, bill depth, flipper length, and body mass), is now represented by just two composite variables.

By plotting the two components, we can examine how penguins cluster in this reduced space.



2. Preserving full variance: PCA with four components

The two-component model does a good job in capturing the main structure of the original features in the Penguins dataset, but it naturally raises an important question: **how many principal components are truly necessary to represent the dataset effectively?** Choosing the optimal number of components requires balancing simplicity against information retention — fewer components make visualization and interpretation easier, while more components preserve a greater share of the original variance. By examining metrics such as the explained variance ratio, we can make an informed choice about the number of components needed to capture the essential patterns in the Penguins dataset.

Here, we consider another case in which the new dataset has four principal components, preserving all of the original features in the Penguins dataset. By setting the parameter `n_components=4` and running the code example, we obtain a full representation of the Penguins data in the new component space. This can be verified by examining the `explained_variance_ratio`, which shows how much variance each component contributes.

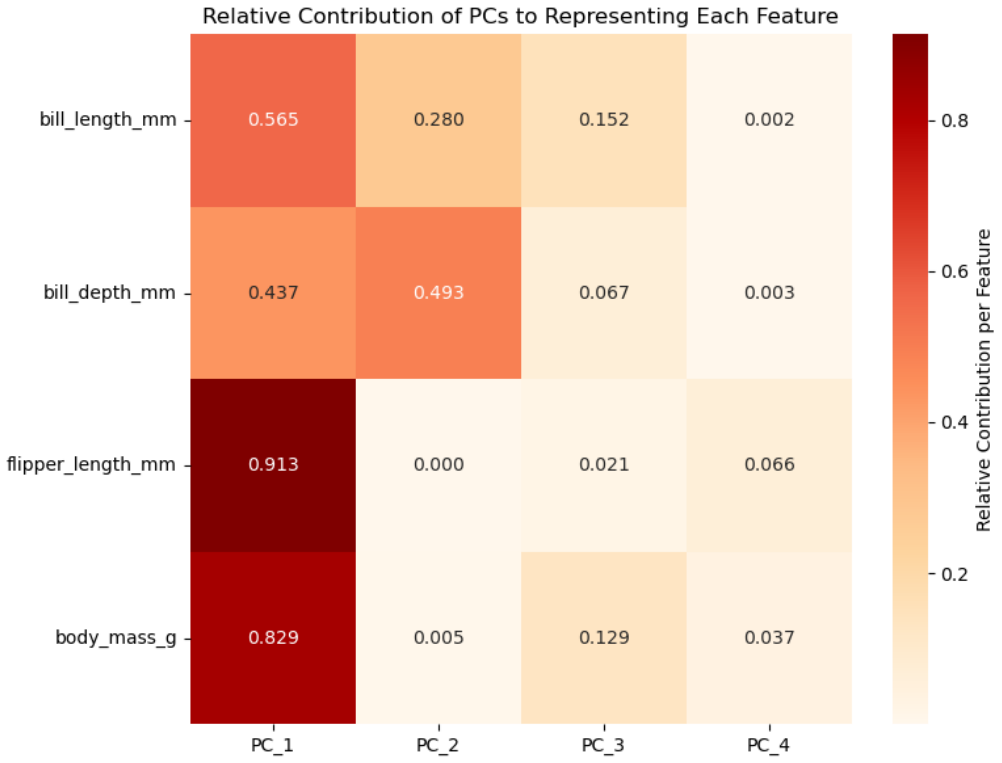
```
X_scaled_temp = pd.DataFrame(X_scaled, columns=X.columns, index=X.index)

# build a new PCA model with 4 PCs
pca_4 = PCA(n_components=4)
X_pca_4 = pca_4.fit_transform(X_scaled_temp)
explained_var_4 = pca_4.explained_variance_ratio_

print(f'''The explained variance of PC1 is {explained_var_4[0]:.2%}
The explained variance of PC2 is {explained_var_4[1]:.2%}
The explained variance of PC3 is {explained_var_4[2]:.2%}
The explained variance of PC4 is {explained_var_4[3]:.2%}
The explained variance of ALL is {explained_var_4.sum():.2%}''')
```

In PCA, each original variable (feature) is essentially correlated with each principal component (PC), which can be described by `corr_var_comp`, which qualifies how strongly each original variable contributes to a component. The `corr_var_comp` ranges from -1 to 1:

- +1 means a perfect positive correlation between the variable and the component.
- 0 indicates no linear correlation, and the component does not explain that variable at all.
- -1 suggests a perfect negative correlation.

In practical applications, the square of the correlation between a variable and a component is often denoted $\cos^2$ (cosine squared). The cosine of the angle between the variable vector and the component axis tells you how much of the variable's variance is explained by that component. Squaring it gives the proportion of variance of that variable explained by the component, hence $\cos^2$.

We first calculate `corr_var_comp` and square it to get the proportion of each variable's variance explained by the component, as shown in the figure below.



This allows us to examine and visualize in detail how each original penguine feature contributes to the four principal components. As discussed in the previous subsection, each principal component is a linear combination of the original features and can be expressed mathematically as follows:

$$PC1_i = w_1 * bill\_length_i + w_2 * bill\_depth_i + w_3 * flipper\_length_i + w_4 * body\_mass_i$$

By analyzing the PCA loadings, we can see that the first two principal components explain approximately 90% of the total variance in the Penguins dataset. This indicates that most of the important information in the original four features is already captured by these two components. Consequently, we can conclude that reducing the dataset to two principal components is sufficient for visualization and analysis, striking a balance between simplicity and information retention while effectively summarizing the underlying structure of the data.

3. Correlation circle

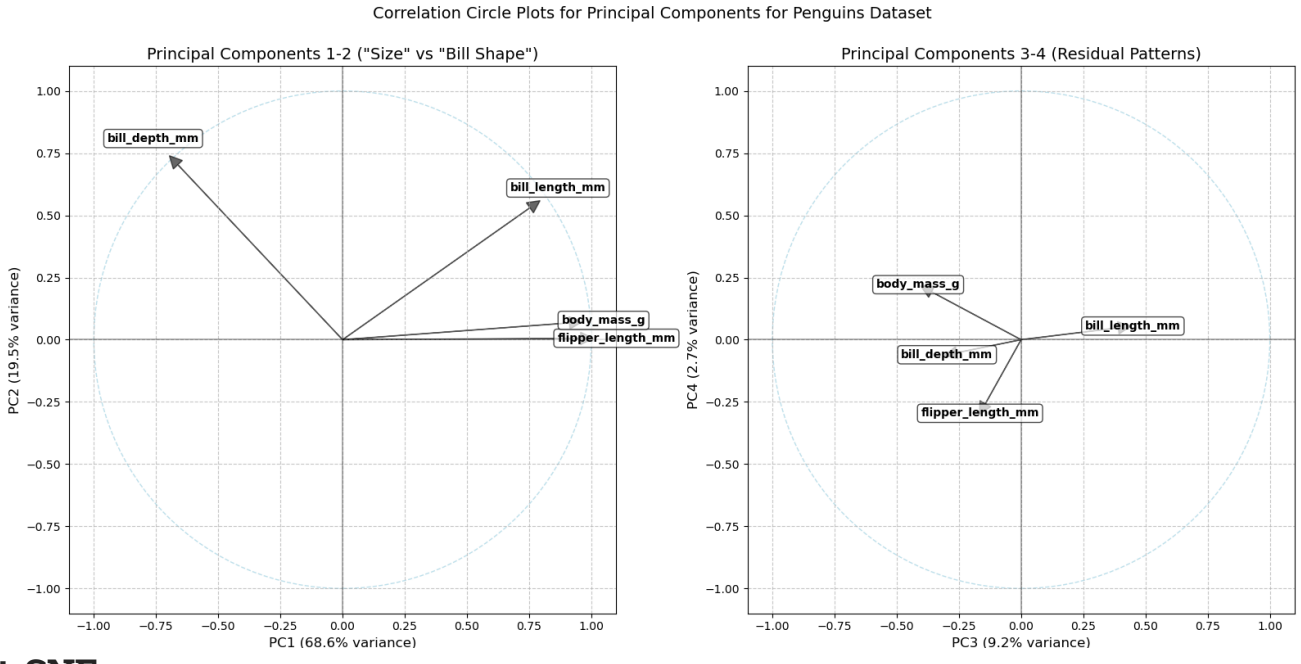
To gain deeper insight into how the original variables relate to the principal components, we can use a **correlation circle**, also known as a **variable factor map**. This graphical tool provides a visual representation of the contribution and correlation of each original variable with the components.

In a 2D PCA plot (PC1 *vs.* PC2 in the left subplot, and PC3 *vs.* PC4 in the right subplot), each original feature is represented as a vector (arrow) pointing from the origin.

- The direction of the arrow indicates whether the variable (feature) is positively or negatively correlated with the principal components.
- The length of the arrow indicates the strength of correlation — longer arrows mean the variable (feature) contributes strongly to the components.
- The circle itself (radius = 1) represents the maximum possible correlation between a variable (feature) and the components, since PCA projects standardized variables (features) .

In addition, the correlations between variables (features) can also be specified.

- Variables (features) pointing in similar directions are positively correlated (body mass and flipper length), that is why we performed regression task yesterday using these two features.
- Variables (features) pointing in opposite directions are negatively correlated (for specific pairs of components).
- Variables (features) at roughly 90° to each other are nearly uncorrelated.



t-SNE

Since PCA is based on linear combinations of all features, it has some inherent limitations. In particular, it may fail to capture complex, non-linear relationships in the data, and it primarily focuses on maximizing global variance, which can overlook subtle local structures, intricate clusters, or hierarchical patterns – that are quite common in real-world datasets.

To address these challenges, we move beyond linear methods and explore advanced non-linear dimensionality reduction techniques. Algorithms such as **t-SNE** (t-Distributed Stochastic Neighbor Embedding) and **UMAP** (Uniform Manifold Approximation and Projection) are particularly effective at capturing the non-linear structure of the data. These

methods are designed to preserve local neighborhood relationships, revealing intricate patterns that PCA might miss and providing a more detailed view of the underlying structure of the penguin population.

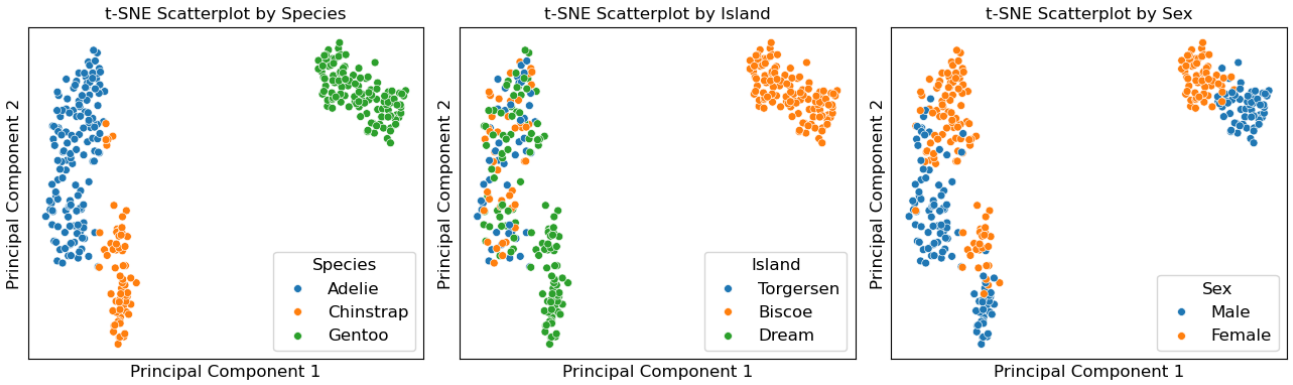
In this subsection, we apply the t-SNE algorithm to visualize local structures and potential clusters, and we will explore UMAP for complementary insights in the next subsection.

t-SNE, is a powerful dimensionality reduction technique primarily used for visualizing high-dimensional data in two or three dimensions. Unlike linear methods such as PCA, t-SNE is non-linear and focuses on preserving the local structure of the data, meaning that points that are close in the high-dimensional space remain close in the lower-dimensional embedding. This is the main idea of t-SNE and it models the pairwise similarities between data points in both the high-dimensional and low-dimensional spaces.

We build a t-SNE model with 2 principle components using the `TSNE` class in the `sklearn.manifold` before fitting the model. The hyperparameter `perplexity` controls the number of effective neighbors each point considers when learning the embedding. Higher values emphasize global structure, lower values emphasize local relationships.

```
from sklearn.manifold import TSNE

# build a t-SNE model having 2 PCs
tsne = TSNE(n_components = 2, perplexity = 50)
X_tsne = tsne.fit_transform(X_scaled)
```

After fitting the t-SNE model, we can visualize the Penguin dataset in two dimensions, allowing us to explore the relationships between individual data points, identify potential clusters, and observe how different species are distributed across the embedding.



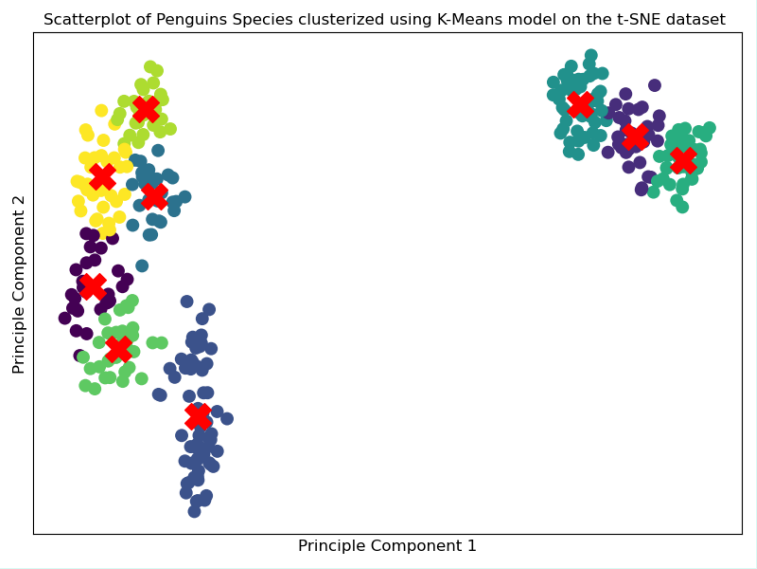
Warning

It is noted that **t-SNE is primarily a visualization tool rather than a feature extraction method**, and the **resulting embedding should not be used directly for downstream tasks such as classification**. Each time you run t-SNE, the coordinates can change slightly due to randomness, even with the same data. As such, the features produced by t-SNE may not be stable or meaningful for predictive modeling.

✍️ Exercise

Here, we will perform a clustering task using K-Means on a dataset reduced to two components obtained from the t-SNE model (code examples are availalbe in the Jupyter Notebook.

- Group the data into 9 clusters `kmeans = KMeans(n_clusters=9)`.
- Repeat the clustering several times to observe how the cluster assignments change.
- Change the number of clusters (*e.g.*, 5 or 11) to see how it affects the result.



UMAP

UMAP (Uniform Manifold Approximation and Projection) is a non-linear dimensionality reduction technique similar in purpose to t-SNE, but with some key differences:

- Preserves both local and some global structure of the data.
- Generally faster and more scalable than t-SNE, especially for large datasets.
- Can be used for visualization (2D/3D) or as a preprocessing step for downstream tasks like clustering or classification.

UMAP is based on **manifold learning** and **graph theory**.

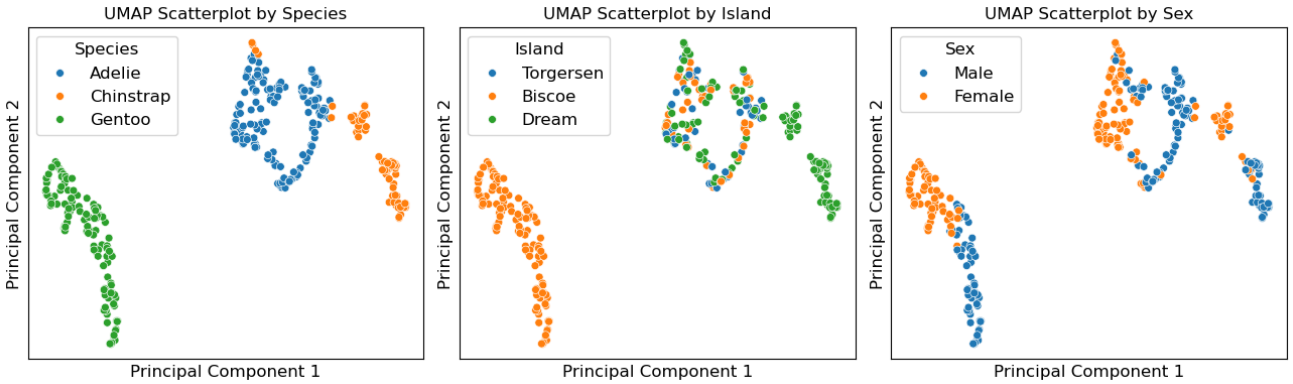
- Manifold learning means it assumes that even though our data might have many dimensions, the real structure of the data lies on a lower-dimensional surface (like a curve or sheet) inside that high-dimensional space. UMAP tries to find and keep that structure.
- Graph theory means UMAP represents the data as a graph: each point is a node, and connections (edges) show how close points are to each other. Then it uses math to squeeze this graph down into 2D or 3D while keeping the structure as much as possible.

UMAP it constructs a high-dimensional graph representation of the data and then projects it onto a low-dimensional space, by applying a series of optimization steps. This results in a visual representation of the data that is both accurate and interpretable. In comparison to

PCA and t-SNE, UMAP offers a good balance of accuracy, efficiency, and scalability, making it a popular choice for dimensionality reduction in machine learning and data analysis.

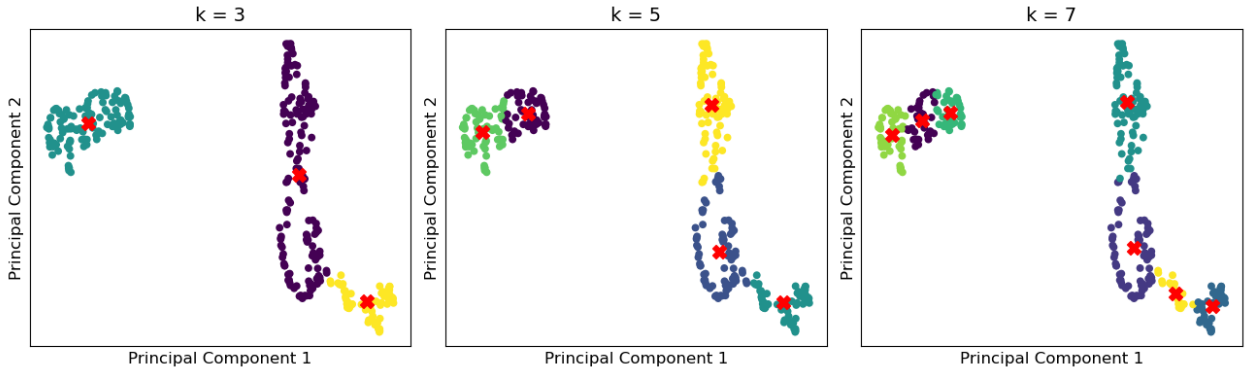
Using the same procedure as in the t-SNE subsection, we build and fit the UMAP model, and visualize the Penguin dataset in two dimensions by species, island, and sex.

```
import umap

umap_model = umap.UMAP(n_components = 2, n_neighbors=10)
X_umap = umap_model.fit_transform(X_scaled)

X_umap_species = penguins_dimR.join(pd.DataFrame(X_umap,
                                                 index =
penguins_dimR.index,
                                                 columns =
['PC_1', 'PC_2']))
```



✍ Exercise

Here we perform the same clustering task using K-Means on a dataset reduced to two components obtained from the UMAP model (code examples are availalbe in the Jupyter Notebook.



Keypoints

- Representative method include PCA, t-SNE, and UMAP
- PCA is a method to reduce dimensions via creating new variables called principal components (PCs), which are linear combinations of the original features.
- Perform PCA task and then decide optmal number of component.

- T-SNE and UMAP are both nonlinear dimensionality reduction methods mainly used to visualize high-dimensional data in 2D or 3D.
- T-SNE focuses on keeping similar points close and dissimilar points apart in lower-dimensional space, but not applicable for feature reduction or predictive modeling.
- UMAP preserves both local relationships (nearby points stay close) and some global structure, scales well to large datasets, and is mainly used for exploration and visualization.

Quick Reference

Instructor's guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Who is the course for?

About the course

See also

Credits