Machine Learning: Scikit-learn + PyTorch

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

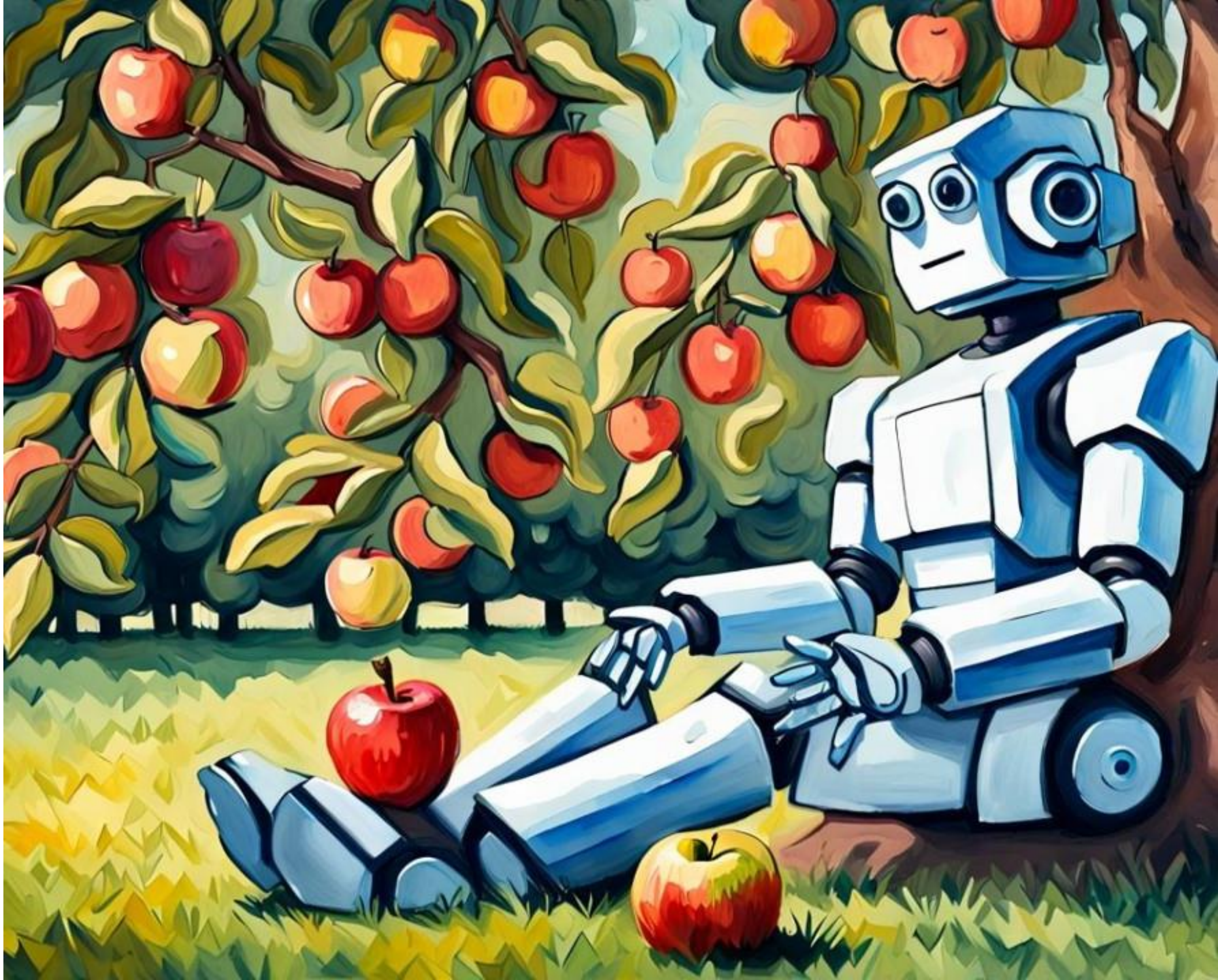
Types of ML

Metrics

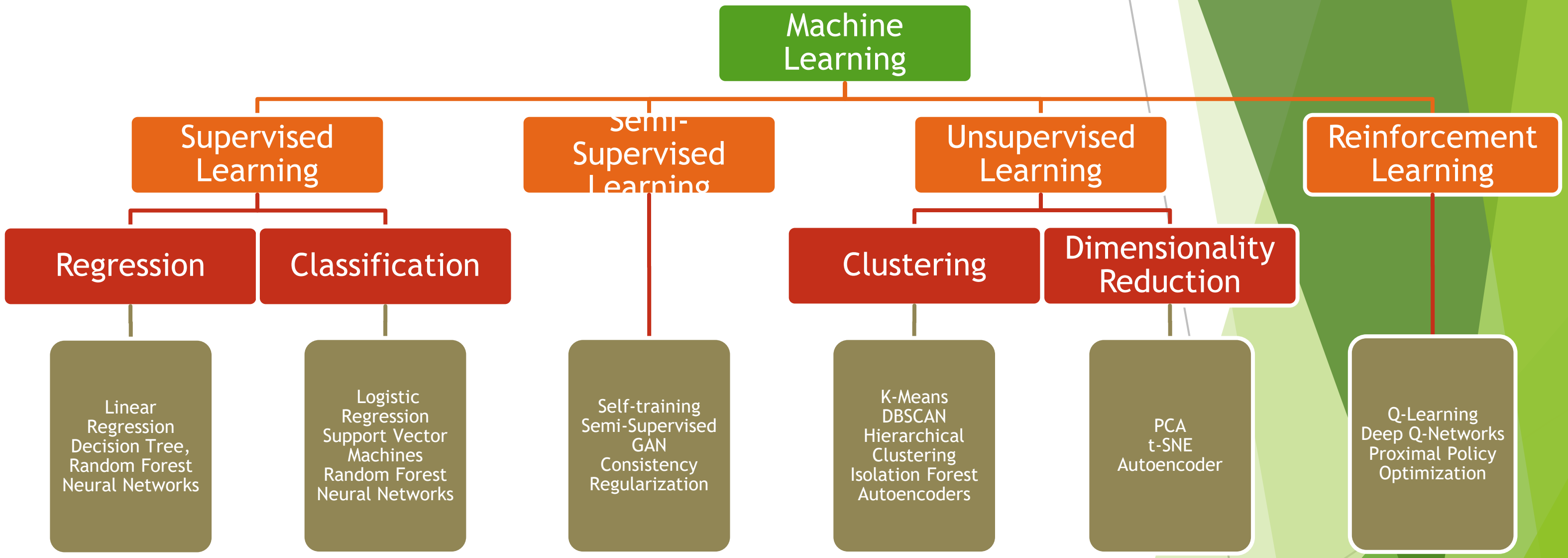
Scikit-learn

PyTorch

Types of ML



ML


Machine Learning
Supervised Learning
Semi-Supervised Learning
Unsupervised Learning
Reinforcement Learning
Regression
Classification
Clustering
Dimensionality Reduction
Linear Regression
Decision Tree,
Random Forest
Neural Networks
Logistic Regression
Support Vector Machines
Random Forest
Neural Networks
Self-training
Semi-Supervised GAN
Consistency Regularization
K-Means
DBSCAN
Hierarchical Clustering
Isolation Forest
Autoencoders
PCA
t-SNE
Autoencoder
Q-Learning
Deep Q-Networks
Proximal Policy Optimization

Supervised Learning

- Supervised Learning is a type of **Machine Learning (ML)** where a model learns from **labeled data**. This means that each input data point (X) is associated with a correct output (Y), and the model is trained to map inputs to outputs.

- **Training Phase**:
  The model is provided with input-output pairs (X, Y).It learns a function $f(X) \rightarrow Y$ that best fits the data.

- **Prediction Phase**:
  Once trained, the model can predict Y' for new, unseen inputs X'.

Supervised Learning

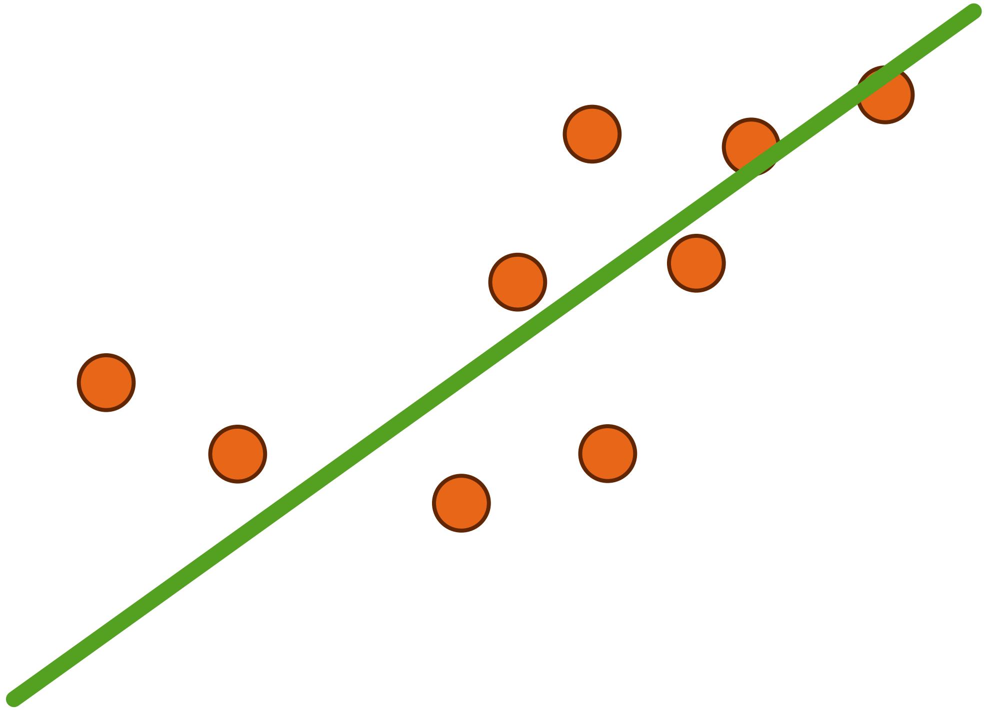
Regression

- Predicting **continuous values** (e.g., temperature, price)
- Algorithms
  - Linear Regression
  - Decision Trees
  - XGBoost
  - LSTM
  - ARIMA

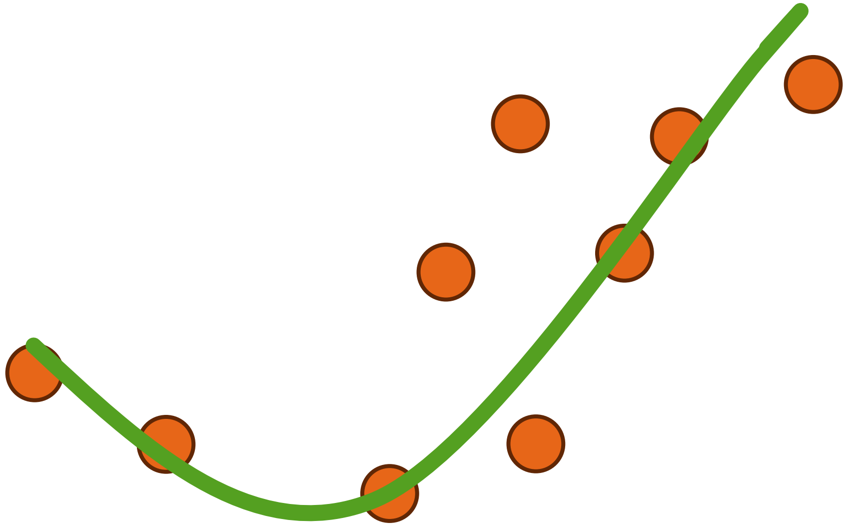
Classification

- Predicting **discrete categories** (e.g., spam vs. not spam)
- Algorithms
  - Logistic Regression
  - Decision Trees
  - K-nn
  - CNN

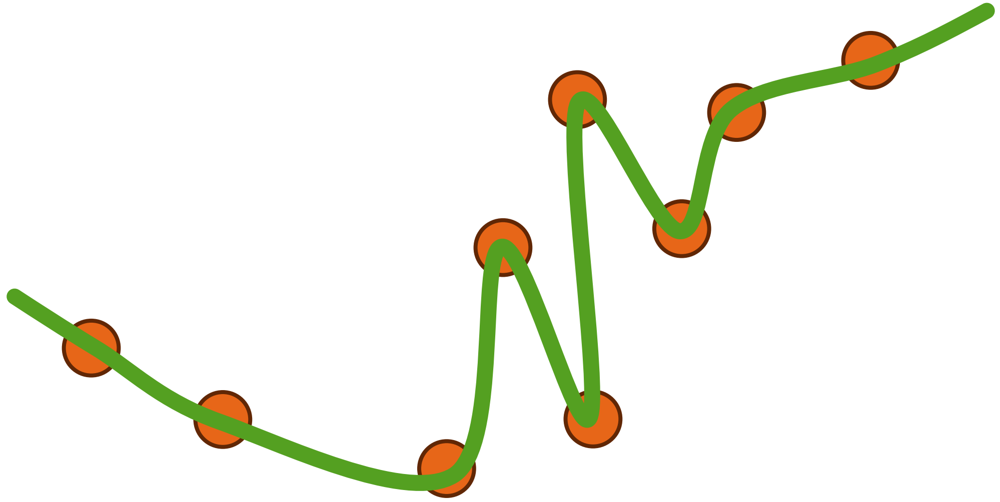
Regression



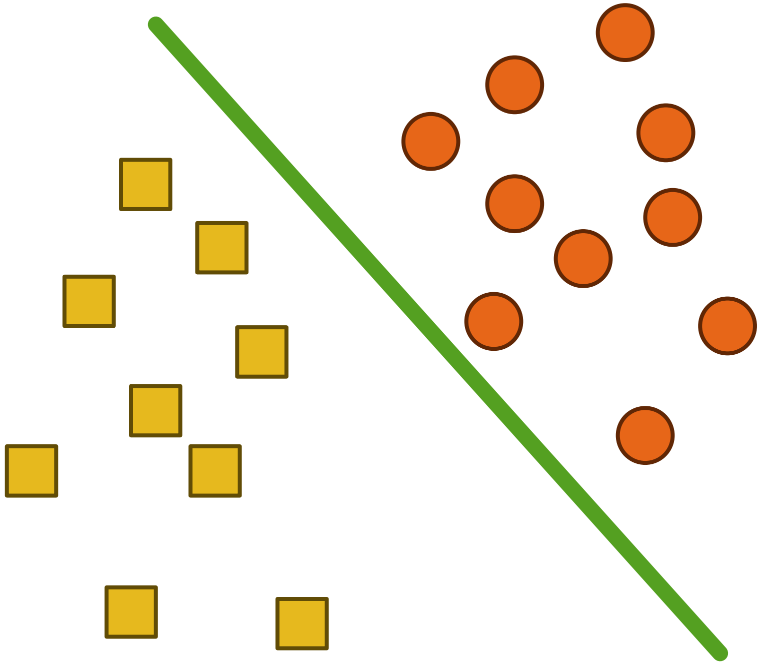
Regression



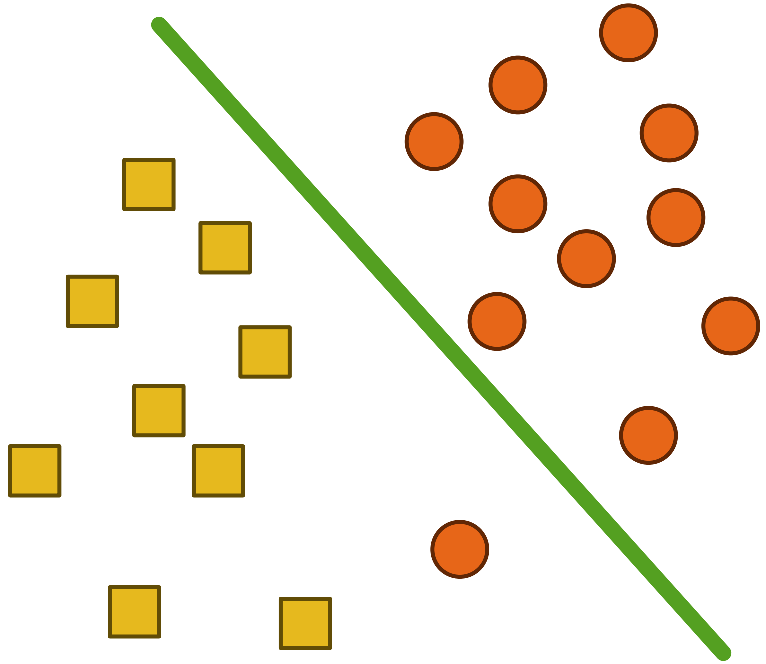
Regression



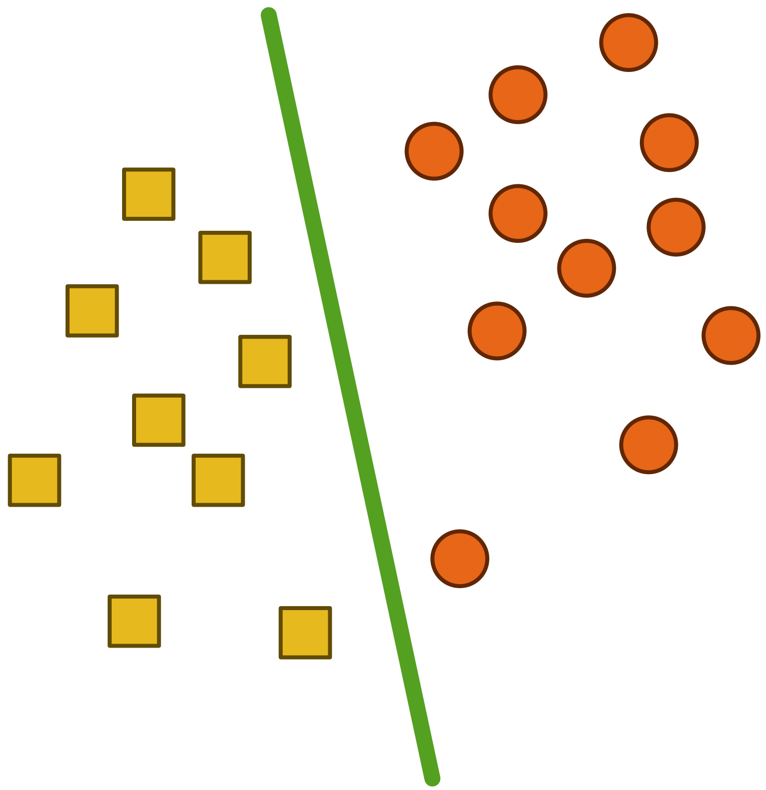
Classification



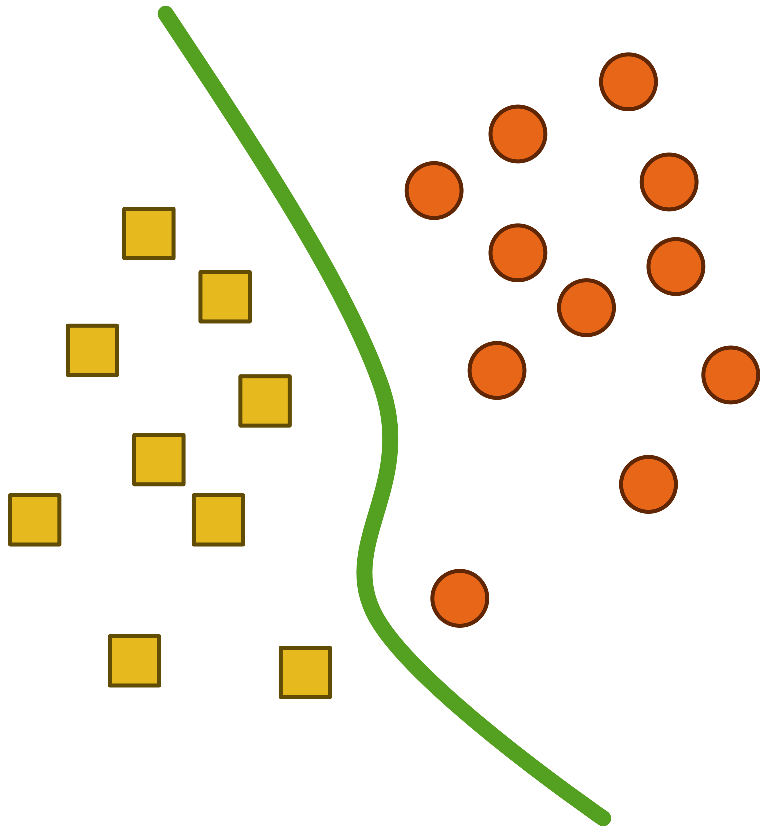
Classification



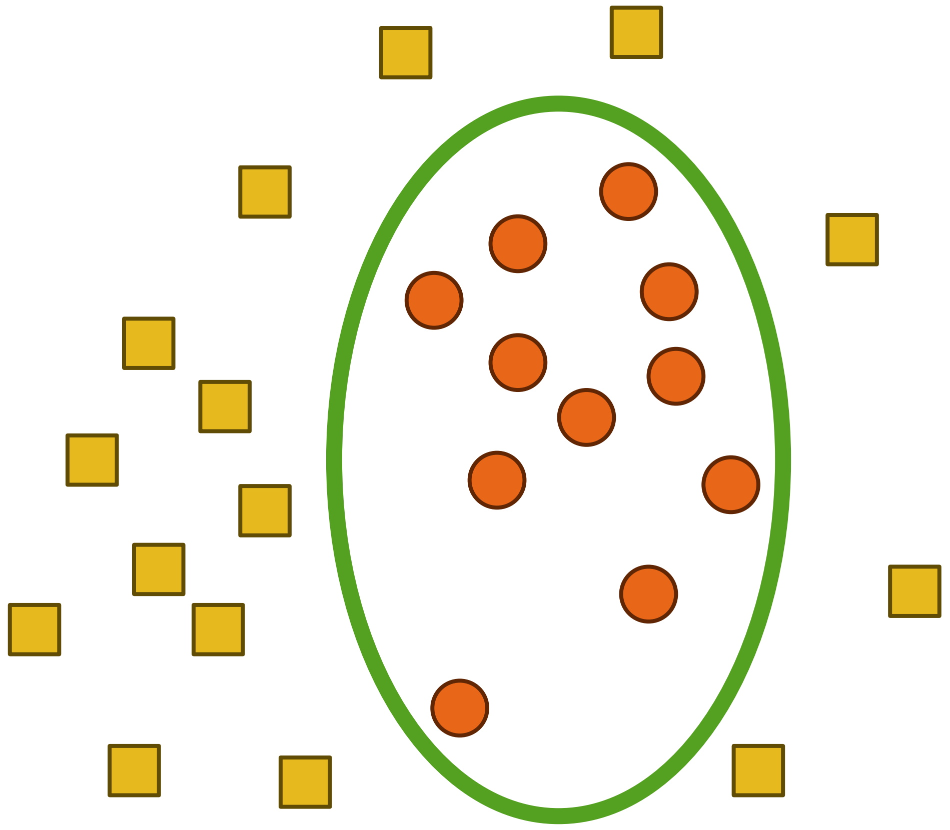
Classification



Classification



Classification



Unsupervised Learning

- Unsupervised Learning is a type of **Machine Learning (ML)** where the model is trained on **unlabeled data**—meaning there are no predefined outputs (Y). Instead, the model identifies **patterns, structures, and relationships** in the data without explicit guidance.

1. The model receives input data (X) without labels.
2. It analyzes the data to detect patterns, clusters, or structures.
3. Finds hidden relationships and groups similar data points together.

Unsupervised Learning

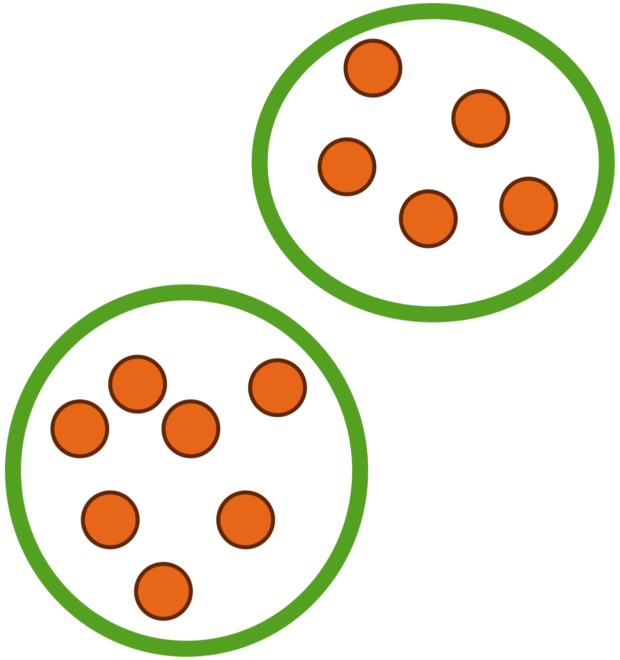
Clustering

- Grouping similar data points together
- Algorithms
  - **Customer Segmentation**
    - K-Means
    - DBSCAN
    - Hierarchical Clustering
  - **Anomaly Detection**
    - Isolation Forest
    - One-Class SVM

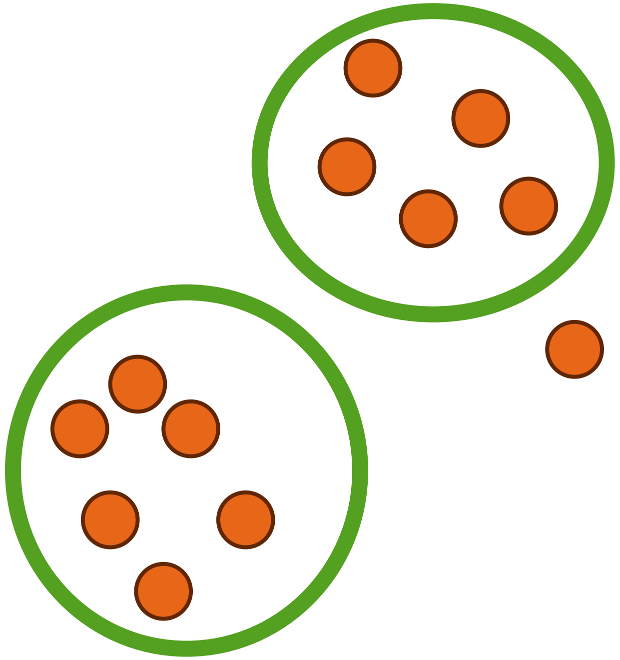
Dimensionality Reduction

- Reducing the number of features while retaining important information.
- Algorithms
  - Feature Extraction
    - PCA
    - t-SNE
    - Autoencoder
  - Topic Modeling in NLP
    - Latent Dirichlet Allocation (LDA)

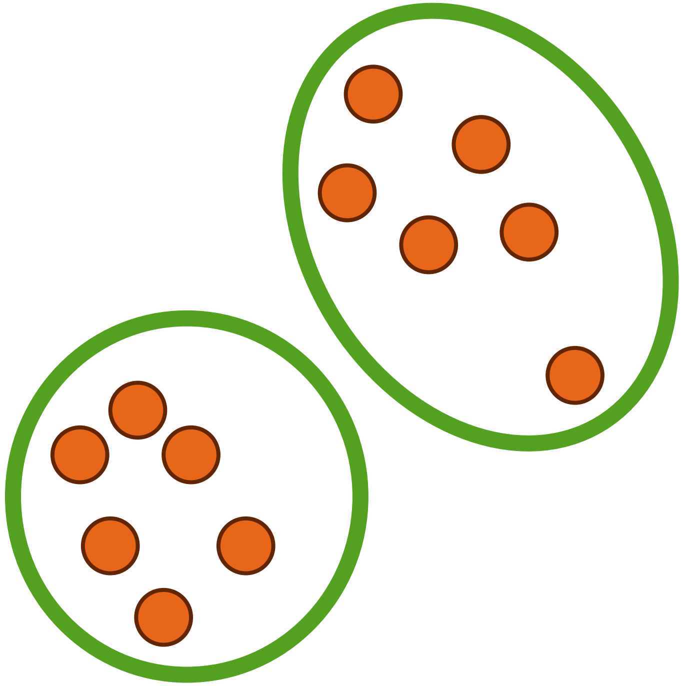
Clustering



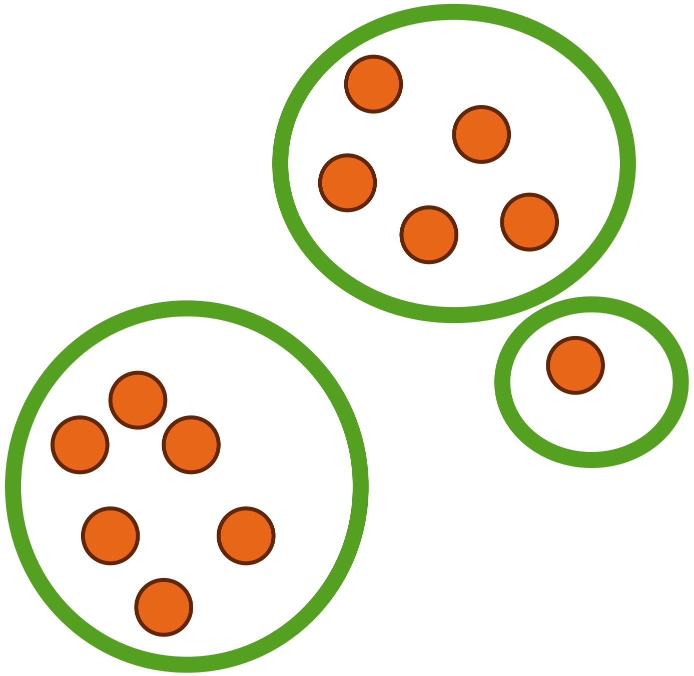
Clustering



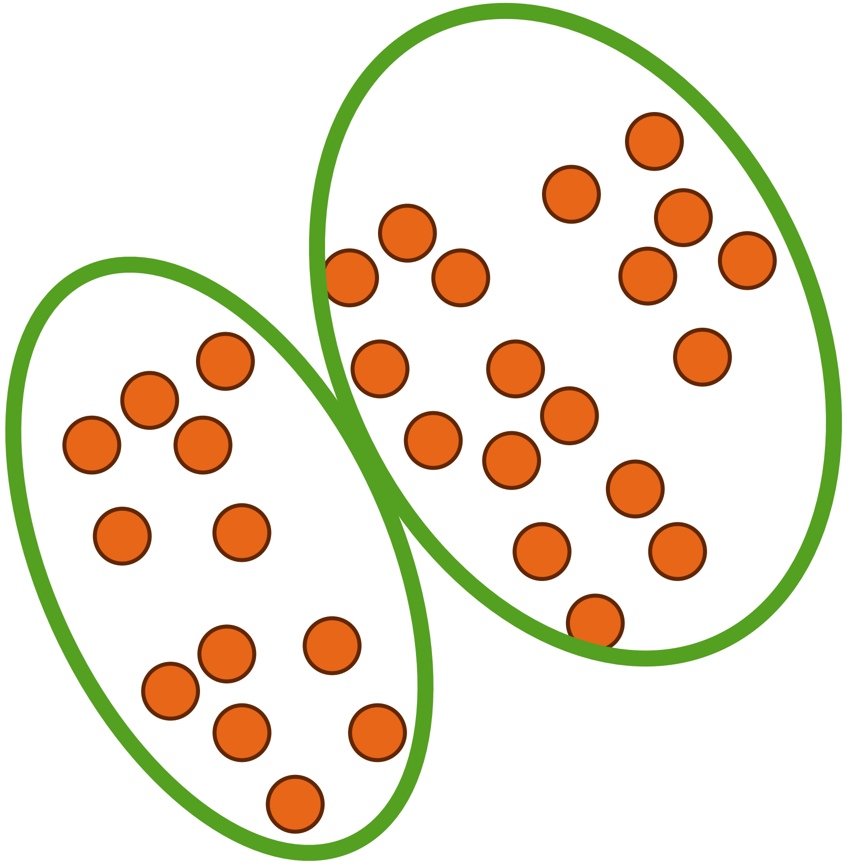
Clustering



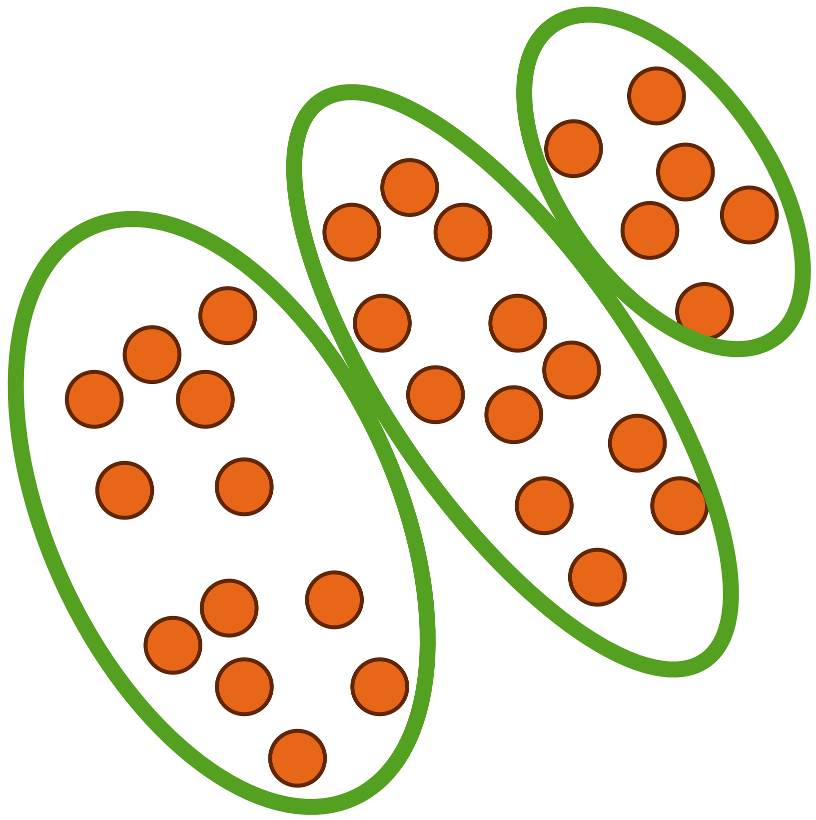
Clustering



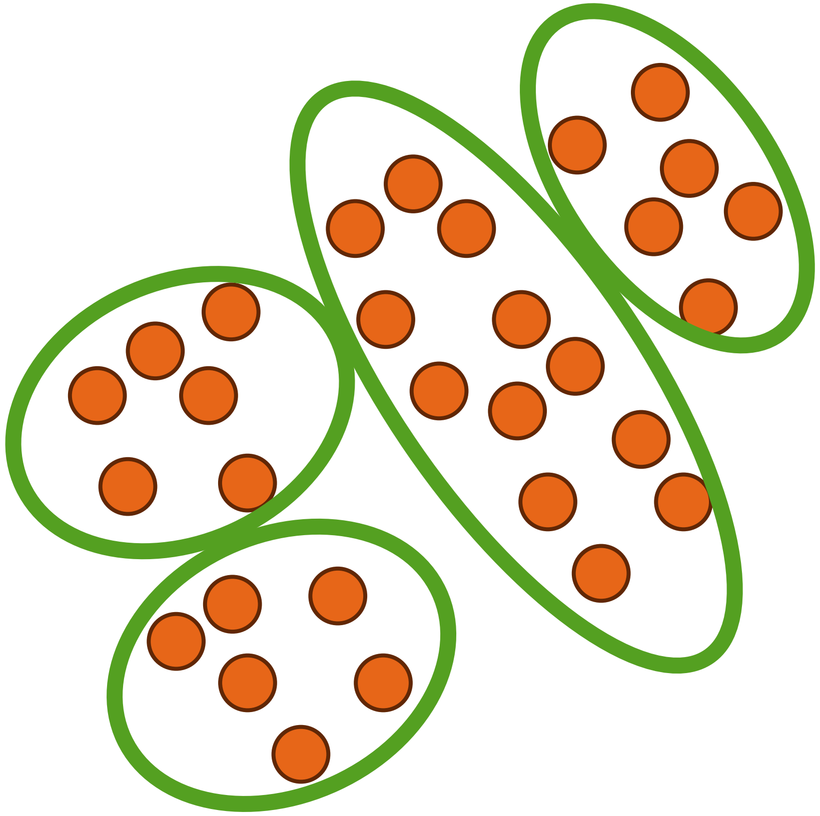
Clustering



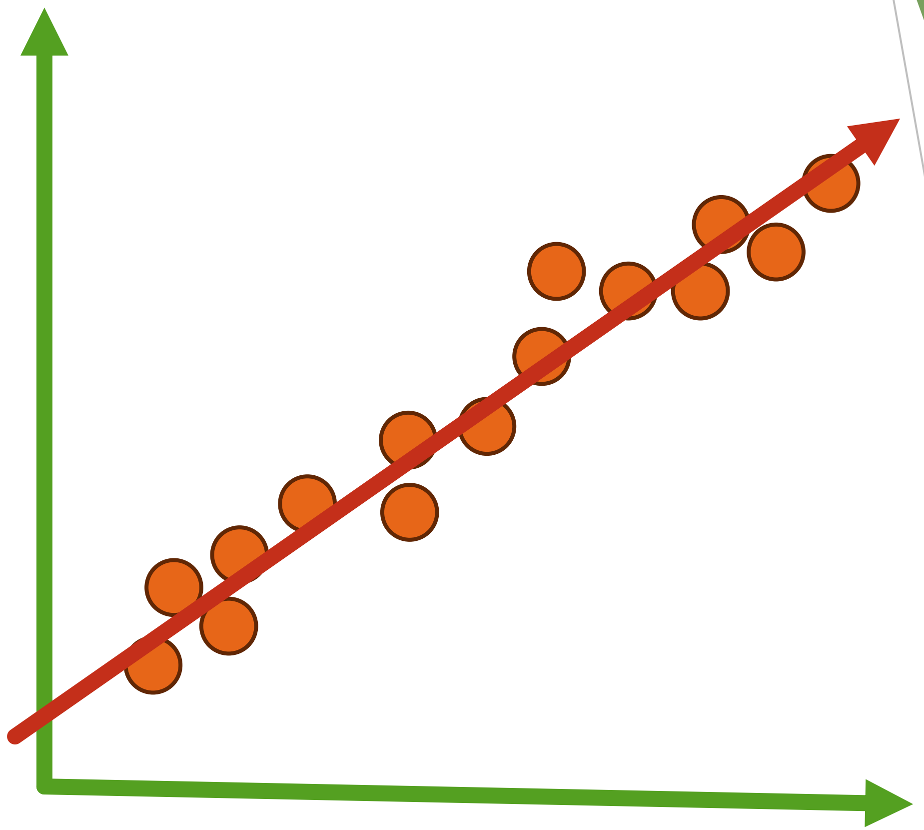
Clustering



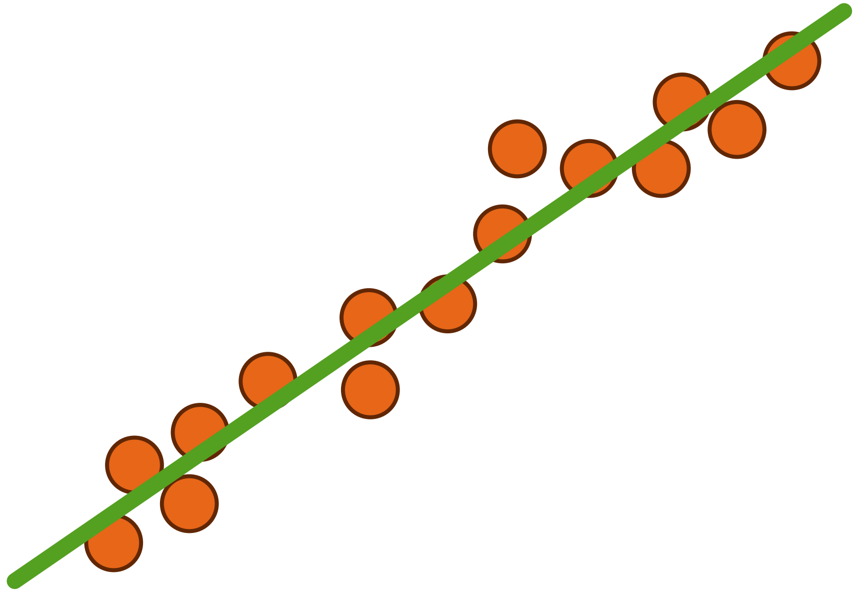
Clustering



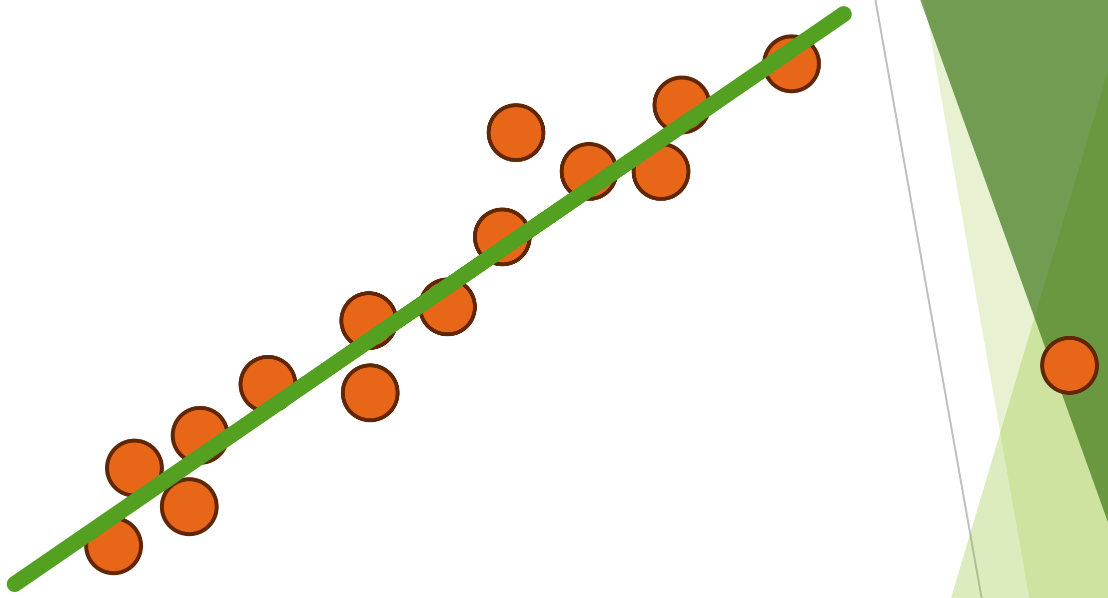
Dimensionality Reduction



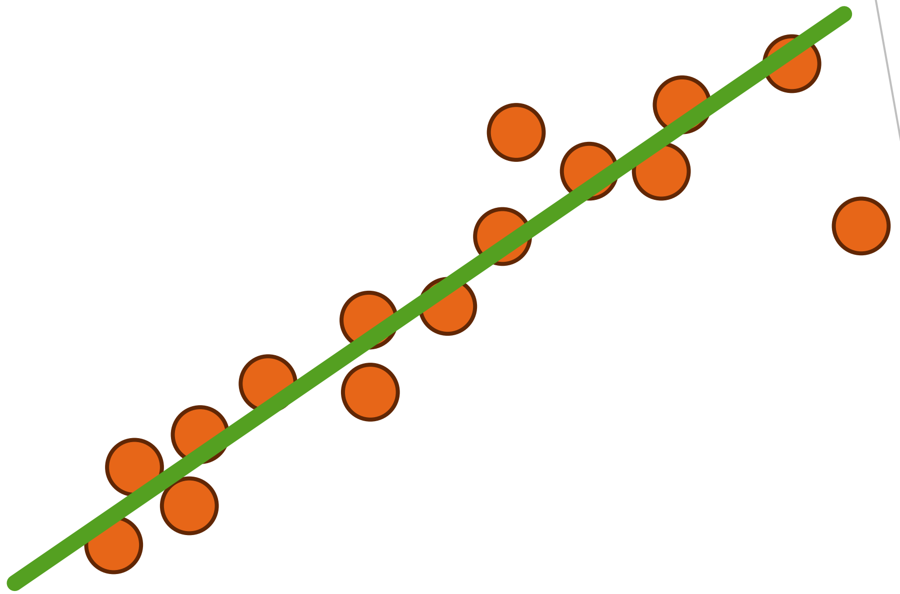
Anomaly detection



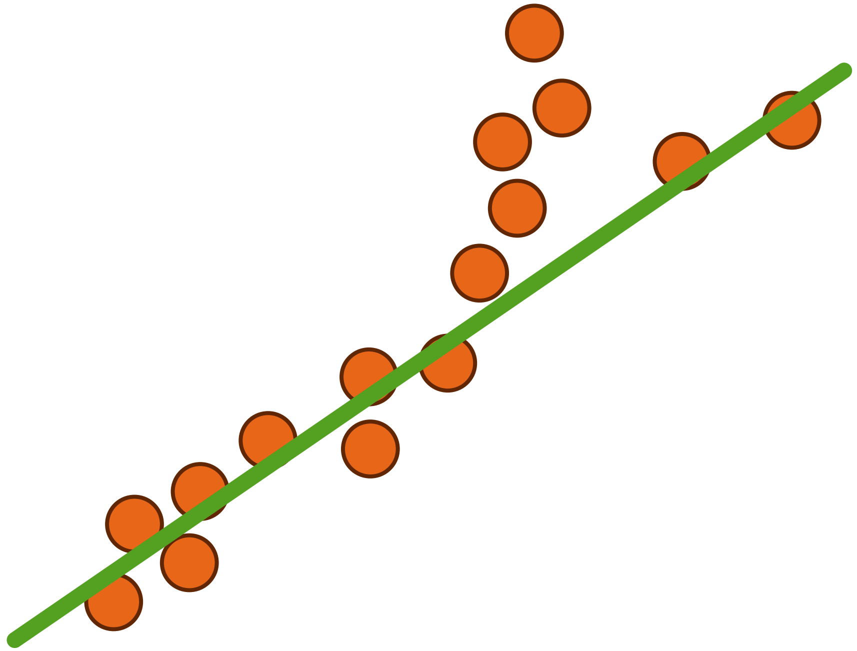
Anomaly detection



Unsupervised Learning



Anomaly detection



Semi-supervised Learning

- **Semi-Supervised Learning** is a type of **Machine Learning (ML)** that combines elements of both **Supervised** and **Unsupervised Learning**. It uses a **small amount of labeled data** and a **large amount of unlabeled data** to train a model.

1. A small set of labeled data (X, Y) is used to guide the learning process.
2. A large set of unlabeled data (X') helps the model generalize better.
3. The model uses patterns in unlabeled data to refine its predictions.

Semi-supervised Learning

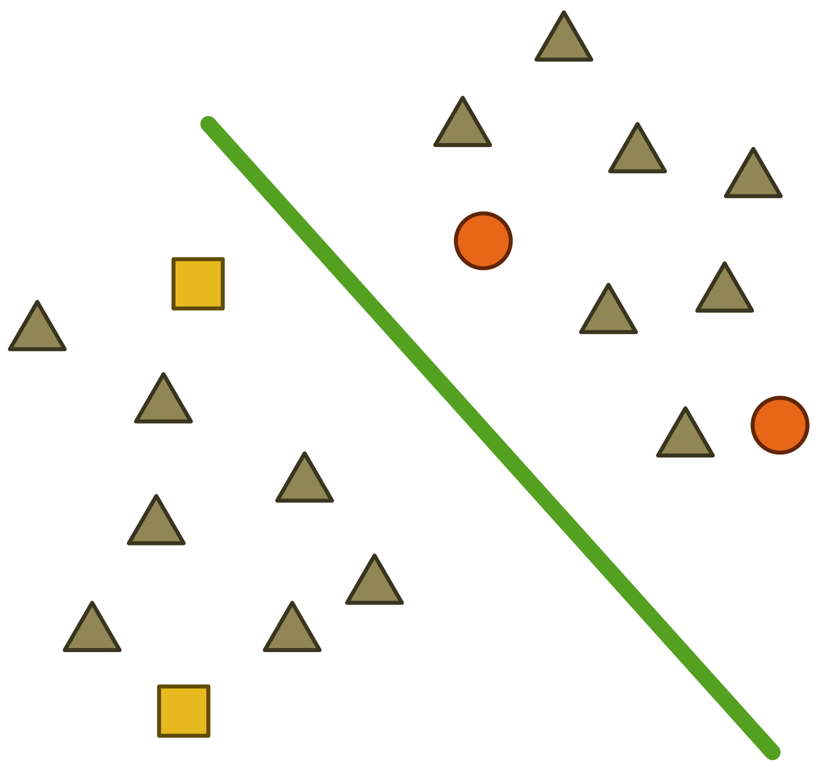
Problems

- Image Classification with Limited Labeled Data
- Speech Recognition
- Text Classification (NLP)

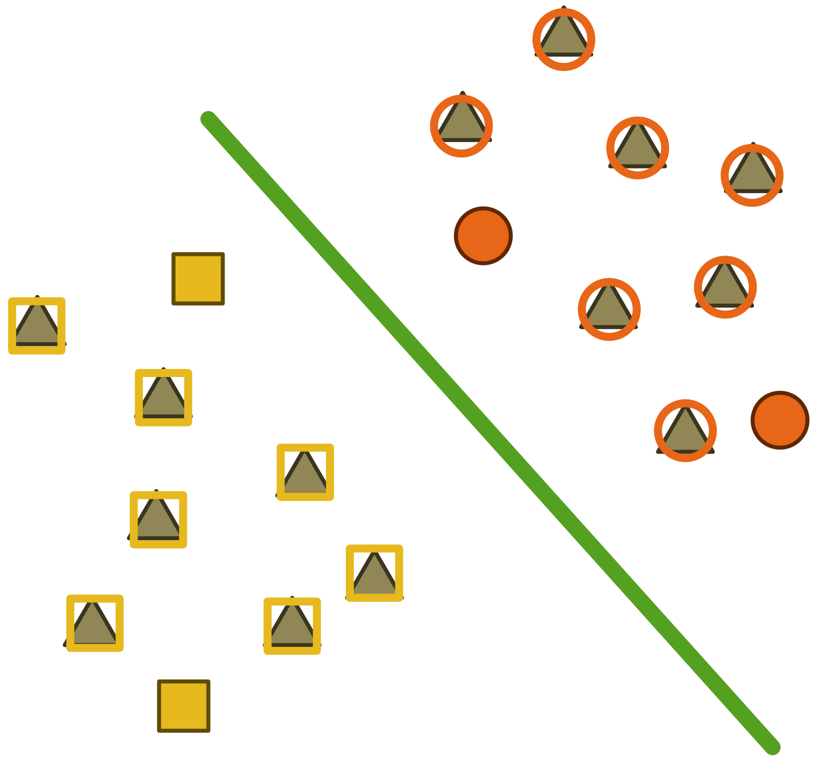
Algorithms

- Self-training,
- Semi-Supervised GANs,
- FixMatch
- Graph Neural Networks (GNNs),
- Variational Autoencoders (VAE)
- Self-training,
- Label Propagation,
- GPT fine-tuning

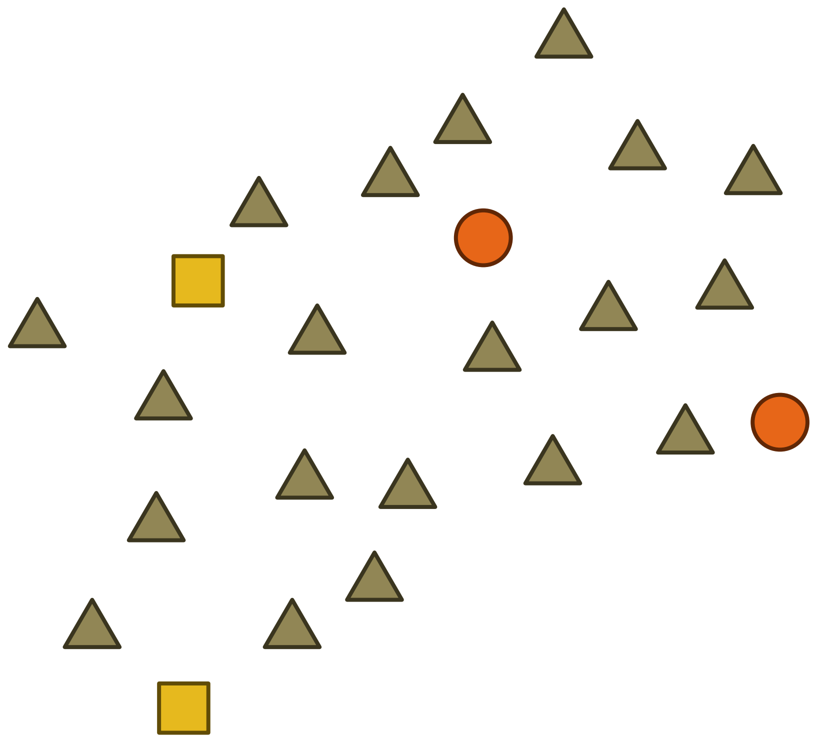
Semi-Supervised Learning



Semi-Supervised Learning



Semi-Supervised Learning



Reinforcement Learning

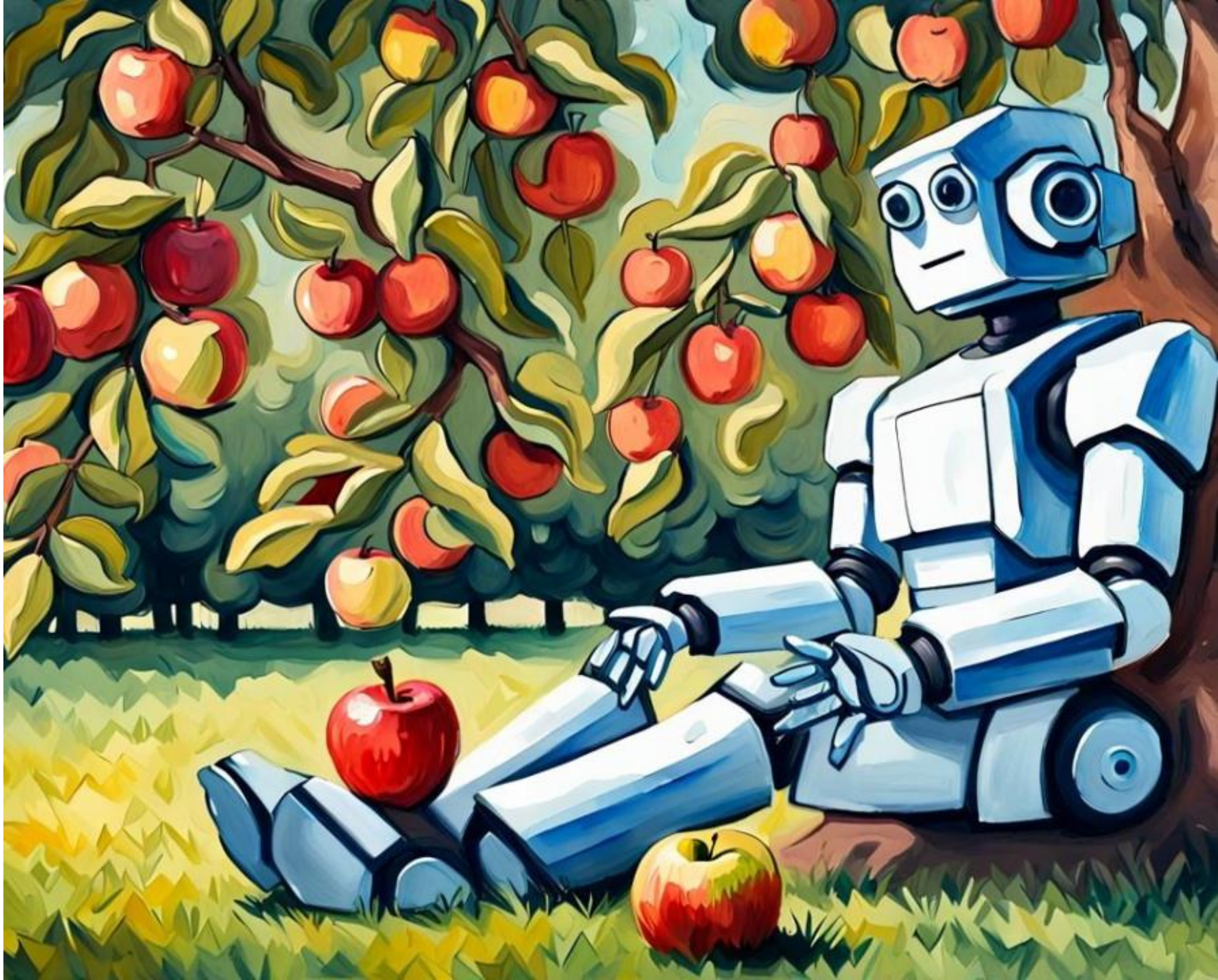
- **Reinforcement Learning (RL)** is a type of **Machine Learning (ML)** where an **agent** learns by interacting with an **environment** to maximize cumulative rewards. Unlike supervised learning, RL does not require labeled data—it learns through trial and error.

1. **Agent:** The decision-maker (e.g., a robot, AI player).
2. **Environment:** The system in which the agent operates (e.g., a game, a self-driving car simulation).
3. **Actions (A):** Choices the agent can make.
4. **Rewards (R):** Positive or negative feedback received for actions taken.
5. **Policy (π):** A strategy that decides the agent's actions based on the current state.

- LEARNING Process:
  The agent takes an **action (A)** → The environment responds with a **reward (R)** and a **new state (S')** → The agent updates its policy to maximize future rewards.

Reinforcement Learning

Metrics

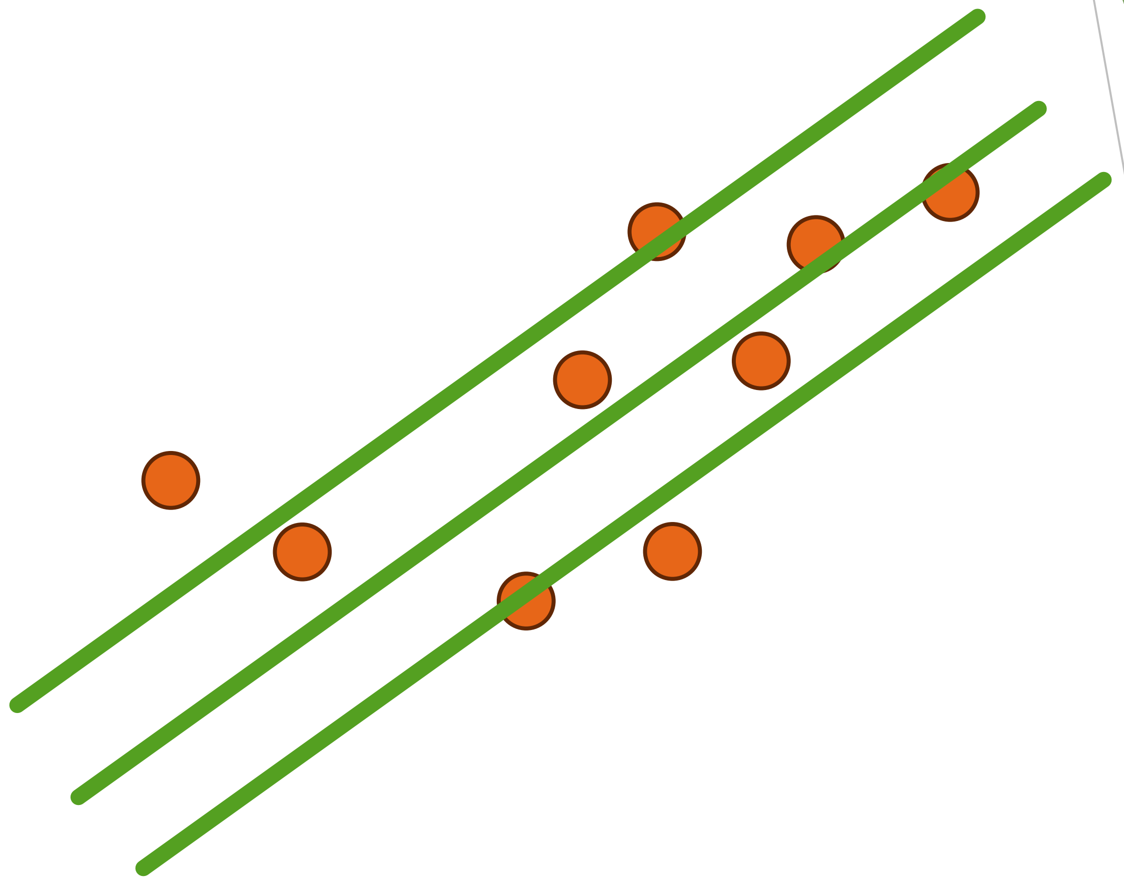


Metrics in Machine Learning

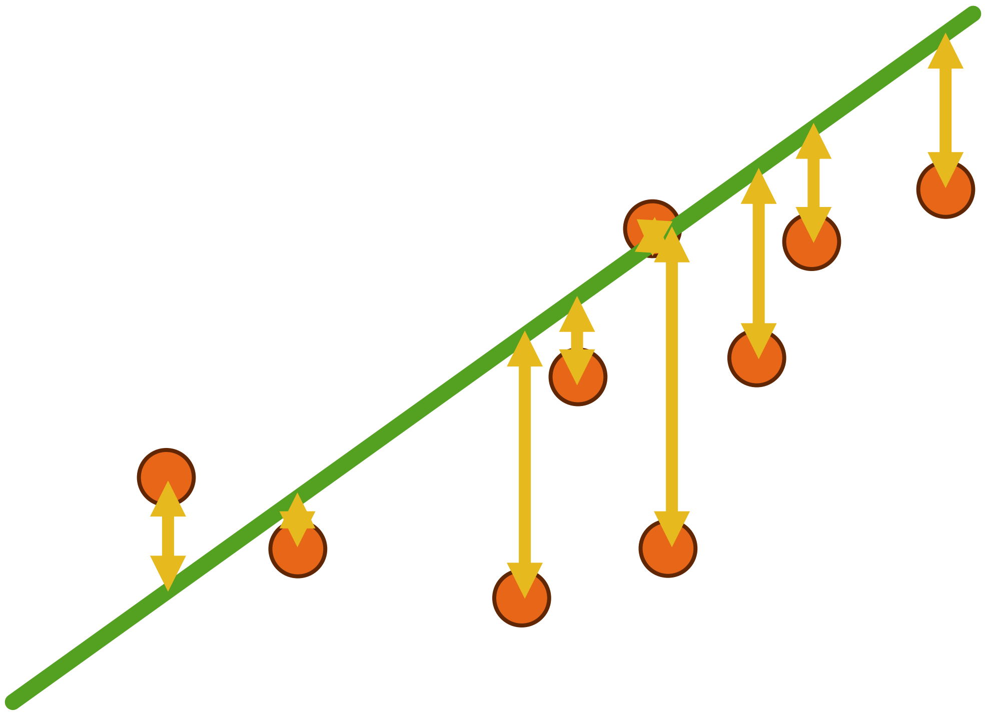
- **Metrics** in Machine Learning are used to evaluate the performance of a model. The choice of metric depends on the type of problem:

1. **Regression Metrics** (For continuous output)
2. **Classification Metrics** (For categorical output)
3. **Clustering Metrics** (For unsupervised learning)

Regression



Regression



Mean Squared Error (MSE)

- **Mean Squared Error (MSE)** is a commonly used metric in regression problems that measures the average squared difference between the actual values ($\mathbf{y}^{\text{true}}$) and the predicted values ($\mathbf{y}^{\text{pred}}$).
- It helps evaluate how well a model's predictions match the actual values.
- **Lower MSE means better accuracy.**

$$MSE = \frac{1}{N}\sum_{n=1}^{N}\left(y_n^{true} - y_n^{pred}\right)^2$$

R^2 Score Coefficient of Determination

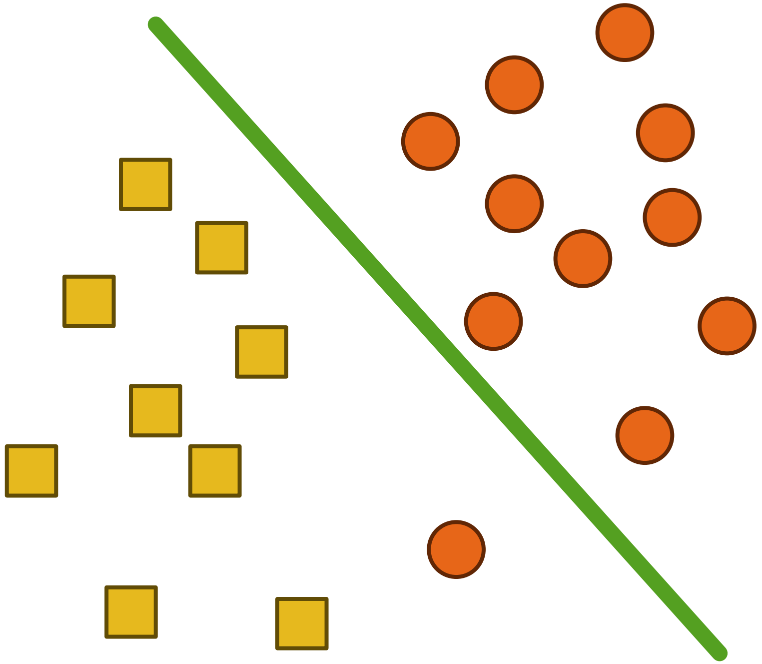
- The **R^2 score**, or **coefficient of determination**, is a metric used to evaluate how well a regression model fits the data. It measures the proportion of variance in the dependent variable (**y^{truey}**) that is predictable from the independent variable(s) (**X**).

- **$R^2 = 1$** → Perfect model (explains 100% of variance)
- **$R^2 = 0$** → Model performs as poorly as the mean
- **$R^2 < 0$** → Model is worse than just predicting the mean

The **closer R^2 is to 1**, the better the model explains the variation in data.

$$MSE = 1 - \frac{\sum_{n=1}^{N}\left(y_n^{true} - y_n^{pred}\right)^2}{\sum_{n=1}^{N}(y_n^{true} - \hat{y})^2}$$

Classification



Confusion Matrix

Actual\Predicted	Positive	Negative
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Positive (TN)

- **True Positive (TP):** Model correctly predicted **positive** (e.g., correctly detecting spam).
- **True Negative (TN):** Model correctly predicted **negative** (e.g., correctly identifying non-spam).
- **False Positive (FP):** Model **incorrectly** predicted **positive** (e.g., classifying a regular email as spam → Type I error).
- **False Negative (FN):** Model **incorrectly** predicted **negative** (e.g., failing to detect spam → Type II error).

Classification

- **Accuracy** – the percentage of correctly classified samples.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Precision** - how many predicted positives are actually correct.

$$Precision = \frac{TP}{TP + FP}$$

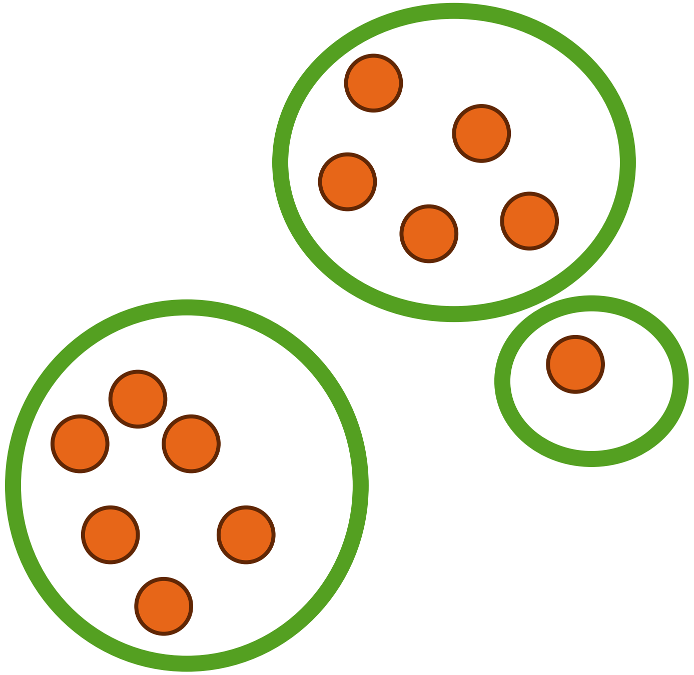
- **Recall** - how many actual positives were correctly predicted.

$$Recall = \frac{TP}{TP + FN}$$

- **F1-score** - harmonic mean of precision and recall.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Clustering



Clustering

- **Silhouette Score**
  Measures how well-separated and dense clusters are.

- 1 → Well-defined clusters
- 0 → Overlapping clusters
- -1 → Poor clustering

$$S = \frac{b - a}{\max(a, b)}$$

a - average intra-cluster distance
b - average nearest-cluster distance (separation)

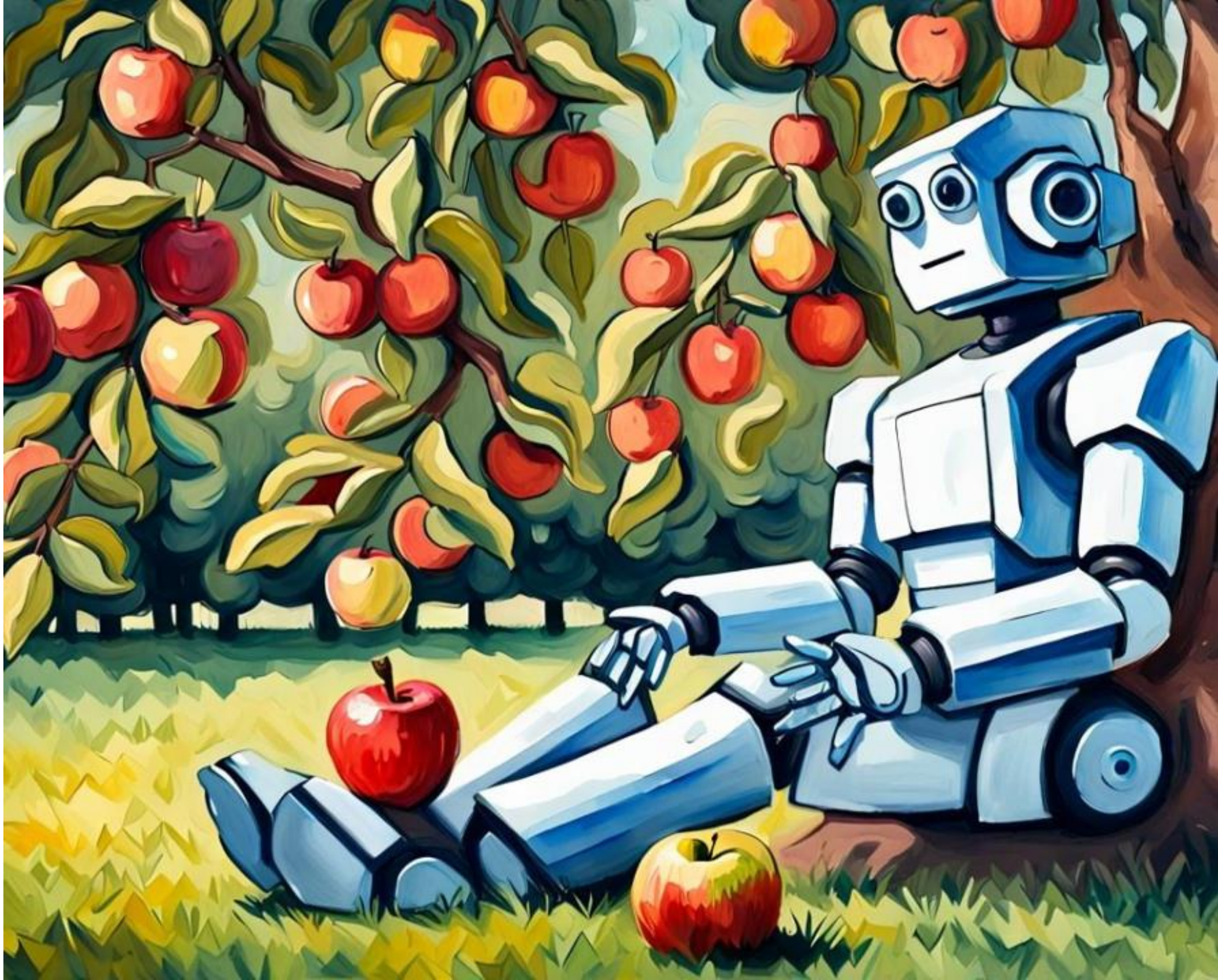
- **Davies-Bouldin Index**
  Measures intra-cluster similarity vs. inter-cluster separation.
  Lower values are better

- **Calinski-Harabasz Index**
  Measures the ratio of intra-cluster variance to inter-cluster variance.
  Higher values are better (more compact and well-separated clusters).

Dataset Split

Dataset	Purpose	Common split (%)
Training	Model training	60-80
Validation	Hyperparameter tuning	10-20
Test	Final model evaluation	10-20

Scikit-learn



Steps to build a ML model

1. Define Problem – Identify ML type (Classification, Regression).
2. Collect Data – Gather structured/unstructured data.
3. Clean Data – Handle missing values, outliers.
4. Split Data – Train, Validation, and Test sets.
5. Feature Engineering – Visualize data & create new features.
6. Choose Model – Select ML algorithm.
7. Train Model – Fit model using training data.
8. Evaluate Model – Check accuracy, MSE, R^2.
9. Optimize Model – Tune hyperparameters.
10. Deploy Model – Save and use in production.

Define Problem

Before starting, clearly define the **goal** of your ML model.

Key Questions:

- What **type of problem** is this? (Classification, Regression, Clustering, etc.)
- What is the **input (features)** and **output (target variable)?**
- What will be the **evaluation metric**? (Accuracy, MSE, etc.)

Collect Data

- Data can come from **CSV files, databases, APIs, or web scraping.**

- Load Data from CSV:

```
import pandas as pd

# Load dataset
df = pd.read_csv("data.csv")

# Display first few rows
print(df.head())
```

Clean Data

- Before training, the data needs to be **cleaned and prepared.**

Common Issues & Fixes:

- **Missing Values** → Fill or Drop
- **Duplicate Rows** → Remove
- **Outliers** → Detect & Handle
- **Incorrect Data Types** → Convert

```
# Fill missing values with mean
df_mean = df["Income"].mean()
df["Income"].fillna(df_mean, inplace=True)
```

Split Data

- To evaluate the model properly, the dataset should be split into **training, validation, and test sets.**

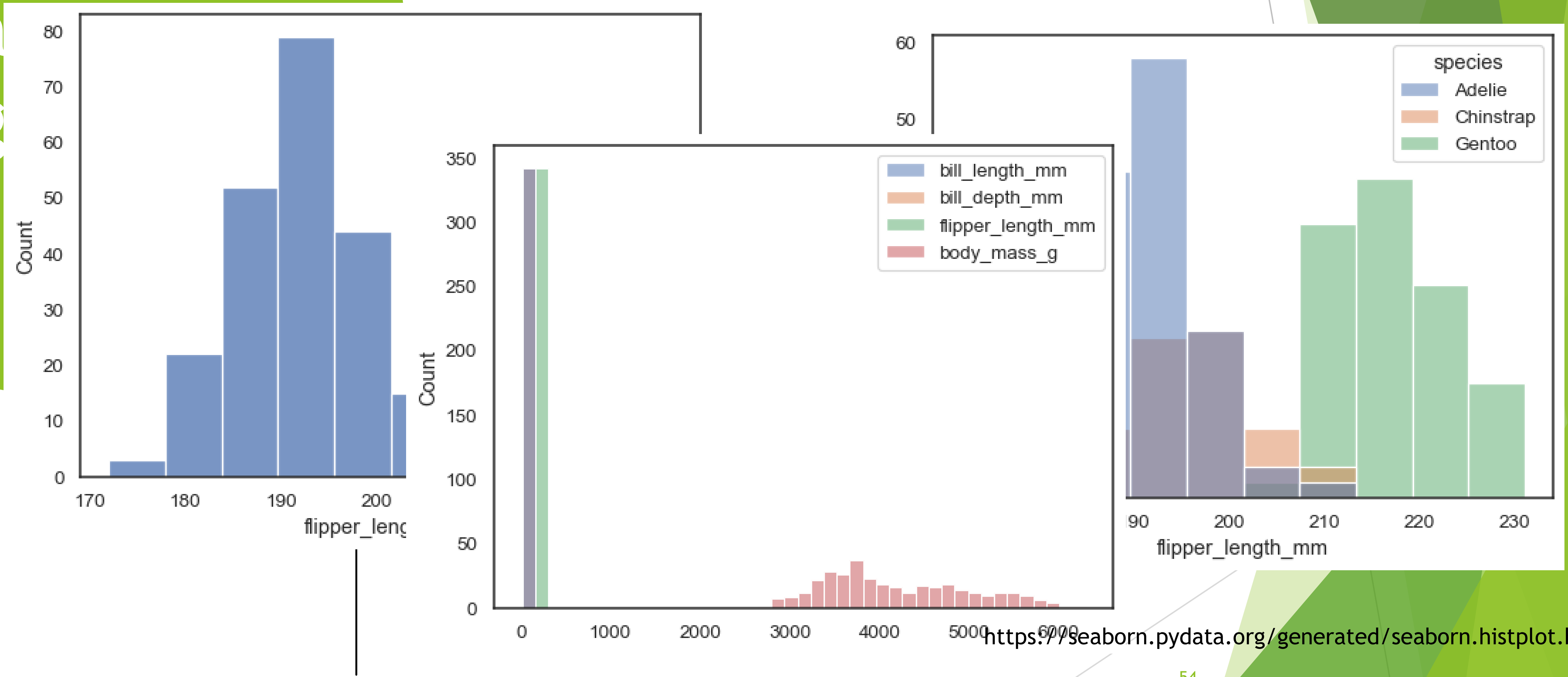
```
from sklearn.model_selection import train_test_split

# Split dataset
X = df.drop(columns=["Churn"])  # Features
y = df["Churn"]  # Target variable

# First, split into train (80%) and test (20%)
X_train, X_test, y_train, y_test =
   train_test_split(X, y, test_size=0.2)
```

Fea Eng

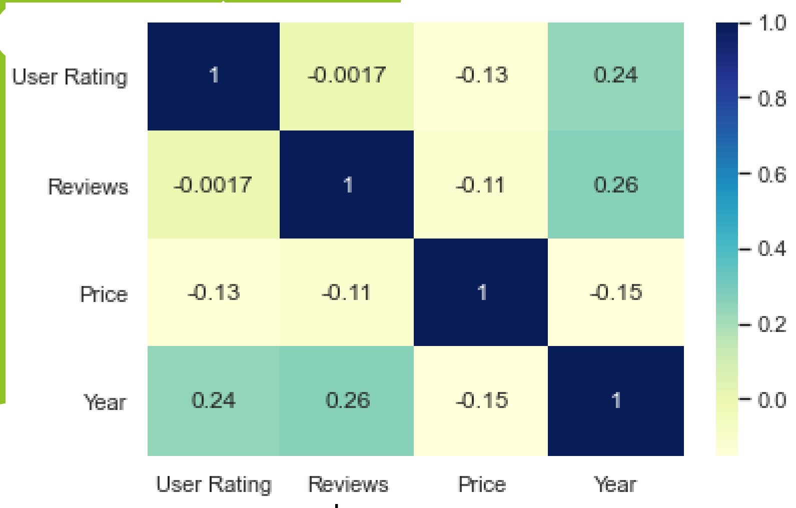
- Summary statistics (`df.describe()`)
- Correlation matrix (`df.corr()`)
- Visualizations (`histograms, scatter plots, boxplots`)

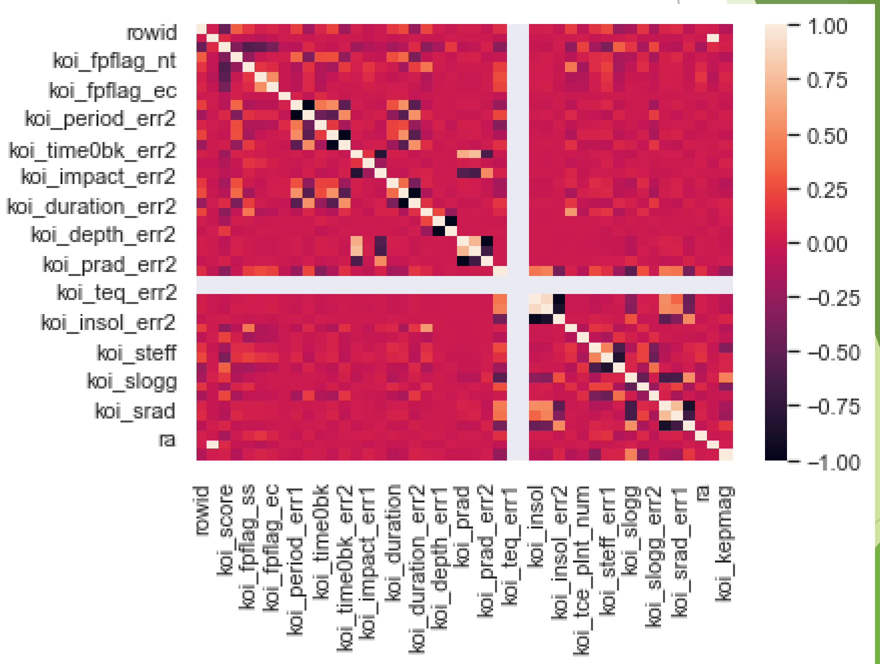


https://seaborn.pydata.org/generated/seaborn.histplot.html

Feature Enginee

- Summary statistics (`df.describe()`)
- Correlation matrix (`df.corr()`)
- Visualizations (`histograms, scatter plots, boxplots`)





https://www.geeksforgeeks.org/how-to-create-a-seaborn-correlation-heatmap-in-python/

Feature Engineering

Feature engineering is the process of **transforming raw data** into meaningful features that improve a machine learning model's performance. Scikit-Learn provides various tools for feature engineering, including **scaling, encoding, polynomial features, feature selection, and transformation**.

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False)
Encoded_data = encoder.fit_transform(df)
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)
```

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2, include_bias=False)
poly_data = poly.fit_transform(df)
```

```
…
```

Choose Model

Choose a model based on the problem type:

Classification:

- LogisticRegression,
- RandomForestClassifier,
- SVC,
- XGBoost

Regression:

- LinearRegression,
- RandomForestRegressor,
- SVR

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100)
```

- DBSCAN

Train Model

Fit the model using training data.

```
model.fit(X_train, y_train)
```

Evaluate Model

After training, evaluate performance using metrics like **accuracy, precision, recall, F1-score, and MSE**.

```
from sklearn.metrics import accuracy_score, classification_report

y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
from sklearn.metrics import mean_squared_error, r2_score

y_pred = model.predict(X_test)
print("MSE:", mean_squared_error(y_test, y_pred))
print("R² Score:", r2_score(y_test, y_pred))
```

Optimize Model

Hyperparameters affect the model's performance and can be **tuned using Grid Search**.

```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [5, 10, None]
}

grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
```

Deploy Model

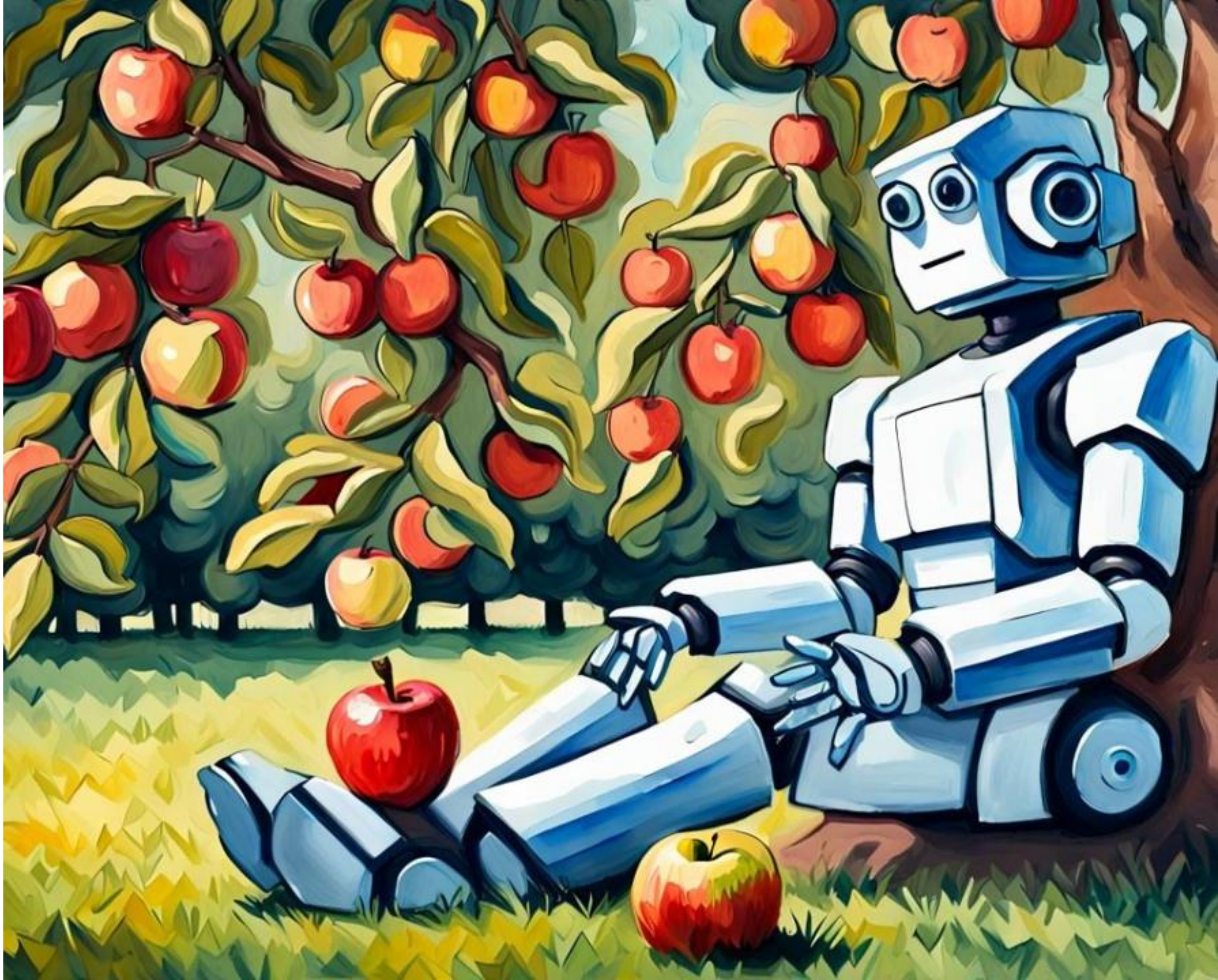
- Save the trained model

```
import pickle
with open("model.pkl", "wb") as f:
    pickle.dump(model, f)
print("Model saved successfully!")
```

- Load the saved model

```
import pickle
with open("model.pkl", "rb") as f:
    loaded_model = pickle.load(f)
# Test loaded model
y_pred = loaded_model.predict(X_test)
print("Loaded model predictions:", y_pred)
```

PyTorch



Define Problem

Before starting, clearly define the **goal** of your ML model.

Key Questions:

- What **type of problem** is this? (Classification, Regression, Clustering, etc.)
- What is the **input (features)** and **output (target variable)?**
- What will be the **evaluation metric**? (Accuracy, MSE, etc.)

Collect Data

- Data can come from **CSV files, databases, APIs, or web scraping.**
- Load Data from CSV:

```
import pandas as pd

# Load dataset
df = pd.read_csv("data.csv")
```

```
from torch.utils.data import TensorDataset, DataLoader

# Convert to PyTorch tensors
X_tensor = torch.tensor(X)
y_tensor = torch.tensor(y)

 # Create dataset and dataloader
dataset = TensorDataset(X_tensor, y_tensor)
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

DataLoader

The DataLoader in PyTorch is used to efficiently handle large datasets by batching, shuffling, and parallel loading. You should use DataLoader when:

- When Working with Large Datasets - If your dataset is too large to fit into memory, DataLoader helps by loading only small batches at a time.
  - Efficient memory management
  - Loads data dynamically (avoids RAM overload)
- When You Need Batching for Training - Most deep learning models don't train on single samples; they train on batches of data.
  - Allows parallel processing with GPUs
  - Improves training efficiency
- When You Want to Load Data in Parallel - If loading data takes time, use multiple workers to load data in parallel.
  - Speeds up training on large datasets

Coll

- Data can come from **CSV files, databases, APIs, or web scraping.**

```
import torch
import pandas as pd
from torch.utils.data import Dataset, DataLoader

# Define a custom dataset
class CustomCSVLoader(Dataset):
    def __init__(self, file_path):
        self.data = pd.read_csv(file_path)
        self.features = torch.tensor(self.data.iloc[:, :-1].values,
            dtype=torch.float32)
        self.labels = torch.tensor(self.data.iloc[:, -1].values,
            dtype=torch.long)
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]

# Load dataset
dataset = CustomCSVLoader("data.csv")
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

- Data can come from **CSV files, databases, APIs, or web scraping.**

Coll

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transforms (resize, convert to tensor, normalize)
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Load dataset from folder
dataset = datasets.ImageFolder(root="data/train", transform=transform)
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Clean Data

- Before training, the data needs to be **cleaned and prepared.**

Common Issues & Fixes:

- **Missing Values** → Fill or Drop
- **Duplicate Rows** → Remove
- **Outliers** → Detect & Handle
- **Incorrect Data Types** → Convert

```
# Fill missing values with mean
df_mean = df["Income"].mean()
df["Income"].fillna(df_mean, inplace=True)
```

Split Data

- To evaluate the model properly, the dataset should be split into **training, validation, and test sets.**

```
from sklearn.model_selection import train_test_split

# Split dataset
X = df.drop(columns=["Churn"])  # Features
y = df["Churn"]  # Target variable

# First, split into train (80%) and test (20%)
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2)
```

Feature Engineering

Feature engineering is the process of **transforming raw data** into meaningful features that improve a machine learning model's performance. Scikit-Learn provides various tools for feature engineering, including **scaling, encoding, polynomial features, feature selection, and transformation**.

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False)
Encoded_data = encoder.fit_transform(df)
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)
```

…

Feature Engineering

Convert data to PyTorch tensor

```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)

X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32).unsqueeze(1)

X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)
```

Define the NN

```
import torch
import torch.nn as nn

Class NNPredictionModel(nn.Module):
    def __init__(self, input_size):
        super(NNPredictionModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

# Initialize model
input_size = X_train.shape[1]
model = NNPredictionModel(input_size)
```

Define the NN

```
import torch
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(input_size, 16),
    nn.ReLU(),
    nn.Linear(16, 8),
    nn.ReLU(),
    nn.Linear(8, 1),
    nn.Sigmoid()
)
```

Loss Function and Optimizer

Regression

MSE

```
loss_fn = nn.MSELoss()
```

Classification

Binary Cross-Entropy

```
loss_fn = nn.BCELoss()
```

Optimizer

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Train Model

Fit the model using training data.

```
num_epochs = 50

for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X_train_tensor)
    loss = loss_fn(outputs, y_train_tensor)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

Evaluate Model

After training, evaluate performance using metrics like **accuracy, precision, recall, F1-score, and MSE**.

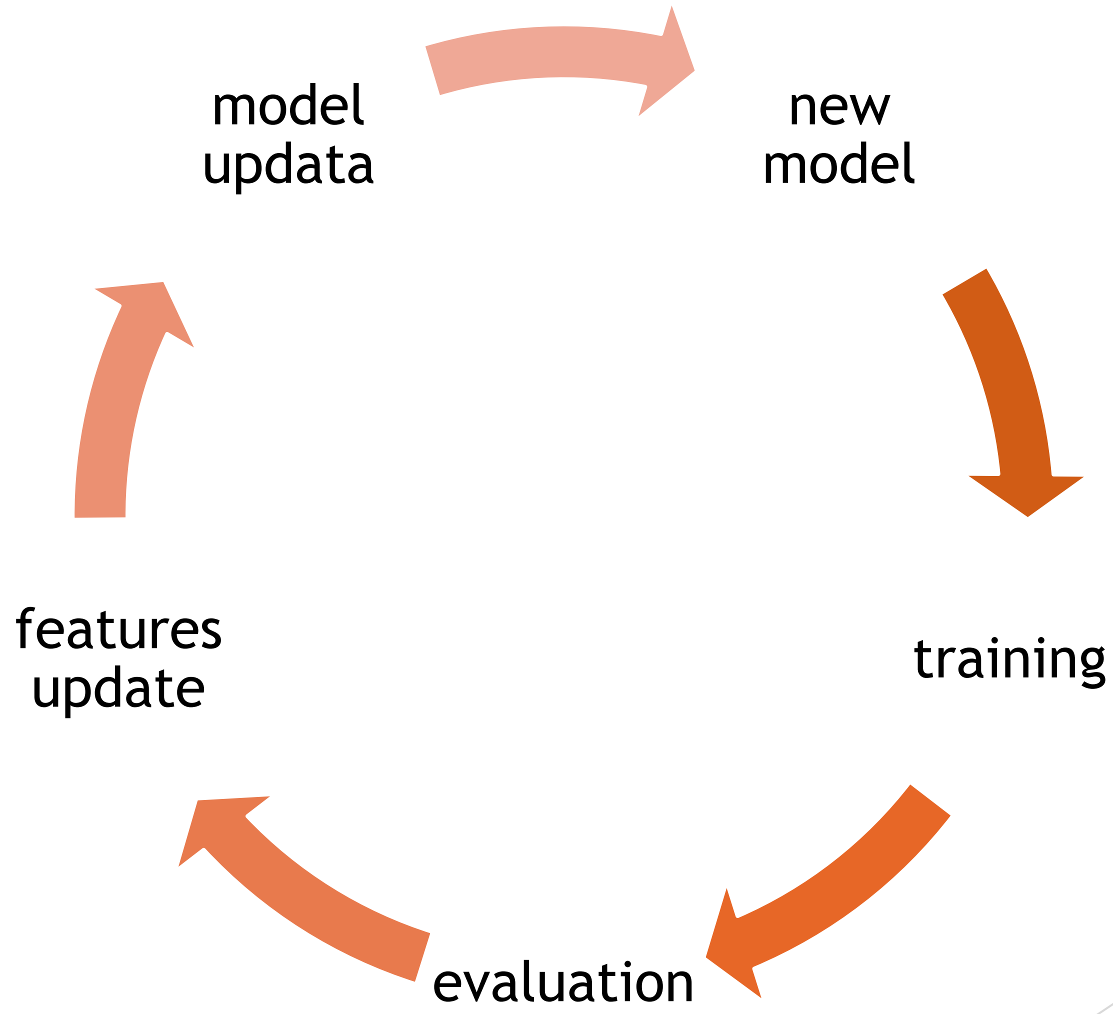
```
model.eval() # Set to evaluation mode

with torch.no_grad(): # Disable gradient calculation


from sklearn.metrics import mean_squared_error, r2_score

y_pred = model(X_test)
print("MSE:", mean_squared_error(y_test, y_pred))
print("R² Score:", r2_score(y_test, y_pred))
```

Optimize Model



Deploy Model

- Save / Load Weights

```
#sace the model
import torch # Save the model's state dictionary
torch.save(model.state_dict(), "model.pth")
#load the model
model = MyModel(input_size=10) # Ensure input size matches
model.load_state_dict(torch.load("model.pth"))
```

- Save / Load the full model

```
#save the model
torch/save(model, ”full_model.pth”)
#load the model
model = torch.load(”full_model.pth”)
```