DI04000061: Introduction Machine Learning

Computer Engineering

4th Semester

Outline

What is Learning? I

Defining Human Learning

Human learning is the cognitive process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences through experience, study, or instruction.

Defining Machine Learning (Tom Mitchell, 1997)

A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

- **Task (T):** Image classification, email spam filtering, medical diagnosis.
- **Performance (P):** Accuracy, precision, recall, mean squared error.
- **Experience (E):** Dataset of labeled examples, historical interactions.

Human Learning vs. Machine Learning Paradigm I

Human Learning

- **Input:** Sensory input, observation, reasoning, contextual experience.
- **Process:** Synaptic plasticity, intuition, abstraction, continuous lifelong learning.
- **Data Efficiency:** Learns concepts from very few examples (few-shot / one-shot).
- **Generalization:** Transfers knowledge across completely different domains effortlessly.

Machine Learning

- **Input:** Structured numerical vectors, matrices, text, images, or audio tensors.
- **Process:** Mathematical optimization, loss minimization, gradient descent.
- **Data Efficiency:** Requires large statistical samples for reliable generalization.
- **Generalization:** Domain-specific; highly vulnerable to distribution shifts.

Traditional Programming vs. Machine Learning I

Traditional Paradigm (Rule-Based Software)

Engineers write explicit rules (algorithms) that operate on input data to produce outputs.

$$\text{Data} + \text{Rules (Code)} \longrightarrow \text{Answers}$$

Machine Learning Paradigm (Data-Driven Software)

Algorithms process data alongside known answers (labels) to automatically discover the underlying rules.

$$\text{Data} + \text{Answers (Labels)} \longrightarrow \text{Rules (Model)}$$

Key Distinction

In traditional programming, logic is explicitly hardcoded. In ML, logic is inferred statistically from empirical data distributions.

Mathematical Formalization of Machine Learning I

Formal Setup

Let X denote the **Input Space** (feature vectors) and Y denote the **Output Space** (labels or target values).

- Unknown Target Function: $f : X \to Y$ (the true underlying relationship)
- Training Dataset: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ drawn i.i.d. from distribution $P(X, Y)$
- Hypothesis Set: $\mathcal{H} = \{h_1, h_2, \ldots, h_m\}$ candidate functions approximating f
- Learning Algorithm $\mathcal{A}$: Maps dataset $\mathcal{D}$ to select final hypothesis $g \in \mathcal{H}$ such that $g \approx f$

Goal of Machine Learning

The objective is not merely to memorize $\mathcal{D}$, but to find g that achieves minimal expected error on unseen samples sampled from $P(X, Y)$ (**Generalization**).

Taxonomy of Machine Learning I

Supervised

- **Data:** Labeled pairs (x, y)
- **Goal:** Predict target y given input x
- **Tasks:**
  - Classification ($y \in \{0, 1\}$)
  - Regression ($y \in \mathbb{R}$)

Unsupervised

- **Data:** Unlabeled inputs $\{x\}$
- **Goal:** Discover underlying structure or pattern
- **Tasks:**
  - Clustering
  - Dimensionality Reduction
  - Density Estimation

Reinforcement

- **Data:** State-Action-Reward tuples
- **Goal:** Learn policy $\pi(a|s)$ to maximize cumulative reward
- **Tasks:**
  - Robotics control
  - Game playing (AlphaGo)

Applications: Computer Vision & Natural Language Processing I

Computer Vision (CV)

Extracting meaningful semantic information from digital images and videos.

- **Medical Diagnostics:** Automated detection of tumors from MRI/CT scans.
- **Autonomous Mobility:** Pedestrian and lane detection for self-driving vehicles.
- **Facial Recognition:** Biometric security and authentication systems.

Natural Language Processing (NLP)

Enabling machines to understand, interpret, generate, and process human language.

- **Large Language Models (LLMs):** Contextual text generation, code synthesis, reasoning.
- **Machine Translation:** Real-time neural translation across hundreds of languages.
- **Sentiment Analysis:** Financial market analysis and customer opinion mining.

Applications: Finance, Healthcare & Smart Systems I

Financial Engineering & Fintech

- **Fraud Detection:** Real-time anomaly detection on millions of credit card transactions.
- **Algorithmic Trading:** Predictive high-frequency modeling of equity price dynamics.
- **Credit Scoring:** Assessing loan default risk using multi-source feature sets.

Healthcare & Precision Medicine

- **Drug Discovery:** Predicting molecular binding affinities to accelerate drug design.
- **Genomics:** Analyzing DNA sequencing data to identify disease markers.
- **Personalized Treatment:** Recommending tailored therapeutics based on patient profiles.

Code Example: Classical Supervised ML Workflow I

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# 1. Generate Synthetic Dataset (Features X, Labels y)
np.random.seed(42)
X = np.random.randn(200, 2)
y = (X[:, 0] + X[:, 1] > 0).astype(int)

# 2. Partition into Train and Test Sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Instantiate and Fit Hypothesis Model (g in H)
model = LogisticRegression()
model.fit(X_train, y_train)

# 4. Evaluate Performance P on Unseen Data E
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy (P): {accuracy * 100:.2f}%")
```

Summary and Core Takeaways I

Key Concepts Covered

1. **Definition of ML:** Mitchell's (T, P, E) framework formalizes machine learning.
2. **Paradigm Shift:** Moving from manually engineered rules to data-driven statistical learning.
3. **Core Goal:** Achieving robust *generalization* on unseen data, rather than overfitting or memorizing training samples.
4. **Pervasive Impact:** ML powers critical domains including healthcare, finance, CV, NLP, and autonomous navigation.

Looking Ahead: Lecture 1.2

In the next lecture, we will explore the foundational mathematical building blocks of machine learning: Linear Algebra, Multivariate Calculus, and Probability Theory.

Taxonomy of Machine Learning I

The Three Core Paradigms

Machine Learning algorithms are broadly categorized based on the nature of the learning signal or feedback available during training.

Supervised

- Labeled training data
- Predict targets (y)
- Direct feedback
- Regression & Classification

Unsupervised

- Unlabeled data
- Discover patterns
- No explicit feedback
- Clustering & Reduction

Reinforcement

- Decision agent
- Environment interaction
- Reward signals
- Policy optimization

Supervised Learning: Mathematical Formulation I

Formal Definition

Given a training dataset of N input-output pairs:

$$\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$$

where $x^{(i)} \in \mathcal{X} \subseteq \mathbb{R}^d$ represents feature vectors and $y^{(i)} \in \mathcal{Y}$ represents target labels.

Learning Goal

Learn a predictive function $h_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by θ such that $h_\theta(x)$ accurately predicts y on unseen data by minimizing empirical risk:

$$\theta^* = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(y^{(i)}, h_\theta(x^{(i)})\right) + \lambda R(\theta)$$

where $\mathcal{L}(\cdot, \cdot)$ is a loss function and $R(\theta)$ is a regularization penalty.

Supervised Learning: Classification vs. Regression I

Classification Task

- **Target Space**: Discrete label set $\mathcal{Y} = \{1, 2, \ldots, K\}$ or binary $\mathcal{Y} \in \{0, 1\}$.
- **Objective**: Learn decision boundaries separating classes.
- **Loss Function**: Categorical Cross-Entropy Loss:

$$\mathcal{L} = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

- **Examples**: Spam filtering, medical diagnosis, image recognition.

Regression Task

- **Target Space**: Continuous values $\mathcal{Y} \subseteq \mathbb{R}$.
- **Objective**: Fit continuous surface estimating quantitative values.
- **Loss Function**: Mean Squared Error (MSE):

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2$$

- **Examples**: Real estate valuation, stock forecasting, temperature prediction.

Supervised Learning: Scikit-Learn Example I

```python
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import numpy as np

# 1. Generate synthetic dataset (Features X, Labels y)
X = np.random.randn(1000, 10)
y = (X[:, 0] + X[:, 1] > 0).astype(int)

# 2. Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Instantiate and train supervised model
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)

# 4. Evaluate predictions
y_pred = model.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Unsupervised Learning: Principles & Tasks I

Formal Definition

Given an unlabeled dataset of N observations:

$$\mathcal{D} = \{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}, \quad x^{(i)} \in \mathbb{R}^d$$

The goal is to model the underlying probability density function $p(x)$ or discover intrinsic geometric structures without ground-truth targets.

Clustering

Group data into homogeneous clusters based on similarity metrics (e.g., Euclidean distance).

- K-Means Clustering
- Hierarchical Clustering
- DBSCAN

Dimensionality Reduction

Compress high-dimensional data $\mathbb{R}^d \rightarrow \mathbb{R}^k$ ($k \ll d$) while preserving variance.

- Principal Component Analysis (PCA)
- t-SNE & UMAP
- Autoencoders

Unsupervised Learning: Key Applications I

Practical Use Cases

- **Customer Segmentation**: Grouping customers by purchasing behavior for targeted marketing.
- **Anomaly Detection**: Identifying fraudulent transactions or fault conditions by detecting low-density data regions.
- **Data Visualization**: Projecting high-dimensional embeddings (e.g., word vectors) to 2D/3D space.
- **Feature Extraction**: Learning compact representations prior to downstream supervised modeling.

Key Challenge

Evaluating unsupervised models is inherently complex due to the absence of ground-truth labels. Metrics rely on internal cohesion and separation (e.g., Silhouette Score).

Unsupervised Learning: Scikit-Learn Example I

```python
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import numpy as np

# 1. Unlabeled feature matrix
X = np.random.randn(500, 20)

# 2. Dimensionality Reduction: Compress 20D to 2D
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)
print(f"Explained Variance Ratio: {pca.explained_variance_ratio_}")

# 3. Clustering: Discover 3 latent clusters
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(X_reduced)

# Output cluster centroids in 2D space
print("Cluster Centers:\n", kmeans.cluster_centers_)
```

Reinforcement Learning: Agent-Environment Interface I

Core Paradigm

An **agent** learns optimal decision-making behavior through trial-and-error interaction with a dynamic **environment**.

Markov Decision Process (MDP)

Formulated as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- **State Space** $\mathcal{S}$: Current state s_t.
- **Action Space** $\mathcal{A}$: Action a_t executed.
- **Transition Prob.**: $\mathcal{P}(s_{t+1}|s_t, a_t)$.
- **Reward Function**: $\mathcal{R}(s_t, a_t) \rightarrow r_{t+1}$.
- **Discount Factor**: $\gamma \in [0, 1)$.

Optimization Objective

Find policy $\pi(a|s)$ maximizing expected return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Value Function:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid s_t = s]$$

Reinforcement Learning: Characteristics & Applications I

The Exploration-Exploitation Dilemma

- **Exploration**: Trying novel actions to discover potential high long-term rewards.
- **Exploitation**: Leveraging current knowledge to maximize immediate expected rewards.
- Balancing this trade-off is central to RL algorithms (e.g., ϵ-greedy action selection).

Modern Applications of RL

- **Autonomous Systems**: Self-driving path planning, robotic manipulation.
- **Complex Games**: DeepMind AlphaGo, AlphaZero, OpenAI Five.
- **LLM Alignment**: Reinforcement Learning from Human Feedback (RLHF) used in state-of-the-art AI models.

Comparison of Paradigms & Hybrid Methods I

Taxonomy Comparison Matrix

Paradigm	**Data Input**	**Feedback**	**Primary Goal**
Supervised	Labeled (x, y)	Direct (Loss)	Predict y from x
Unsupervised	Unlabeled (x)	None	Discover patterns / $p(x)$
Reinforcement	Environment states	Delayed (Reward)	Maximize cumulative return

Hybrid Learning Paradigms

- **Semi-Supervised Learning**: Small set of labeled data + large volume of unlabeled data.
- **Self-Supervised Learning**: Model creates artificial labels from input data structure (e.g., Masked Language Modeling in BERT).
- **Transfer Learning**: Reusing representations learned on large datasets for domain-specific tasks.

Topic 1.6: Tools & Technology Stack for ML I

Modern Machine Learning Infrastructure

The ML software ecosystem spans data wrangling, algorithmic modeling, deep learning, and hardware acceleration.

Core Data Science Stack

- **Python**: Primary programming language for ML/DL.
- **NumPy**: Vectorized matrix computations.
- **Pandas**: High-performance tabular data structures (DataFrames).
- **Matplotlib / Seaborn**: Exploratory data visualization.

Machine Learning Stack

- **Scikit-Learn**: Industry standard for classical ML algorithms.
- **XGBoost / LightGBM**: Gradient boosted decision trees.
- **PyTorch / TensorFlow**: Deep learning framework with automatic differentiation.
- **Jupyter / Google Colab**: Interactive environments.

Scikit-Learn: Design Principles & API Pattern I

Unified Estimator API

Scikit-Learn enforces a consistent object-oriented interface across all machine learning algorithms:

- **Estimator** (`model.fit(X, y)`): Learns model parameters from data.
- **Transformer** (`transformer.transform(X)`): Preprocesses or scales features (e.g., `StandardScaler`).
- **Predictor** (`model.predict(X)`): Generates predictions on new feature arrays.

Design Advantage

Allows seamless pipeline construction (`Pipeline([scaler, model])`), preventing data leakage between training and validation folds.

Deep Learning Ecosystem: PyTorch vs. TensorFlow I

PyTorch (Meta AI)

- **Dynamic Computational Graph** (Eager execution by default).
- Pythonic syntax, intuitive debugging.
- Dominant framework in modern academic research and LLM ecosystem (Hugging Face).
- Native GPU tensor operations via CUDA.

TensorFlow / Keras (Google)

- **Static / Hybrid Graph execution** via `tf.function`.
- Robust production deployment ecosystem (TF Serving, TF Lite, TF.js).
- High-level Keras API for rapid prototyping.
- Strong enterprise production track record.

Complete Python ML Pipeline Example I

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

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

# 2. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

# 3. Create standardized ML pipeline
pipeline = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=200)
)

# 4. Train pipeline & Evaluate
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

Summary & Key Takeaways I

Lecture 1.2 Summary

- **Supervised Learning**: Requires labeled pairs (x, y) for classification and regression tasks.
- **Unsupervised Learning**: Extracts intrinsic representations or clusters from unlabeled data x.
- **Reinforcement Learning**: Optimizes decision-making policy via agent-environment reward interaction.
- **Tooling Stack**: Python ecosystem provides unified tools: NumPy/Pandas for data, Scikit-Learn for classical models, and PyTorch/TensorFlow for deep learning.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 1**: Introduction to Machine Learning
- **Topic**: 1.3 Benefits of Machine Learning

Learning Objectives:

1. Differentiate between traditional algorithmic programming and data-driven learning paradigms.
2. Identify key advantages of Machine Learning in handling complex, high-dimensional, and non-linear patterns.
3. Understand how adaptability and continuous model updates drive business and technical value.
4. Analyze real-world applications demonstrating ML efficiency over rule-based systems.

Paradigm Shift: Traditional Programming vs. Machine Learning I

Traditional Software Engineering

- **Inputs**: Explicit Rules (Code) + Data.
- **Output**: Answers / Decisions.
- **Limitation**: Requires humans to manually program every rule, edge case, and logical branch.
- **Brittle**: Fails when problem complexity scales or data patterns shift.

Machine Learning Paradigm

- **Inputs**: Historical Data + Answers (Targets).
- **Output**: Learned Model / Rules ($f : X \rightarrow y$).
- **Advantage**: Algorithmic discovery of complex relationships directly from empirical data.
- **Scalable**: Automatically updates with new data streams.

1. Solving Problems Beyond Human Rule Crafting I

The Limits of Hard-Coded Logic

For perception and cognitive tasks (e.g., Computer Vision, Natural Language Processing), writing explicit rules is intractable:

$$\text{Rules for recognizing a cat} \implies \text{Millions of pixel combinations, lighting, angles, breeds.}$$

- **Automatic Feature Discovery**: Deep Neural Networks extract hierarchical features (edges $\rightarrow$ textures $\rightarrow$ object parts) without manual feature engineering.
- **Handling Non-Linearity**: ML models approximate complex non-linear functions $y = f(\mathbf{x})$ over arbitrary vector spaces.
- **High Dimensionality**: Operates effectively in high-dimensional feature spaces ($\mathbf{x} \in \mathbb{R}^d$ where $d \gg 10^6$).

Code Comparison: Rule-Based vs. Machine Learning I

Spam Detection: Hardcoded Rules vs Naive Bayes

Traditional rule-based systems decay quickly as spammers change tactics; ML adapts effortlessly.

```python
# —— Traditional Rule-Based Approach ——
def is_spam_rules(email_text):
    keywords = ["BUY-NOW", "FREE-MONEY", "CLICK-HERE", "WINNER"]
    score = sum(1 for kw in keywords if kw in email_text.upper())
    return score >= 2  # Brittle: misses subtle spam, flags safe mail

# —— Machine Learning Approach ——
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB

# ML learns statistical word associations directly from data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(corpus_emails)
clf = MultinomialNB()
clf.fit(X_train, y_labels) # Automatically generalizes to unseen text
```

2. Adaptability and Continuous Improvement I

Dynamic Environment Adaptation

Traditional software requires manual code refactoring and redeployment when system environment parameters change. Machine learning models continuously adapt through retraining.

- **Online & Incremental Learning**: Updating weights $\theta^{(t+1)} = \theta^{(t)} - \eta \nabla \mathcal{L}(\theta^{(t)})$ as new streaming data arrives.
- **Concept Drift Handling**: Detecting statistical shifts in target distribution $P(y|\mathbf{x})$ over time and automatically retraining models.
- **Feedback Loops**: Reinforcement Learning agents continuously improve policy $\pi(a|s)$ via environmental rewards.

3. Scalability, Efficiency, and Hyper-Personalization I

Scalability & Automation

- Automated decision systems process millions of queries per second (e.g., credit scoring, fraud detection).
- Reduces operational overhead by replacing manual inspection with high-throughput inference engines.

Hyper-Personalization

- Recommendation systems (Collaborative Filtering & Matrix Factorization) deliver personalized content feeds to billions of users.
- Optimization:
  $\min_{P,Q} \sum_{(u,i)\in R}(R_{ui} - \mathbf{p}_u^T \mathbf{q}_i)^2 + \lambda(\|\mathbf{p}_u\|^2 + \|\mathbf{q}_i\|^2)$.

Practical Python Example: Retraining ML Model on Stream Data I

```python
import numpy as np
from sklearn.linear_model import SGDClassifier

# Initialize incremental SGD model (Online Learning)
model = SGDClassifier(loss='log_loss', random_state=42)

# Simulated streaming batches of user interaction data
classes = np.array([0, 1])  # 0: No Click, 1: Click
for batch_idx in range(5):
    X_batch = np.random.randn(100, 10) # 100 incoming requests
    y_batch = np.random.randint(0, 2, size=100)

    # Partial fit updates model parameters without full dataset reload
    model.partial_fit(X_batch, y_batch, classes=classes)
    print(f"Batch {batch_idx+1} processed. Model updated.")
```

4. Key Benefits Across High-Impact Domains I

- **Healthcare & Diagnostics**:
  - Automated radiology scan classification (CNNs achieving expert level sensitivity).
  - Drug discovery: Predicting protein folding structures (AlphaFold).
- **Finance & Risk Management**:
  - Real-time anomaly detection in transaction processing streams.
  - Algorithmic trading and automated credit scoring systems.
- **Autonomous Systems**:
  - Sensor fusion (LiDAR, Radar, Camera) for self-driving vehicles.
  - Robotics trajectory planning and obstacle avoidance.

Quantitative Comparison: Rule-Based vs. Machine Learning I

Dimension	Rule-Based Systems	Machine Learning
Development	High manual logic coding	Data curation & model tuning
Maintainability	Degrades with complexity	High; retrain on new data
Pattern Complexity	Linear & simple logic	High non-linear interactions
Execution Speed	Instant deterministic rules	Inference latency ($\sim$ms)
Generalization	Poor (fails on edge cases)	High generalization capability

Table: Comparison Matrix of Software Engineering Paradigms

Summary & Key Takeaways I

Core Takeaway

Machine Learning transforms software engineering from *explicit instruction writing* to *statistical estimation from data*, enabling systems to solve previously intractable perception, prediction, and automation tasks.

- **Primary Benefits**:
  1. **Generalization**: Capability to make accurate predictions on unseen data.
  2. **Automation**: Replaces complex human heuristics with data-driven models.
  3. **Adaptability**: Seamlessly adjusts to shifting environments via model updates.
  4. **Scalability**: Processes massive multi-dimensional datasets beyond human comprehension.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 1**: Introduction to Machine Learning
- **Topic**: Lecture 1.4: Challenges of Machine Learning

Learning Objectives:

1. Identify primary data-centric challenges including data scarcity, noise, non-representative sampling, and high dimensionality.
2. Understand model-centric challenges such as overfitting, underfitting, and the bias-variance tradeoff.
3. Analyze deployment and operational challenges including data drift, concept drift, and model interpretability.
4. Implement Python code to simulate and mitigate key challenges in machine learning workflows.

Taxonomy of Machine Learning Challenges I

The Three Pillars of ML Obstacles

Developing effective machine learning systems requires overcoming challenges spanning data quality, algorithm design, and real-world operational deployment.

1. Data Challenges

- Insufficient data quantity
- Poor data quality & noise
- Non-representative data
- Irrelevant features

2. Model Challenges

- Overfitting vs. underfitting
- Hyperparameter tuning
- High dimensionality
- Computational complexity

3. Operational Challenges

- Data & concept drift
- Black-box opacity (XAI)
- Deployment latency
- Fairness & bias

Data Quality and Quantity Issues I

Data-Centric Bottlenecks

Machine learning algorithms are fundamentally driven by data: *"Garbage in, garbage out."*

- **Insufficient Data Quantity**:
  - Complex models (e.g., Deep Networks) require 10^5–10^8 samples to generalize well.
  - Data acquisition can be prohibitively expensive or limited by rare domain occurrences.
- **Non-Representative Training Data**:
  - Sampling Bias: Training distribution $P_{\text{train}}(X)$ differs from deployment distribution $P_{\text{test}}(X)$.
  - Results in poor generalization on out-of-distribution real-world instances.
- **Poor Data Quality (Noise & Outliers)**:
  - Sensor errors, corrupted logs, and missing values degrade boundary estimation and model stability.

Python Example: Impact of Noise and Outliers I

Simulating Corrupted Target Data

Demonstration of how random label noise degrades linear regression boundary estimations.

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Generate synthetic clean dataset
np.random.seed(42)
X = np.linspace(0, 10, 50).reshape(-1, 1)
y_clean = 2.5 * X.squeeze() + 5.0

# Add high-magnitude noise/outliers to 10% of labels
y_noisy = y_clean.copy()
noise_idx = np.random.choice(50, size=5, replace=False)
y_noisy[noise_idx] += np.random.normal(0, 50, size=5)

# Fit models on clean vs noisy data
model_clean = LinearRegression().fit(X, y_clean)
model_noisy = LinearRegression().fit(X, y_noisy)

print(f"Clean Slope: {model_clean.coef_[0]:.2f}, Intercept: {model_clean.intercept_:.2f}")
print(f"Noisy Slope: {model_noisy.coef_[0]:.2f}, Intercept: {model_noisy.intercept_:.2f}")
```

Model Generalization: Overfitting vs. Underfitting I

Fundamental Tradeoff

The ultimate goal of machine learning is to minimize expected generalization error (risk):

$$R(f) = \mathbb{E}_{(X,Y)\sim P}[L(f(X), Y)]$$

- **Underfitting (High Bias)**:
  - Model capacity is insufficient to capture underlying relationships.
  - Characterized by high training error AND high test error.
- **Overfitting (High Variance)**:
  - Model memorizes training samples including idiosyncratic noise.
  - Characterized by low training error, but high test error.
- **Mitigation Strategies**:
  - Regularization (L_1, L_2), Cross-Validation, Early Stopping, Pruning.

Python Example: Polynomial Overfitting I

Demonstrating Model Capacity vs. Overfitting

Fitting high-degree polynomials leads to severe variance on unseen test points.

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

X_train = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y_train = np.array([2.1, 3.9, 6.2, 8.1, 10.2])
X_test  = np.array([2.5, 4.5]).reshape(-1, 1)
y_test  = np.array([5.0, 9.1])

# High degree polynomial (Degree 4) overfits 5 points exactly
poly = PolynomialFeatures(degree=4)
X_train_poly = poly.fit_transform(X_train)
X_test_poly  = poly.transform(X_test)

model = LinearRegression().fit(X_train_poly, y_train)
train_rmse = np.sqrt(mean_squared_error(y_train, model.predict(X_train_poly)))
test_rmse  = np.sqrt(mean_squared_error(y_test, model.predict(X_test_poly)))

print(f"Train RMSE: {train_rmse:.4f} | Test RMSE: {test_rmse:.4f}")
```

The Curse of Dimensionality I

High-Dimensional Geometric Paradoxes

As feature dimension d increases, the volume of space grows exponentially, making data points extremely sparse.

- **Data Sparsity**: To maintain uniform sample density, required sample size N grows exponentially: $N \propto e^d$.
- **Distance Concentration**: In high-dimensional spaces, pairwise Euclidean distances become nearly uniform:

$$\lim_{d \to \infty} \frac{d_{\max} - d_{\min}}{d_{\min}} = 0$$

  Distances lose discriminatory power for algorithms like k-NN, K-Means, and SVMs.

- **Solutions**:
  - Dimensionality Reduction: PCA, t-SNE, UMAP, Autoencoders.
  - Feature Selection: Variance thresholding, mutual information, L_1 Lasso.

Python Example: Distance Concentration Effect I

Measuring Distance Range Across Dimensions

Observing how relative distance contrast collapses as feature dimension d grows.

```
import numpy as np
from scipy.spatial.distance import pdist

def distance_contrast(dim, n_samples=500):
    # Sample uniform random vectors in hypercube [0, 1]^dim
    X = np.random.uniform(0, 1, size=(n_samples, dim))
    distances = pdist(X, metric='euclidean')
    d_min, d_max = np.min(distances), np.max(distances)
    return (d_max - d_min) / d_min

for d in [2, 10, 100, 1000]:
    contrast = distance_contrast(d)
    print(f"Dimension {d:4d} --> Relative Distance Contrast: {contrast:.4f}")
```

Production Challenges: Distribution Drift I

Dynamic Real-World Environments

Models assume stationary distributions $P(X, Y)$. Real-world data distributions evolve over time.

- **Covariate Shift (Data Drift)**:
  - Input distribution changes: $P_{t_1}(X) \neq P_{t_2}(X)$, while relationship $P(Y|X)$ remains constant.
  - Example: User demographic shift in an e-commerce platform over time.
- **Concept Drift**:
  - Target relationship changes: $P_{t_1}(Y|X) \neq P_{t_2}(Y|X)$.
  - Example: Financial fraud patterns evolving post-security updates.
- **Prior Probability Shift**:
  - Label distribution changes: $P_{t_1}(Y) \neq P_{t_2}(Y)$.

Model Interpretability vs. Performance I

The Accuracy-Explainability Spectrum

Tradeoff between high predictive accuracy and human-understandable decision rules.

- **White-Box Models** (High Interpretability, Lower Capacity):
  - Linear/Logistic Regression, Decision Trees, Naive Bayes.
  - Explicit coefficients / rules allow transparent domain auditing.
- **Black-Box Models** (Low Interpretability, High Capacity):
  - Deep Neural Networks, Gradient Boosted Trees, Random Forests.
  - High capacity for complex non-linear structures, but decision paths are opaque.
- **Explainable AI (XAI) Frameworks**:
  - Local explanation models: SHAP (Shapley Additive exPlanations), LIME.

Summary: Overcoming ML Challenges I

Challenge	**Root Cause**	**Standard Solutions**
Data Scarcity	High labeling costs	Data Augmentation, Transfer Learning
Noise / Outliers	Sensor errors, corrupted data	Robust Scalers, Isolation Forests, Cleaning
Overfitting	Model over-capacity	L_1/L_2 Regularization, Cross-Validation
Curse of Dim.	Sparse feature space	PCA, Feature Selection, Autoencoders
Data/Concept Drift	Non-stationary distributions	Continuous Monitoring, MLOps Retraining
Black-Box Opacity	Complex parameters	SHAP, LIME, Feature Importance Plots

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 2**: Python Libraries for Machine Learning
- **Topic**: Lecture 2.1: NumPy — Key Concepts and Features

Learning Objectives:

1. Understand the fundamental role of NumPy as the foundation for scientific computing and Machine Learning in Python.
2. Analyze the internal architecture of the `ndarray` object, including memory layout, strides, and data types (`dtype`).
3. Contrast NumPy vectorized operations with standard Python loops in terms of performance and memory efficiency.
4. Master the mechanics and formal rules of NumPy Broadcasting for multi-dimensional operations.
5. Differentiate between memory views and deep copies to avoid subtle bug mutations in data processing.

Introduction to NumPy in Machine Learning I

What is NumPy?

NumPy (**Num**erical **Py**thon) is the core library for scientific computing in Python. It provides a high-performance N-dimensional array object (`ndarray`) and tools for working with these arrays.

- **Ecosystem Foundation**: Serves as the numerical backbone for SciPy, Pandas, Scikit-Learn, PyTorch, and TensorFlow.
- **Machine Learning Data Representation**:
  - **Feature Matrix (X)**: 2D array of shape (N, d) where N is samples and d is features.
  - **Target Vector (y)**: 1D array of shape $(N,)$ containing ground-truth labels.
  - **Model Parameters ($\mathbf{w}, b$)**: Weights and bias vectors updated via matrix algebra.
- **Performance Engine**: Core routines written in C/Fortran for low-level execution speed.

Why Python Standard Lists are Inefficient for ML

Python lists store pointers to objects scattered across non-contiguous memory locations. Each element lookup incurs dynamic type checking and pointer dereferencing overhead.

- **Contiguous Memory Allocation**: NumPy arrays allocate a single, contiguous block of memory for elements of uniform data type (`dtype`).
- **Cache Locality**: Contiguous storage maximizes CPU L1/L2 cache hit rates during sequential iteration.
- **SIMD Hardware Acceleration**: Enables Single Instruction, Multiple Data (SIMD) processor instructions for parallel numerical execution.

Feature	**Python List**	**NumPy** `ndarray`
Data Types	Heterogeneous	Homogeneous (`dtype`)
Memory Layout	Non-contiguous (Pointers)	Contiguous Block
Execution	Interpreted Loop	Compiled C / SIMD
Element Operations	Requires explicit loops	Vectorized Ufuncs

Object

Key Attributes of `ndarray`

An `ndarray` is a multidimensional container of items of the same type and size, defined by a small set of metadata.

- `shape`: Tuple of integers indicating the size of the array along each dimension (e.g., `(500, 10)`).
- `ndim`: Total number of dimensions (axes) in the array.
- `dtype`: Data type object describing the layout of bytes (e.g., `float64`, `int32`).
- `size`: Total number of elements, equal to the product of `shape` elements.
- `itemsize` & `nbytes`: Byte size of each element and total array memory usage (size $\times$ itemsize).
- `strides`: Tuple of bytes to step in each dimension when traversing the array in memory.

Python Code: Creating Arrays & Inspecting Attributes I

Demonstration of Array Attributes

Creating arrays and examining internal structural properties.

```
import numpy as np

# Create a 2D feature matrix (3 samples, 4 features)
X = np.array([[1.5, 2.3, 0.0, 4.1],
              [3.1, 0.5, 2.2, 1.9],
              [0.8, 4.0, 1.1, 3.5]], dtype=np.float64)

print("Shape:---", X.shape)      # Output: (3, 4)
print("Dimensions:", X.ndim)     # Output: 2
print("Data-Type:-", X.dtype)    # Output: float64
print("Item-Size:-", X.itemsize)# Output: 8 bytes
print("Total-Bytes:", X.nbytes)  # Output: 3 * 4 * 8 = 96 bytes
print("Strides:--", X.strides)   # Output: (32, 8)
```

Key Concept: Vectorization & Universal Functions I

Vectorization

The process of performing operations on whole arrays simultaneously without explicit Python `for` loops.

- **Universal Functions** (`ufuncs`): Functions that operate element-wise on `ndarray` objects, implemented as fast compiled C loops.
- **Mathematical Expressiveness**: Allows writing mathematical formulas cleanly, matching linear algebra notation.

Example: Logistic Regression Activation (Sigmoid)

Mathematical equation applied vectorially to a logit vector $\mathbf{z} \in \mathbb{R}^N$:

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}}$$

In NumPy, this executes across all N elements simultaneously without a single Python loop.

Python Code: Vectorization vs. Python Loops I

Performance Benchmark

Comparing element-wise computation speed between Python lists and NumPy arrays.

```
import numpy as np
import time

size = 1_000_000
# Pure Python Loop
list_a, list_b = list(range(size)), list(range(size))
t0 = time.time()
py_result = [a + b for a, b in zip(list_a, list_b)]
t_py = time.time() - t0

# NumPy Vectorized Addition
arr_a, arr_b = np.arange(size), np.arange(size)
t0 = time.time()
np_result = arr_a + arr_b
t_np = time.time() - t0

print(f"Python Loop Time: {t_py:.4f} sec")
print(f"NumPy Vector Time: {t_np:.4f} sec")
print(f"Speedup Factor: ~~~~{t_py / t_np:.1f}x")
```

Key Concept: Broadcasting Rules I

Definition

Broadcasting describes how NumPy treats arrays of different shapes during arithmetic operations. It expands the smaller array to match the larger array without making unnecessary data copies.

Formal Broadcasting Rules

To determine if two arrays are compatible for broadcasting:

1. Compare their shapes element-wise, starting from the **trailing (rightmost) dimensions** and moving left.
2. Two dimensions are compatible if:
   - They are **equal**, OR
   - One of them is **1**.
3. If neither condition is met, NumPy raises a `ValueError: operands could not be broadcast together`.

Python Code: Broadcasting in Machine Learning I

Practical Example: Feature Normalization (Mean Subtraction)

Subtracting feature mean vector $\boldsymbol{\mu} \in \mathbb{R}^d$ from data matrix $X \in \mathbb{R}^{N \times d}$.

```
import numpy as np

# Data matrix X: 3 samples, 2 features (Shape: 3, 2)
X = np.array([[100.0, 2.0],
              [200.0, 4.0],
              [300.0, 6.0]])

# Feature-wise mean vector (Shape: 2,)
mu = np.mean(X, axis=0)  # Output shape: (2,) -> [200.0, 4.0]

# Broadcasting subtracts (2,) from (3, 2) by expanding axis 0
X_centered = X - mu
print("Centered Matrix:\n", X_centered)
# Output:
# [[-100.   -2.]
#  [   0.    0.]
#  [ 100.    2.]]
```

Memory Management: Views vs. Deep Copies I

Views vs. Copies

- **View**: A new array object pointing to the **same memory buffer**. Modifying data in a view mutates the original array!
- **Copy**: A complete duplication of data into a new memory location.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Basic slicing creates a VIEW
sub_view = arr[1:4]
sub_view[0] = 999
print("Original Array modified:", arr)
# Output: [ 10 999  30  40  50]

# Explicit copy allocates NEW memory
arr_copy = arr.copy()
arr_copy[0] = -1
print("Original Array untouched:", arr[0]) # Output: 10
```

Summary: NumPy Best Practices for ML I

- **Eliminate Python Loops**: Replace iterative code with vectorized expressions and built-in `ufuncs` for order-of-magnitude speedups.
- **Master Shape Mechanics**: Always check `X.shape` and ensure broadcasting dimensions align before performing element-wise operations.
- **Optimize Data Types**: Choose minimal necessary precision (`float32` vs `float64`) to reduce memory usage and accelerate GPU transfers.
- **Beware of Unintended Side Effects**: Use `.copy()` when extracting sub-matrices to avoid implicit mutation of training datasets.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 2**: Python Libraries for Machine Learning
- **Topic**: 2.1.1 Creating and Accessing Array — 2.4.2 Functions

Learning Objectives:

1. Master fundamental NumPy array creation techniques: `array()`, `zeros()`, `ones()`, and `arange()`.
2. Understand array dimensionality, memory layout, and efficient reshaping using `reshape()`.
3. Utilize Scikit-Learn's built-in dataset loaders (`load_*()`) to import benchmark datasets.
4. Implement reproducible dataset partitioning using `train_test_split()` with stratification.
5. Construct end-to-end data preprocessing pipelines for machine learning models.

Why NumPy for Machine Learning?

The core data structure of NumPy is `ndarray` (N-dimensional array), providing contiguous memory storage and C-optimized vectorized computation essential for linear algebra operations in ML models.

- **Creation from Python Sequences**: Convert lists or tuples into homogeneous numerical vectors or matrices.
- **Key Structural Attributes**:
  - `shape`: Tuple representing dimensions along each axis.
  - `ndim`: Number of array dimensions (axes).
  - `dtype`: Uniform data type of array elements (e.g., `float64`, `int32`).

```
import numpy as np

# 1D Array (Target Vector y)
y_arr = np.array([0, 1, 0, 1, 1])
print(f"y shape: {y_arr.shape}, ndim: {y_arr.ndim}, dtype: {y_arr.dtype}")

# 2D Array (Feature Matrix X: 3 samples, 2 features)
X_arr = np.array([[1.5, 2.3],
                  [3.1, 4.0],
                  [0.8, 1.2]])
print(f"X shape: {X_arr.shape}, ndim: {X_arr.ndim}")
```

and `np.ones()`

Motivation in Machine Learning

Pre-allocating arrays filled with zeros or ones is crucial for initializing model weights, bias vectors, gradient placeholders, and target masks.

- `np.zeros(shape, dtype=float)`: Returns a new array of given shape filled with 0.0.
- `np.ones(shape, dtype=float)`: Returns a new array of given shape filled with 1.0.

```
import numpy as np

# Initializing bias vector b in R^5 with zeros
bias = np.zeros(5, dtype=np.float32)
print("Bias Vector:", bias)

# Initializing a 3x4 matrix of ones (e.g., dummy feature matrix)
ones_matrix = np.ones((3, 4), dtype=np.float64)
print("Ones Matrix Shape:", ones_matrix.shape)

# Adding an intercept column (bias feature) filled with 1s
N_samples = 4
intercept_col = np.ones((N_samples, 1))
print("Intercept Column:\n", intercept_col)
```

Definition

`np.arange([start,] stop[, step,], dtype=None)` creates 1D arrays with evenly spaced values within the half-open interval [*start*, *stop*).

- **Parameters**:
  - `start`: Start of interval (inclusive, default is 0).
  - `stop`: End of interval (exclusive).
  - `step`: Spacing between values.
- **Use Cases**: Generating sample indices, synthetic 1D feature grids, and time-step sequences.

```
import numpy as np

# Integer sequence from 0 to 9
indices = np.arange(10)
print("Indices:", indices)

# Feature grid with step size 0.5 from -2 to 2 (stop is exclusive)
grid = np.arange(-2.0, 2.5, 0.5)
print("Feature Grid:", grid)

# Even numbers sequence
evens = np.arange(0, 10, 2)
print("Evens:", evens)
```

Dimension Transformation

`np.reshape(a, newshape)` gives a new shape to an array without changing its underlying data memory. The total element count $N = \prod \text{shape\_dim}_i$ must remain invariant.

- **Inferring Dimensions (-1)**: One dimension parameter can be -1, allowing NumPy to automatically compute the required length.
- **Scikit-Learn Formatting**: Converting 1D arrays of shape $(N,)$ to 2D matrices of shape $(N, 1)$ via `reshape(-1, 1)`.

```
import numpy as np

# 1D array of 12 elements
a = np.arange(12) # shape: (12,)

# Reshape into 2D matrix (3 samples, 4 features)
X_2d = a.reshape(3, 4)
print("Reshaped (3,4):\n", X_2d)
```

```
# Reshape 1D vector to 2D Column Vector (N, 1) for Scikit-Learn
y_col = a.reshape(-1, 1)
print(f"Original shape: {a.shape} -> Column Vector shape: {y_col.shape}")
```

Functions

Built-in Benchmark Datasets

`sklearn.datasets` provides toy datasets pre-packaged for quick algorithm experimentation and prototyping. Common loaders include `load_iris()`, `load_digits()`, and `load_breast_cancer()`.

- `Bunch` **Object**: Returns a dictionary-like structure containing:
  - `data`: Feature matrix $X \in \mathbb{R}^{N \times d}$.
  - `target`: Target vector $y \in \mathbb{R}^{N}$.
  - `feature_names`, `target_names`, `DESCR`: Metadata descriptions.
- **Direct Extraction**: Setting `return_X_y=True` returns (X, y) directly as NumPy arrays.

```
from sklearn.datasets import load_iris, load_breast_cancer

# Load Iris dataset as (X, y) tuple directly
X_iris, y_iris = load_iris(return_X_y=True)
print(f"Iris X shape: {X_iris.shape}, y shape: {y_iris.shape}")

# Load Breast Cancer dataset object to inspect metadata
cancer_data = load_breast_cancer()
print("Features:", cancer_data.feature_names[:3])
print("Target Classes:", cancer_data.target_names)
```

Purpose in Supervised Learning

To measure generalization performance and detect overfitting, datasets are partitioned into non-overlapping training and testing subsets: $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$.

- **Key Arguments in** `sklearn.model_selection.train_test_split`:
  - `*arrays`: Arrays (X, y) with matching first dimension length N.
  - `test_size`: Proportion of dataset for test split (e.g., 0.2 for 20%).
  - `random_state`: Integer seed ensuring reproducible pseudo-random shuffling.
  - `shuffle`: Boolean flag (default `True`).

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

X, y = load_iris(return_X_y=True)

# Split dataset: 80% Training, 20% Testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(f"Original X: {X.shape}")
print(f"Train X: {X_train.shape}, Test X: {X_test.shape}")
```

Advanced Splitting: Stratification I

The Imbalanced Dataset Challenge

Random splitting on imbalanced or multi-class datasets can produce class distribution mismatches between training and test sets, biasing model evaluation.

- **Stratified Sampling (`stratify=y`)**: Forces the split to maintain the exact target class proportions in both training and testing partitions.

```
import numpy as np
from sklearn.model_selection import train_test_split

# Imbalanced target vector: 90% Class 0, 10% Class 1
X_dummy = np.arange(100).reshape(100, 1)
y_dummy = np.array([0]*90 + [1]*10)

# Stratified split to preserve 9:1 class ratio
X_tr, X_te, y_tr, y_te = train_test_split(
    X_dummy, y_dummy, test_size=0.2, random_state=42, stratify=y_dummy
)

print("Train Class 1 Count:", np.sum(y_tr == 1)) # Exactly 8
print("Test Class 1 Count:", np.sum(y_te == 1)) # Exactly 2
```

End-to-End ML Data Preparation Pipeline I

Workflow Integration

Combining array creation, shape inspection, dataset loading, reshaping, and splitting into a unified, clean machine learning ingestion pipeline.

```python
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

# 1. Load benchmark digits dataset
digits = load_digits()
X, y = digits.data, digits.target

# 2. Inspect original dimensions
print(f"Raw Feature Matrix: {X.shape}") # (1797, 64)

# 3. Partition into Train (70%), Val (15%), Test (15%)
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp
)

print(f"Train: {X_train.shape} | Val: {X_val.shape} | Test: {X_test.shape}")
```

Summary of Functions & Best Practices I

Function	Library / Module	Primary Purpose in ML
`np.array()`	`numpy`	Convert sequences to N-dimensional arrays
`np.zeros()` / `ones()`	`numpy`	Pre-allocate memory for weights & masks
`np.arange()`	`numpy`	Generate numerical ranges & grid indices
`np.reshape()`	`numpy`	Re-dimension arrays (e.g. 1D to 2D column vector)
`load_*()`	`sklearn.datasets`	Load benchmark datasets (X, y)
`train_test_split()`	`sklearn.model_selection`	Partition data into train/test subsets

Key Best Practice

Always set `random_state` when calling `train_test_split()` to ensure reproducible model training and evaluation across experiments!

Overview of Array Stacking & Splitting I

Role in Machine Learning Workflows

In machine learning, data often comes from disparate sources or requires transformation during preprocessing. Stacking and splitting operations are critical for feature assembly, mini-batch generation, dataset concatenation, and train-validation-test partitioning.

- **Array Stacking**: Joining existing arrays together along an existing or newly created dimension.
  - `np.vstack()`: Stack arrays vertically (row-wise, axis 0).
  - `np.hstack()`: Stack arrays horizontally (column-wise, axis 1).
  - `np.dstack()`: Stack arrays depth-wise along third axis (axis 2).
  - `np.stack()`: Join arrays along a new dimension.
- **Array Splitting**: Dividing a single array into multiple sub-arrays along specified axes.
  - `np.split()`: Flexible splitting by equal parts or index locations.
  - `np.vsplit()` / `np.hsplit()`: Shortcut functions for row and column splitting.

Vertical and Horizontal Stacking I

Dimensionality Constraints

- `np.vstack()`: Concatenates along axis 0. All input arrays must have matching dimensions except along axis 0.
- `np.hstack()`: Concatenates along axis 1 (for 2D arrays). All input arrays must have matching dimensions except along axis 1.

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Vertical Stacking (Row-wise concatenation)
v_stacked = np.vstack((a, b))  # Output shape: (4, 2)

# Horizontal Stacking (Column-wise concatenation)
h_stacked = np.hstack((a, b))  # Output shape: (2, 4)

print("v_stacked:\n", v_stacked)
print("h_stacked:\n", h_stacked)
```

Difference Between Concatenation and Stacking

Unlike `np.concatenate()` or `np.vstack()`/`np.hstack()` which operate on existing axes, `np.stack()` creates a **new axis** and joins arrays along it.

- Given k arrays of shape (M, N):
- `np.stack((a, b), axis=0)` $\rightarrow$ Output shape (k, M, N).
- `np.stack((a, b), axis=-1)` $\rightarrow$ Output shape (M, N, k).

```
import numpy as np

# Two 2D grayscale image frames of shape (28, 28)
img1 = np.ones((28, 28))
img2 = np.zeros((28, 28))

# Stack along new axis 0 (Batch dimension for model input)
batch = np.stack((img1, img2), axis=0)  # Shape: (2, 28, 28)

# Stack along new axis -1 (Channel dimension, e.g., dual-channel)
channels = np.stack((img1, img2), axis=-1)  # Shape: (28, 28, 2)

print("Batch shape:", batch.shape)
print("Channels shape:", channels.shape)
```

& `np.column_stack()`

Depth Stacking & Feature Column Assembly

- `np.dstack()`: Stacks 2D matrices depth-wise along axis 2 (3rd dimension). Essential for building multi-channel image tensors (RGB).
- `np.column_stack()`: Takes 1D arrays and stacks them as columns into a 2D matrix (ideal for feature matrices X).

```python
import numpy as np

# 1D Feature columns -> 2D Feature Matrix X
f1 = np.array([1.5, 2.3, 3.1])
f2 = np.array([10.0, 20.0, 30.0])
X = np.column_stack((f1, f2))
print("Feature Matrix X (shape:", X.shape, "):\n", X)

# Stacking R, G, B planes into 3D RGB image tensor
r_plane = np.full((2, 2), 255)
g_plane = np.full((2, 2), 128)
b_plane = np.full((2, 2), 0)

rgb_image = np.dstack((r_plane, g_plane, b_plane))
print("RGB Image shape:", rgb_image.shape)  # (2, 2, 3)
```

Equal Partition vs. Index-based Partition

- **Equal Split** (`indices_or_sections = N`): Splits array into N equal-sized sub-arrays along specified axis. Requires array size along that axis to be divisible by N.
- **Index-based Split** (`indices_or_sections = [i, j]`): Splits array at specified indices into slices `arr[:i]`, `arr[i:j]`, and `arr[j:]`.

```python
import numpy as np

data = np.arange(12)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

# Equal split into 3 parts
part1, part2, part3 = np.split(data, 3)
print("Equal split part1:", part1)  # [0, 1, 2, 3]

# Custom split at index positions 3 and 7
```

```
s1, s2, s3 = np.split(data, [3, 7])
print("Slice 1 (0..2):", s1)  # [0, 1, 2]
print("Slice 2 (3..6):", s2)  # [3, 4, 5, 6]
print("Slice 3 (7..11):", s3) # [7, 8, 9, 10, 11]
```

& `np.hsplit()`

Row-wise and Column-wise Splitting

- `np.vsplit(arr, N)`: Splits array vertically (along axis 0, row-wise). Equivalent to `np.split(arr, N, axis=0)`.
- `np.hsplit(arr, N)`: Splits array horizontally (along axis 1, column-wise). Equivalent to `np.split(arr, N, axis=1)`.

```
import numpy as np

matrix = np.arange(16).reshape(4, 4)

# Vertical Split: divide rows into two equal (2, 4) matrices
top_half, bottom_half = np.vsplit(matrix, 2)

# Horizontal Split: divide columns into two equal (4, 2) matrices
left_half, right_half = np.hsplit(matrix, 2)

print("Top half shape:", top_half.shape)    # (2, 4)
print("Left half shape:", left_half.shape)  # (4, 2)
```

ML Practical Workflow: Feature/Target Splitting I

Separating Feature Matrix X from Target Vector y

When loading datasets stored as a single matrix $[X \mid y]$, splitting is used to isolate predictors X from target labels y.

```
import numpy as np

# Simulated dataset: 5 rows, 3 feature columns + 1 target column
dataset = np.array([
    [1.2, 0.5, 3.1, 0],
    [2.1, 1.1, 4.0, 1],
    [0.9, 0.2, 2.5, 0],
    [3.4, 1.8, 5.2, 1],
    [2.8, 1.4, 4.8, 1]
])

# Split horizontally at column index 3
X, y = np.hsplit(dataset, [3])

print("Feature Matrix X (shape", X.shape, "):\n", X)
print("Target Vector y (shape", y.shape, "):\n", y.ravel())
```

Summary & Cheat Sheet I

Summary of Stacking & Splitting Functions

Operation	Axis / Dimension	Primary Use Case
`np.vstack()`	Axis 0 (Rows)	Combining rows / dataset instances
`np.hstack()`	Axis 1 (Columns)	Combining feature columns
`np.dstack()`	Axis 2 (Depth)	Stacking channels (e.g. RGB)
`np.stack()`	New Axis	Adding batch/channel dimension
`np.vsplit()`	Axis 0 (Rows)	Train/Test row partitioning
`np.hsplit()`	Axis 1 (Columns)	Separating features X and labels y

Common Errors to Avoid

- `ValueError: all input array dimensions must match...`: Ensure array shapes match on non-stacked axes.
- `ValueError: array split does not result in an equal division`: Use index lists `[i, j]` for unequal splits instead of an integer.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 2**: Python Libraries for Machine Learning
- **Topic**: 2.1.3 Maths Functions: `add()`, `subtract()`, `multiply()`, `divide()`, `power()`, `sqrt()`

Learning Objectives:

1. Understand NumPy's universal functions (`ufuncs`) for fast, element-wise mathematical operations.
2. Master fundamental arithmetic functions: `np.add()`, `np.subtract()`, `np.multiply()`, and `np.divide()`.
3. Apply non-linear transformations using `np.power()` and `np.sqrt()`.
4. Leverage broadcasting and in-place memory allocation (`out` parameter) for optimized machine learning code.
5. Build core machine learning components (loss functions, Euclidean distance, feature scaling) using NumPy math functions.

NumPy Universal Functions (ufuncs) I

What is a Universal Function (ufunc)?

A **ufunc** is a vectorized wrapper around compiled C loops that operates on `ndarray` objects element-by-element. They provide high computational efficiency compared to standard Python loops.

- **Operator Overloading**: Standard Python operators map directly to NumPy ufuncs:
  - `+` $\rightarrow$ `np.add()`
  - `-` $\rightarrow$ `np.subtract()`
  - `*` $\rightarrow$ `np.multiply()`
  - `/` $\rightarrow$ `np.divide()`
  - `**` $\rightarrow$ `np.power()`
- **Key Advantages of Explicit Function Calls**:
  - Support for optional arguments like `out` (in-place modification) and `where` (conditional operations).
  - Functional pipeline integration (e.g., passing functions as callbacks or map targets).

Mathematical Definition

For matrices $A, B \in \mathbb{R}^{m \times n}$, element-wise addition yields matrix C where:

$$C_{i,j} = A_{i,j} + B_{i,j} \quad \forall\, i \in \{1, \ldots, m\},\, j \in \{1, \ldots, n\}$$

```
import numpy as np

# Define input feature vectors / matrices
A = np.array([[1.0, 2.0], [3.0, 4.0]])
B = np.array([[5.0, 6.0], [7.0, 8.0]])

# Explicit function call
C_func = np.add(A, B)

# Equivalent operator syntax
C_op = A + B

print("Function Output:\n", C_func)
# [[ 6.  8.]
#  [10. 12.]]
```

Mathematical Definition & Machine Learning Context

Computes $C_{i,j} = A_{i,j} - B_{i,j}$. Subtraction is essential in machine learning for calculating residual errors $(y - \hat{y})$ and performing gradient descent updates $(w^{(t+1)} = w^{(t)} - \eta \nabla L)$.

```
import numpy as np
```

```
# Ground truth values (y) and model predictions (y_hat)
y_true = np.array([100.0, 150.0, 200.0])
y_pred = np.array([110.0, 140.0, 205.0])

# Compute prediction residual errors (residuals)
residuals = np.subtract(y_pred, y_true)
print("Residuals (y_hat - y):", residuals)  # Output: [10. -10.  5.]

# Gradient Descent Parameter Update Step
weights = np.array([0.5, -0.2, 1.1])
gradients = np.array([0.04, -0.01, 0.12])
lr = 0.01  # Learning rate

updated_weights = np.subtract(weights, np.multiply(lr, gradients))
print("Updated Weights:", updated_weights) # Output: [0.4996 -0.1999 1.0988]
```

Hadamard Product ($A \odot B$)

`np.multiply()` computes the element-wise (Hadamard) product, **not** matrix multiplication ($A \cdot B$).

$$(A \odot B)_{i,j} = A_{i,j} \cdot B_{i,j}$$

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[10, 20], [30, 40]])

# Hadamard (element-wise) multiplication
hadamard = np.multiply(A, B)
print("Element-wise (np.multiply):\n", hadamard)
```

```
# [[ 10  40]
#  [ 90 160]]

# Note: Matrix product uses np.matmul() or @ operator
matrix_prod = A @ B  # Matrix product: [[70, 100], [150, 220]]
```

Mathematical Definition & Special Values

Computes true division $C_{i,j} = A_{i,j}/B_{i,j}$. When dividing by zero, NumPy returns IEEE 754 floating-point values (`inf`, `-inf`, `nan`) and issues runtime warnings.

```
import numpy as np

features = np.array([10.0, 25.0, 50.0])
scale_factors = np.array([2.0, 5.0, 10.0])

# Feature normalization scaling
scaled_features = np.divide(features, scale_factors)
print("Scaled-Features:", scaled_features)  # [5. 5. 5.]

# Handling Division by Zero safely
x = np.array([1.0, 0.0, -2.0])
y = np.array([0.0, 0.0,  0.0])

# Using np.errstate to handle or suppress warnings
with np.errstate(divide='ignore', invalid='ignore'):
    res = np.divide(x, y)
print("Division-by-Zero-Result:", res)  # [ inf  nan -inf]
```

Mathematical Definition

Computes $y_i = x_i^P$ element-wise. Both the base x and exponent p can be scalars or arrays.

$$\text{If } A = [a_1, a_2],\ B = [b_1, b_2] \implies \texttt{np.power}(A, B) = [a_1^{b_1}, a_2^{b_2}]$$

```
import numpy as np

# Scalar exponent: Squaring differences for MSE
errors = np.array([-2.0, 0.5, 3.0])
squared_errors = np.power(errors, 2)
print("Squared Errors:", squared_errors)  # [4.   0.25 9.  ]

# Array exponent: Element-specific powers
bases = np.array([2.0, 3.0, 4.0])
exponents = np.array([1.0, 2.0, 3.0])
powers = np.power(bases, exponents)
print("Element-wise Exponentiation:", powers)  # [ 2.  9. 64.]
```

Mathematical Definition & Domain Restrictions

Computes $y_i = \sqrt{x_i} = x_i^{0.5}$ for non-negative inputs ($x_i \geq 0$). Input of negative values produces `nan` (Not a Number) for real floating-point dtypes.

```
import numpy as np

# Computing variance and standard deviation step
variances = np.array([4.0, 16.0, 25.0, 100.0])
std_devs = np.sqrt(variances)
print("Standard Deviations:", std_devs)  # [ 2.  4.  5. 10.]
```

```python
# Domain warning example: negative numbers
neg_values = np.array([9.0, -4.0, 16.0])
with np.errstate(invalid='ignore'):
    sqrt_neg = np.sqrt(neg_values)
print("Sqrt with Negative Input:", sqrt_neg)  # [ 3.  nan  4.]
```

Broadcasting with Math Functions I

Broadcasting Principle

Broadcasting allows NumPy to execute element-wise operations on arrays of different shapes without making unneeded copies of data in memory.

- **Rule 1**: If arrays differ in rank, prepends 1s to the smaller shape.
- **Rule 2**: Arrays are compatible along a dimension if sizes match or one size is 1.

```
import numpy as np

# Feature matrix X: 3 samples, 2 features (shape: 3x2)
X = np.array([[10.0, 100.0],
              [20.0, 200.0],
              [30.0, 300.0]])

# Feature mean vector mu: 2 features (shape: 2,) -> broadcast to (3,2)
mu = np.array([20.0, 200.0])

# Mean-centering features using np.subtract and broadcasting
X_centered = np.subtract(X, mu)
print("Centered Data:\n", X_centered)
# [[-10. -100.]
#  [  0.    0.]
#  [ 10.  100.]]
```

Why Use out?

By default, `np.add(A, B)` allocates a **new array** in memory to store results. Pass the `out` argument to write results into an existing array, avoiding allocation overhead on large datasets.

```
import numpy as np

# Pre-allocate large arrays
A = np.ones((1000, 1000), dtype=np.float64)
B = np.full((1000, 1000), 2.0, dtype=np.float64)

# 1. Standard approach: Allocates new memory
C = np.add(A, B)  # Creates brand new (1000,1000) array

# 2. In-place modification: Overwrites array A directly
np.add(A, B, out=A)
print("A after in-place addition first element:", A[0, 0])  # 3.0

# 3. Write output to pre-allocated matrix C
np.multiply(A, 0.5, out=C)
print("C output after in-place multiply:", C[0, 0])          # 1.5
```

ML Case Study: Euclidean Distance & RMSE I

Formulas

- **Euclidean Distance**: $d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{d}(u_i - v_i)^2} = \sqrt{\texttt{sum(power(subtract(u, v), 2))}}$
- **Root Mean Squared Error**: RMSE $= \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$

```
import numpy as np

# Feature vectors for two data points u and v
u = np.array([3.0, 4.0, 5.0])
v = np.array([1.0, 1.0, 1.0])

# Compute Euclidean distance using NumPy ufuncs
diff = np.subtract(u, v)
sq_diff = np.power(diff, 2)
euclidean_dist = np.sqrt(np.sum(sq_diff))
print(f"Euclidean-Distance:-{euclidean_dist:.4f}")  # 5.3852

# Compute RMSE for predictions y_hat and targets y
y_true = np.array([3.0, -0.5, 2.0, 7.0])
y_pred = np.array([2.5,  0.0, 2.1, 7.8])

rmse = np.sqrt(np.mean(np.power(np.subtract(y_true, y_pred), 2)))
print(f"RMSE:-{rmse:.4f}")  # 0.4950
```

Summary of Math Functions in Machine Learning I

Function	Operator	Primary ML Applications
`np.add()`	`+`	Bias addition, vector combination
`np.subtract()`	`-`	Residual errors ($y - \hat{y}$), gradient update
`np.multiply()`	`*`	Hadamard product, feature weighting, masks
`np.divide()`	`/`	Min-max normalization, Standard score calculation
`np.power()`	`**`	Polynomial features, MSE loss computation
`np.sqrt()`	`N/A`	Standard deviation, RMSE, Euclidean distance

Key Takeaway

Combining vectorised arithmetic ufuncs with NumPy's broadcasting capabilities allows writing high-performance, readable ML algorithms without explicit Python loops.

Overview of Statistical Functions in NumPy I

Role of Descriptive Statistics in Machine Learning

Statistical functions provide essential summary metrics for understanding data distributions, detecting outliers, and performing feature preprocessing.

- **Central Tendency**: Measures that locate the center of a distribution.
  - `np.mean()`: Arithmetic average of elements.
  - `np.median()`: Middle value separating upper and lower halves.
- **Dispersion / Spread**: Measures that describe data variability.
  - `np.var()`: Average squared deviation from the mean.
  - `np.std()`: Square root of variance (same units as data).
- **Multidimensional Aggregation**: All functions accept an `axis` parameter to compute statistics along specific dimensions (rows or columns).

Mathematical Definition

The mean μ of a dataset $X = \{x_1, x_2, \ldots, x_N\}$ is defined as:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

```
import numpy as np

data = np.array([[10, 20, 30],
                 [40, 50, 60]])

# Global mean across all elements
print("Global-Mean:", np.mean(data))          # Output: 35.0

# Mean along axis 0 (column-wise mean across rows)
print("Axis-0-Mean:", np.mean(data, axis=0))  # Output: [25. 35. 45.]

# Mean along axis 1 (row-wise mean across columns)
print("Axis-1-Mean:", np.mean(data, axis=1))  # Output: [20. 50.]
```

Definition & Properties

- **Median**: The middle element in a sorted, 1D numerical array.
- If N is even, it is the average of the two middle values.
- **Robustness**: Unlike the mean, the median is resistant to extreme outliers.

```
import numpy as np
```

```python
# Dataset containing a severe outlier (e.g., income data)
incomes = np.array([45000, 48000, 52000, 50000, 5000000])

mean_val = np.mean(incomes)
median_val = np.median(incomes)

print(f"Mean Income: ${mean_val:,.2f}")   # $1,039,000.00 (Skewed)
print(f"Median Income: ${median_val:,.2f}") # $52,000.00    (Robust)
```

Mathematical Definition & Degrees of Freedom

Variance measures the average squared distance from the mean:

$$\sigma^2 = \frac{1}{N - \text{ddof}} \sum_{i=1}^{N} (x_i - \mu)^2$$

- ddof=0 (Default): **Population Variance** (divides by N).
- ddof=1: **Sample Variance** (divides by $N - 1$, Bessel's correction).

```python
import numpy as np

samples = np.array([12.5, 14.2, 11.8, 15.0, 13.1])

# Population variance (ddof=0)
pop_var = np.var(samples, ddof=0)

# Sample variance (ddof=1, unbiased estimator for sample data)
sample_var = np.var(samples, ddof=1)
```

```
print(f"Population Variance: {pop_var:.4f}") # Output: 1.2504
print(f"Sample Variance:     {sample_var:.4f}") # Output: 1.5630
```

Definition & Interpretation

Standard deviation σ is the square root of the variance:

$$\sigma = \sqrt{\text{Var}(X)} = \sqrt{\frac{1}{N - \text{ddof}} \sum_{i=1}^{N} (x_i - \mu)^2}$$

Interpretable in the original units of measurement. In a Normal distribution:

- $\approx 68.27\%$ of values lie within $\mu \pm 1\sigma$.
- $\approx 95.45\%$ of values lie within $\mu \pm 2\sigma$.

```
import numpy as np

scores = np.array([88, 92, 79, 95, 85])

std_pop = np.std(scores)          # Population Std Dev
std_sample = np.std(scores, ddof=1) # Sample Std Dev

print(f"Mean: {np.mean(scores):.2f}")
print(f"Population Std: {std_pop:.2f} | Sample Std: {std_sample:.2f}")
```

ML Application: Feature Standardization (Z-score) I

Z-Score Normalization Formula

Standardizing features to have zero mean ($\mu = 0$) and unit variance ($\sigma = 1$):

$$z = \frac{x - \mu}{\sigma}$$

Essential for gradient-based optimizers and distance-based algorithms (KNN, SVM, PCA).

```
import numpy as np

# Feature matrix X: 4 samples, 2 features
X = np.array([[1.0, 200.0],
              [2.0, 500.0],
              [3.0, 300.0],
              [4.0, 800.0]])

# Compute column-wise statistics
mean_X = np.mean(X, axis=0)
std_X = np.std(X, axis=0)

# Apply Z-score transformation
X_scaled = (X - mean_X) / std_X

print("Standardized Features:\n", np.round(X_scaled, 2))
```

ML Application: Feature Standardization (Z-score) II

```
print("Scaled Mean:", np.round(np.mean(X_scaled, axis=0), 2)) # [0., 0.]
print("Scaled Std:", np.round(np.std(X_scaled, axis=0), 2))  # [1., 1.]
```

Summary of NumPy Statistics Functions I

Function	Mathematical Concept	Key Parameter / Property
`np.mean()`	Arithmetic Mean (μ)	Affected by outliers, `axis`
`np.median()`	Median (50th Percentile)	Robust to outliers, `axis`
`np.var()`	Variance (σ^2)	Uses `ddof=0` (Pop) / `ddof=1` (Sample)
`np.std()`	Standard Deviation (σ)	Expressed in original data units

Key Takeaway for ML Pipelines

Always verify parameter settings (`axis` and `ddof`) when normalizing training sets vs test sets to prevent data leakage and ensure unbiased scaling estimators!

Introduction to Pandas in Machine Learning I

What is Pandas?

Pandas is an open-source Python library built on top of NumPy, designed for fast, flexible, and expressive data manipulation and tabular data analysis.

- **Role in the ML Pipeline**:
  - **Data Ingestion**: Loading structured datasets from CSV, JSON, SQL, Parquet, and Excel.
  - **Data Cleaning**: Handling missing values, duplicate entries, and incorrect data types.
  - **Feature Engineering**: Creating derived variables, binning, and one-hot encoding.
  - **Exploratory Data Analysis (EDA)**: Computing summary statistics, correlations, and distributions.
- **Core Data Structures**:
  - `Series`: 1D homogeneous labeled array.
  - `DataFrame`: 2D heterogeneous tabular data structure with labeled axes (rows and columns).

Core Data Structures: Series and DataFrames I

```python
import pandas as pd
import numpy as np

# Creating a Series (1D)
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

# Creating a DataFrame (2D) from a dictionary
data = {
    'Age': [25, 30, 35, 40],
    'Salary': [50000.0, 64000.0, 71000.0, 88000.0],
    'Purchased': [False, True, True, False]
}
df = pd.DataFrame(data)

print(df)
print("Index:", df.index)
print("Columns:", df.columns)
print("Underlying-Values:\n", df.values)
```

Data Ingestion & Initial Exploration I

Essential Data Inspection Methods

Before modeling, always inspect dataset shapes, column types, and basic statistical summaries.

```
# Loading tabular data from CSV
df = pd.read_csv('housing_data.csv')

# Shape and dimensions
print("Dataset Shape:", df.shape)  # Returns (n_samples, n_features)

# Inspect first few rows
print(df.head(3))

# Data types and non-null value counts
df.info()

# Summary statistics (mean, std, min, max, quartiles)
print(df.describe())
```

Indexing and Selection: loc vs iloc I

- `.loc[]`: Selection based on row and column labels.
- `.iloc[]`: Selection based on integer positions (0-indexed).
- **Boolean Indexing**: Filtering rows using conditional expressions.

```
# Select specific rows and columns by label
subset1 = df.loc[0:2, ['Age', 'Salary']]

# Select by integer position indices
subset2 = df.iloc[0:3, 0:2]

# Filtering rows with a boolean condition
high_earners = df[df['Salary'] > 60000.0]

# Multi-condition boolean filtering (& for AND, | for OR)
target_group = df[(df['Age'] >= 30) & (df['Purchased'] == True)]
```

Handling Missing Data in ML Datasets I

Impact of Missing Values

Most machine learning algorithms (e.g., linear models, SVMs, neural networks) fail if input matrices contain missing values (`NaN` or `None`).

```
# Count missing values per column
print(df.isna().sum())

# Strategy 1: Drop rows with missing values
df_clean = df.dropna(axis=0)

# Strategy 2: Drop columns with excessive missing data (> 50%)
df_filtered = df.dropna(thresh=len(df) * 0.5, axis=1)

# Strategy 3: Impute missing numerical values with median
median_salary = df['Salary'].median()
df['Salary'] = df['Salary'].fillna(median_salary)
```

Modifying and Creating Features

Pandas supports efficient vectorized element-wise operations and custom mappings.

```
# Vectorized element-wise feature engineering
df['Salary_Per_Year_Age'] = df['Salary'] / df['Age']

# Custom feature transformation using .apply()
def categorize_age(age):
    return 'Young' if age < 32 else 'Senior'

df['Age_Group'] = df['Age'].apply(categorize_age)

# Categorical mapping using a dictionary
df['Target'] = df['Purchased'].map({True: 1, False: 0})

# Dropping unnecessary columns
df = df.drop(columns=['Purchased'])
```

Grouping and Aggregation I

Split-Apply-Combine Paradigm

GroupBy splits data into groups based on key columns, applies aggregate functions, and combines results.

```
# Compute mean salary grouped by Age_Group
group_means = df.groupby('Age_Group')['Salary'].mean()
print(group_means)

# Computing multiple aggregate statistics simultaneously
stats = df.groupby('Age_Group').agg({
    'Salary': ['mean', 'std', 'min', 'max'],
    'Age': 'count'
})
print(stats)
```

Combining Datasets: Merging and Concatenation I

- **Concatenation** (`pd.concat`): Stacking DataFrames vertically (axis=0) or horizontally (axis=1).
- **Merging** (`pd.merge`): Database-style joins (Inner, Left, Right, Outer) on key columns.

```
# Concatenating rows of two DataFrames
df_combined = pd.concat([df1, df2], axis=0, ignore_index=True)

# Relational inner join on key column 'user_id'
df_merged = pd.merge(
    left=df_user_profile,
    right=df_user_transactions,
    on='user_id',
    how='inner'
)
```

Pandas Integration with ML Pipelines I

Preparing Data for Scikit-Learn

Machine learning models require clean numerical feature matrices ($X \in \mathbb{R}^{n \times d}$) and target vectors ($y \in \mathbb{R}^n$).

```
# One-Hot Encoding for categorical features
df_encoded = pd.get_dummies(df, columns=['Age_Group'], drop_first=True)

# Separate Feature Matrix X and Target Vector y
X = df_encoded.drop(columns=['Target'])
y = df_encoded['Target']

# Extracting NumPy arrays for Scikit-Learn
X_mat = X.to_numpy()
y_vec = y.to_numpy()

print("Feature matrix shape X:", X_mat.shape)
print("Target vector shape y:", y_vec.shape)
```

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 2**: Python Libraries for Machine Learning
- **Topic**: 2.2.1 Data structure: `Series()`, `DataFrame()`

Learning Objectives:

1. Understand the core architectural differences between 1D `pd.Series` and 2D `pd.DataFrame`.
2. Master techniques for creating, indexing, and manipulating Pandas data structures.
3. Utilize index alignment and vectorized operations for efficient feature computation.
4. Apply Series and DataFrames to construct Machine Learning feature matrices (X) and target vectors (y).

Role of Pandas in Machine Learning Pipelines I

Why Pandas for Machine Learning?

Pandas provides high-performance, easy-to-use data structures and data analysis tools built on top of NumPy. It bridges the gap between raw data storage (CSV, SQL, JSON) and numerical ML algorithms.

- **Data Ingestion & Preprocessing**: Loading structured datasets into tabular form.
- **Handling Heterogeneous Types**: Managing numerical, categorical, string, and timestamp data seamlessly within a single container.
- **Explicit Indexing**: Label-based data alignment preventing data misalignment errors during transformations.
- **ML Integration**: Direct conversion to NumPy arrays ($X \in \mathbb{R}^{n \times d}$) required by algorithms (Scikit-Learn, PyTorch, TensorFlow).

Pandas Series: 1D Labeled Array I

Definition

A `pd.Series` is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects).

- **Components**:
  - **Data Values**: Contiguous block of memory stored as a 1D NumPy array (`s.values`).
  - **Index Labels**: Axis labels governing data retrieval and alignment (`s.index`).
  - **Data Type**: Uniform `dtype` across all elements (`s.dtype`).

```
import pandas as pd
import numpy as np

# Creating Series from a List with Custom Index
weights = pd.Series([68.5, 74.2, 59.0],
                    index=['Patient_A', 'Patient_B', 'Patient_C'],
                    name='Weight_kg')
print(weights)
# Patient_A     68.5
# Patient_B     74.2
# Patient_C     59.0
# Name: Weight_kg, dtype: float64
```

Series: Creation Methods & Index Alignment I

- **Creation from Dictionary**: Dictionary keys automatically become index labels.
- **Automatic Index Alignment**: Operations align on index labels, not position.

```
# Series from Dictionary
income_2022 = pd.Series({'Alice': 85000, 'Bob': 92000, 'Charlie': 78000})
income_2023 = pd.Series({'Bob': 98000, 'Charlie': 81000, 'David': 105000})

# Vectorized Addition with Automatic Label Alignment
growth = income_2023 - income_2022
print(growth)
# Alice        NaN  (Missing in 2023)
# Bob       6000.0  (Aligned)
# Charlie   3000.0  (Aligned)
# David        NaN  (Missing in 2022)
# dtype: float64
```

Pandas DataFrame: 2D Tabular Structure I

Definition

A `pd.DataFrame` is a 2D labeled, size-mutable, and potentially heterogeneous tabular data structure with aligned rows and columns.

- Conceptualized as a container of aligned `pd.Series` objects sharing a common row index.
- **Dual Axes**: Axis 0 represents rows (samples N), Axis 1 represents columns (features d).

```
# Constructing DataFrame from Dictionary of Equal-Length Lists
data = {
    'Age': [25, 30, 35, 40],
    'Salary': [50000.0, 64000.0, 82000.0, 110000.0],
    'Target_Purchased': [0, 1, 1, 0]
}
df = pd.DataFrame(data, index=['User_1', 'User_2', 'User_3', 'User_4'])
print(df.shape)  # (4, 3) -> 4 rows, 3 columns
```

DataFrame: Essential Structural Inspection I

- Always inspect data structures before feeding them to ML models to check for data types and missing values.

```
# Key inspection attributes and methods
print(df.dtypes)          # Returns data type of each feature column
print(df.shape)           # Returns tuple (num_rows, num_cols)
print(df.columns)         # Index object containing column names

# Summary Statistics for Numerical Columns
print(df.describe())
#                Age         Salary  Target_Purchased
# count    4.000000       4.000000          4.000000
# mean    32.500000   76500.000000          0.500000
# std      6.454972   25916.532690          0.577350
# min     25.000000   50000.000000          0.000000
# max     40.000000  110000.000000          1.000000
```

Subsetting & Accessing Data: loc vs iloc I

- `.loc[]`: Purely label-based indexing (Row label, Column label).
- `.iloc[]`: Purely integer-positional indexing (0-based integer offsets).

```
# 1. Column Selection
age_series = df['Age']                    # Returns pd.Series
sub_df = df[['Age', 'Salary']]            # Returns pd.DataFrame

# 2. Label-based Slicing with .loc
row_user2 = df.loc['User_2', ['Age', 'Salary']]

# 3. Positional Slicing with .iloc
sub_matrix = df.iloc[0:2, 0:2]            # First 2 rows, first 2 columns

# 4. Conditional (Boolean) Filtering for Data Cleaning
high_earners = df[df['Salary'] > 70000]
```

Comparison: pd.Series vs pd.DataFrame I

Property	pd.Series	pd.DataFrame
Dimensions	1D (Vector)	2D (Matrix/Table)
Data Homogeneity	Single `dtype` across all elements	Heterogeneous (different `dtype` per column)
ML Representation	Target Vector $y \in \mathbb{R}^N$ or single Feature	Feature Matrix $X \in \mathbb{R}^{N \times d}$
Axis Labels	Single Index (Row labels)	Row Index (Axis 0) & Column Index (Axis 1)
Conversion	`s.to_frame()` converts to DataFrame	Extracting single column yields Series

Table: Architectural Comparison of Series and DataFrame

Machine Learning Workflow Integration I

Constructing Feature Matrix X and Target Vector y

In supervised ML, input features must be isolated from the target label.

```
# Splitting DataFrame into Feature Matrix (X) and Target Vector (y)
X = df.drop(columns=['Target_Purchased'])  # 2D DataFrame (N samples, d features)
y = df['Target_Purchased']                 # 1D Series (N samples)

# Extracting raw NumPy arrays for Scikit-Learn / PyTorch model fitting
X_mat = X.to_numpy()    # Shape: (4, 2), dtype: float64
y_vec = y.to_numpy()    # Shape: (4,),   dtype: int64

print(f"X-type:-{type(X)},-Shape:-{X.shape}")
print(f"y-type:-{type(y)},-Shape:-{y.shape}")
```

Lecture Summary & Best Practices I

Key Takeaways

- **Series**: 1D labeled array designed for single variables/targets.
- **DataFrame**: 2D tabular container for multidimensional datasets (X).
- **Indexing Alignment**: Arithmetic operations automatically align on index labels.
- **Explicit Indexing**: Prefer `.loc` and `.iloc` over chained indexing (`df[][]`) to avoid copy warnings.

Best Practice Note

Avoid using `inplace=True` in modern Pandas code as it is being deprecated. Always reassign returned DataFrames (e.g., `df = df.drop(...)`) for cleaner code execution.

Data Manipulation Essentials in Pandas I

Overview

Data manipulation is a fundamental step in the Machine Learning pipeline, enabling data exploration, cleaning, indexing, and transformation before model training.

- **Data Structure & Inspection**: Examining shape, column titles, and quick previews.
- **Indexing & Selection**: Retrieving specific rows and columns using labels.
- **Handling Missing Data**: Detecting and removing null or missing values.
- **Data Cleaning**: Removing duplicate entries and unnecessary features.
- **Aggregations & Sorting**: Summary statistics and reordering datasets.

, columns, head(), and tail()

Dataset Structure & Quick Preview

Before applying ML algorithms, inspect the dimensions and structure of the dataset.

- shape: Attribute returning a tuple (num_rows, num_cols).
- columns: Attribute containing an index of all column labels.
- head(n): Method returning the first n rows (default $n = 5$).
- tail(n): Method returning the last n rows (default $n = 5$).

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'Score': [85.5, 92.0, 78.0]}
df = pd.DataFrame(data)

print(df.shape)      # (3, 3)
print(df.columns)    # Index(['Name', 'Age', 'Score'], dtype='object')
print(df.head(2))    # Returns first 2 rows
```

Label-Based Indexer: loc[]

Accesses a group of rows and columns by label(s) or a boolean array.

- Syntax: df.loc[row_indexer, column_indexer]
- Slicing with labels includes **both** the start and stop endpoints.

- Supports boolean selection for conditional filtering.

```
df_idx = df.set_index('Name')

# Access specific entry by label
age_bob = df_idx.loc['Bob', 'Age']  # 30

# Conditional indexing: Select rows where Score > 80
high_scores = df.loc[df['Score'] > 80, ['Name', 'Score']]
print(high_scores)
```

and `sum()`

Detecting Missing Values

Real-world ML datasets often contain missing values (`NaN` or `None`).

- `isnull()`: Returns a boolean DataFrame indicating `True` where values are null.
- `sum()`: When chained (`df.isnull().sum()`), counts total missing values per column.

```
import numpy as np

raw_data = {'A': [1, np.nan, 3],
            'B': [np.nan, 5, 6],
            'C': [7, 8, np.nan]}
df_null = pd.DataFrame(raw_data)

# Count missing values per column
print(df_null.isnull().sum())
# Output: A: 1, B: 1, C: 1
```

and drop()

Removing Missing Values & Dropping Features

Clean datasets by discarding incomplete records or removing irrelevant features.

- `dropna(axis=0/1, how='any'/'all')`: Drops rows (`axis=0`) or columns (`axis=1`) containing missing values.
- `drop(labels, axis=0/1)`: Removes specified labels from rows or columns.

```
# Remove all rows with at least one missing value
df_clean = df_null.dropna()

# Drop specific column ('C') from DataFrame
df_no_c = df_null.drop(columns=['C'])

# Drop row by index label (e.g., index 0)
df_no_row0 = df_null.drop(index=0)
```

Identifying Redundant Data

Duplicate entries can distort statistical analysis and model evaluations.

- `duplicated(keep='first'/'last'/False)`: Returns a boolean Series flagging duplicate rows.
- Parameter `keep='first'` marks duplicates except for the first occurrence.
- Combine with boolean indexing to inspect duplicate rows.

```
dup_data = {'ID': [101, 102, 102, 103],
            'Feature': ['A', 'B', 'B', 'C']}
df_dup = pd.DataFrame(dup_data)
```

```
# Get boolean mask of duplicate rows
print(df_dup.duplicated())

# Filter and display duplicate rows
print(df_dup[df_dup.duplicated()])
```

, `max()`, and `sum()`

Numerical Aggregations

Pandas provides fast summary statistic methods across Series or DataFrames.

- `sum()`: Returns sum of values over requested axis (`axis=0` column-wise, `axis=1` row-wise).
- `min()`: Computes minimum value along specified axis.
- `max()`: Computes maximum value along specified axis.

```
scores = pd.DataFrame({
    'Midterm': [75, 88, 92],
    'Final': [80, 85, 95]
})

print("Column-Min:", scores.min())
print("Column-Max:", scores.max())
print("Total-Sum-per-Student:\n", scores.sum(axis=1))
```

Sorting Datasets

Reorder rows based on values in one or more columns.

- `sort_values(by, ascending=True, inplace=False)`: Sorts DataFrame by specified column(s).

- Supports sorting by multiple columns with individual ordering directions.

```
df_students = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Group': ['B', 'A', 'A', 'B'],
    'Score': [88, 95, 92, 85]
})

# Sort primarily by Group (ascending), secondarily by Score (descending)
df_sorted = df_students.sort_values(
    by=['Group', 'Score'],
    ascending=[True, False]
)
print(df_sorted)
```

Summary: Data Manipulation in ML Workflow I

Preprocessing Pipeline Roadmap

Connecting Pandas data manipulation operations to the ML pipeline:

1. **Exploration**: Use `shape`, `columns`, `head()`, and `tail()` for initial dataset assessment.
2. **Cleaning**: Detect missing entries with `isnull().sum()`, clean with `dropna()`, and filter duplicates with `duplicated()`.
3. **Feature Engineering**: Isolate features/labels via `loc[]`, drop irrelevant variables using `drop()`.
4. **Analysis & Ordering**: Summarize features with `min()`, `max()`, `sum()`, and structure results with `sort_values()`.

Introduction to CSV Handling in Pandas I

What is a CSV File?

- **CSV (Comma-Separated Values):** A plain-text tabular data format widely used in machine learning datasets.
- Records are stored line-by-line; data fields are separated by a delimiter (commonly a comma).

Pandas I/O Capabilities

- `pd.read_csv()`: Parses flat CSV files into a Pandas `DataFrame`.
- `DataFrame.to_csv()`: Exports `DataFrame` records to disk as a CSV file.
- Utilizes an optimized C-parsing engine for fast data loading in ML workflows.

Reading CSV Files: Core Parameters of read_csv() I

Essential Arguments

- `filepath_or_buffer`: File path, URL string, or file-like object.
- `sep / delimiter`: Character separator (default is ','; e.g., '\t', ';').
- `header`: Line index for column titles (default 0; set to `None` if no header exists).
- `names`: List of custom column names to assign.

```
import pandas as pd

# Basic loading with header inference
df = pd.read_csv('housing.csv')

# Custom delimiter and explicit column names
df_custom = pd.read_csv(
    'data.txt',
    sep=';',
    header=None,
    names=['Age', 'Income', 'Target']
)
```

read_csv(): Indexing, Column Selection, and Dtypes I

Optimizing Import Performance

- `index_col`: Column(s) to set as row labels in the `DataFrame`.
- `usecols`: Load only a specific subset of columns to save memory.
- `dtype`: Explicitly declare column data types to avoid inference overhead.

```
# Select columns, set index, and specify exact dtypes
df = pd.read_csv(
    'dataset.csv',
    index_col='ID',
    usecols=['ID', 'Age', 'Salary', 'Purchased'],
    dtype={'Age': 'int32', 'Salary': 'float64', 'Purchased': 'category'}
)

print(df.dtypes)
```

read_csv(): Handling Missing Data and Dates I

Data Cleaning at Load Time

- `na_values`: Custom scalar, string, or list of values to recognize as `NaN`.
- `keep_default_na`: Boolean determining whether to append custom `na_values` to standard NaN strings.
- `parse_dates`: Automatically convert date-like string columns into `datetime64` objects.

```
# Custom missing value sentinel values and date parsing
df = pd.read_csv(
    'sales_log.csv',
    na_values=['NA', 'missing', '-999', 'N/A'],
    keep_default_na=True,
    parse_dates=['Date_Time']
)

print(df.isna().sum())
```

Handling Large Datasets: Chunking with read_csv() I

Memory-Efficient Batch Loading

- Loading multi-gigabyte CSV files all at once can exceed system memory.
- `chunksize`: Returns an iterable `TextFileReader` object yielding `DataFrame` chunks of specified size.

```
# Process a large CSV file iteratively in chunks of 50,000 rows
chunk_size = 50000
total_filtered = 0

for chunk in pd.read_csv('big_data.csv', chunksize=chunk_size):
    # Filter rows per chunk without loading entire file into RAM
    filtered_chunk = chunk[chunk['score'] > 0.8]
    total_filtered += len(filtered_chunk)

print(f"Total matching records: {total_filtered}")
```

Writing CSV Files: DataFrame.to_csv() I

Key Export Arguments

- `path_or_buf`: Destination file path or write stream.
- `index`: Set to `False` to prevent writing row index numbers as an un-named column.
- `header`: Write column titles (default `True`).
- `na_rep`: String representation for missing `NaN` values (default `''`).

```
import pandas as pd

df = pd.DataFrame({
    'ID': [101, 102, 103],
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Score': [85.5, None, 92.0]
})

# Standard export excluding auto-generated integer index
df.to_csv('processed_data.csv', index=False, na_rep='NaN')
```

to_csv(): Advanced Export Features I

Selective Output, Encoding, and Compression

- `columns`: Specify a subset of columns to write out.
- `encoding`: Character encoding format (e.g., `'utf-8'`, `'latin1'`, `'utf-8-sig'`).
- `compression`: Compress output directly on write (`'gzip'`, `'zip'`, `'bz2'`).

```
# Export selected columns directly to a gzipped CSV with UTF-8 encoding
df.to_csv(
    'output_selected.csv.gz',
    columns=['ID', 'Score'],
    index=False,
    encoding='utf-8',
    compression='gzip'
)
```

Complete Workflow: Load, Process, and Save I

Practical Machine Learning Data Pipeline

An end-to-end workflow reading raw data, filtering missing targets, selecting features, and writing clean output.

```
import pandas as pd

# 1. Load raw CSV with missing value sentinels and date parsing
df = pd.read_csv('raw_data.csv', na_values=['?'], parse_dates=['timestamp'])

# 2. Clean missing labels and select relevant features
clean_df = df.dropna(subset=['target']).copy()
features = ['timestamp', 'feature_1', 'feature_2', 'target']
final_df = clean_df[features]

# 3. Export processed dataset for downstream modeling
final_df.to_csv('clean_dataset.csv', index=False)
print("Pipeline complete: clean_dataset.csv created successfully.")
```

Introduction to Matplotlib & Core Architecture I

What is Matplotlib?

- The foundational 2D visualization library in Python's scientific computing stack.
- Provides full control over figure elements, rendering publication-quality graphics.

Matplotlib Architecture: Object-Oriented vs. Pyplot

- **Pyplot API (`pyplot`):** State-based interface (similar to MATLAB); quick for simple scripts.
- **Object-Oriented (OO) API:** Explicitly creates `Figure` (canvas) and `Axes` (plot window); recommended for robust ML visualization pipelines.

```
import matplotlib.pyplot as plt
import numpy as np

# Object-Oriented approach (Preferred in ML workflows)
fig, ax = plt.subplots(figsize=(8, 4))
ax.set_title("Object-Oriented Interface Example")
```

Line Plots: Tracking Model Training Curves I

Applications in Machine Learning

- Monitoring loss function minimization over training epochs.
- Detecting overfitting (divergence between training and validation loss).
- Tracking hyperparameter tuning metrics across iterations.

```
epochs = np.arange(1, 11)
train_loss = [0.9, 0.6, 0.4, 0.3, 0.25, 0.2, 0.18, 0.15, 0.14, 0.12]
val_loss   = [0.95, 0.65, 0.48, 0.38, 0.35, 0.36, 0.39, 0.42, 0.45, 0.48]

fig, ax = plt.subplots(figsize=(7, 3.5))
ax.plot(epochs, train_loss, label='Training Loss', color='blue', marker='o')
ax.plot(epochs, val_loss, label='Validation Loss', color='red', linestyle='--')
ax.set_xlabel('Epochs')
ax.set_ylabel('Cross-Entropy Loss')
ax.set_title('Training vs Validation Loss (Overfitting Detection)')
ax.legend()
ax.grid(True, linestyle=':', alpha=0.6)
```

Scatter Plots: Visualizing Feature Relationships I

EDA and Clustering Inspection

- `ax.scatter()`: Plots bivariate data points to analyze feature correlation.
- Useful for visualizing 2D projection of classes, cluster assignments, and decision boundaries.
- Color-coding by class labels (c=y) and scaling markers by feature magnitude.

```
# Generate synthetic 2-feature classification dataset
X = np.random.randn(100, 2)
y = (X[:, 0] + X[:, 1] > 0).astype(int)

fig, ax = plt.subplots(figsize=(7, 3.5))
scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k', alpha=0.8)
ax.set_xlabel('Feature-1-($x_1$)')
ax.set_ylabel('Feature-2-($x_2$)')
ax.set_title('2D-Feature-Distribution-by-Class-Label')
cbar = fig.colorbar(scatter, ax=ax)
cbar.set_label('Class-Target')
```

Histograms & Box Plots: Data Distribution Analysis I

Assessing Feature Distributions

- **Histograms (`ax.hist`):** Inspect normality, skewness, multimodality, and zero-inflation.
- **Box Plots (`ax.boxplot`):** Identify median, interquartile range (IQR), and empirical outliers.

```
feature_data = np.random.normal(loc=50, scale=15, size=500)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 3.5))

# Histogram with probability density normalization
ax1.hist(feature_data, bins=25, color='skyblue', edgecolor='black', density=True)
ax1.set_title('Feature Histogram & Density')
ax1.set_xlabel('Feature Value')

# Box plot for outlier detection
ax2.boxplot(feature_data, patch_artist=True, boxprops=dict(facecolor='lightgreen'))
ax2.set_title('Feature Box Plot')
ax2.set_ylabel('Value Scale')
```

Bar Charts: Class Imbalance & Feature Importance I

Categorical Visualization in ML

- Visualizing class frequencies to detect data imbalance before model training.
- Displaying relative feature importances from decision trees or random forests.

```
features = ['Age', 'Income', 'Credit_Score', 'Education', 'Zipcode']
importance = [0.35, 0.28, 0.20, 0.12, 0.05]

fig, ax = plt.subplots(figsize=(7, 3.5))
# Horizontal bar chart for readable feature labels
y_pos = np.arange(len(features))
ax.barh(y_pos, importance, align='center', color='teal', alpha=0.85)
ax.set_yticks(y_pos)
ax.set_yticklabels(features)
ax.invert_yaxis()  # Labels read top-to-bottom
ax.set_xlabel('Gini Importance Score')
ax.set_title('Random Forest Feature Importance')
```

Multi-Panel Layouts: Creating Subplot Dashboards I

Subplot Management with `plt.subplots()`

- `plt.subplots(nrows, ncols)`: Generates a grid of Axes objects.
- Essential for building multi-metric diagnostics (e.g., Loss, Accuracy, Precision, Recall).
- Use `plt.tight_layout()` to automatically adjust spacing and prevent overlapping labels.

```
fig, axes = plt.subplots(2, 2, figsize=(8, 4.5))

axes[0, 0].plot([1, 2, 3], [2, 4, 9], 'b-')
axes[0, 0].set_title('Training Loss')

axes[0, 1].plot([1, 2, 3], [0.5, 0.8, 0.95], 'g-')
axes[0, 1].set_title('Validation Accuracy')

axes[1, 0].hist(np.random.randn(100), color='purple')
axes[1, 0].set_title('Residuals')

axes[1, 1].scatter(np.random.rand(20), np.random.rand(20), color='orange')
axes[1, 1].set_title('Predictions vs Actuals')

plt.tight_layout()
```

Matrix Visualizations: Heatmaps & Confusion Matrices I

Visualizing 2D Data Matrices

- `ax.imshow()` renders numeric 2D arrays as color-encoded heatmaps.
- Widely used for confusion matrix evaluation and feature correlation matrices.

```
# 3x3 Confusion Matrix Example
conf_matrix = np.array([[45, 3, 2],
                        [ 1, 38, 5],
                        [ 4, 2, 50]])

fig, ax = plt.subplots(figsize=(5, 4))
cax = ax.imshow(conf_matrix, cmap='Blues')

# Add numeric labels to each matrix cell
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(j, i, str(conf_matrix[i, j]), ha='center', va='center',
                color='white' if conf_matrix[i, j] > 25 else 'black')

fig.colorbar(cax)
ax.set_xticks([0, 1, 2]); ax.set_yticks([0, 1, 2])
ax.set_xticklabels(['Class-0', 'Class-1', 'Class-2'])
ax.set_yticklabels(['Class-0', 'Class-1', 'Class-2'])
ax.set_xlabel('Predicted-Label'); ax.set_ylabel('True-Label')
```

Customization & Saving Publication-Quality Figures I

Exporting and Styling Figures

- **Built-in Styles:** `plt.style.use('seaborn-v0_8')` or `'ggplot'` for pre-configured aesthetics.
- **Text Annotations:** `ax.annotate()` to highlight critical points (e.g., minimum loss).
- **High-Resolution Export:** `fig.savefig()` for vector (PDF, SVG) or high-DPI raster (PNG) formats.

```
# Custom styling and vector export
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(6, 3.5))
x = np.linspace(0, 10, 100)
ax.plot(x, np.sin(x), label=r'$\sin(x)$', color='darkblue', lw=2)

# Annotation for key point
ax.annotate('Peak', xy=(np.pi/2, 1), xytext=(np.pi/2 + 1, 0.8),
            arrowprops=dict(facecolor='black', shrink=0.05))

# Export with tight bounding box to remove extra padding
fig.savefig('sine_plot.pdf', dpi=300, bbox_inches='tight')
```

Summary: Key Matplotlib Functions for Machine Learning I

Visualization Type	Matplotlib Method	ML Use Case
Line Plot	`ax.plot()`	Learning curves, loss tracking
Scatter Plot	`ax.scatter()`	Cluster analysis, 2D projections
Histogram	`ax.hist()`	Feature distribution, skewness
Box Plot	`ax.boxplot()`	Outlier detection, IQR range
Bar Chart	`ax.bar()` / `ax.barh()`	Feature importance, class balance
Heatmap	`ax.imshow()`	Confusion matrix, correlation

Best Practices

- Always label axes (`set_xlabel`, `set_ylabel`) and include legends.
- Prefer the Object-Oriented API (`fig, ax`) over stateful `plt.*` calls.
- Export vector graphics (`.pdf`, `.svg`) for academic reports.

Overview of Matplotlib Visualization Functions I

Role of Data Visualization in Machine Learning

Visualizing data is a fundamental step in Exploratory Data Analysis (EDA), model evaluation, and diagnostic reporting. Matplotlib provides lower-level, highly customizable plotting functions.

- **Basic Trend & Relationship Functions**:
  - `plt.plot()`: Line plots for continuous functions, time series, and loss curves.
  - `plt.scatter()`: Scatter plots for feature relationships, clustering, and classification data.
- **Distribution & Summary Functions**:
  - `plt.bar()` & `plt.pie()`: Categorical distributions and class composition.
  - `plt.hist()`: Histograms for univariate continuous feature distributions.
  - `plt.boxplot()`: Box plots for identifying outliers and quartile statistics.
- **Annotation & Exporting Functions**:
  - `plt.title()`, `plt.xlabel()`, `plt.ylabel()`, `plt.grid()`: Chart formatting.
  - `plt.show()` & `plt.savefig()`: Rendering and saving figure artifacts.

Tracking Continuous Metrics

`plt.plot()` connects numerical data points with lines. It is widely used to visualize training loss curves, evaluation accuracy over iterations, and time-series data.

```
import matplotlib.pyplot as plt
import numpy as np

epochs = np.arange(1, 11)
train_loss = [0.9, 0.7, 0.5, 0.35, 0.25, 0.18, 0.14, 0.11, 0.09, 0.08]
val_loss   = [0.95, 0.75, 0.55, 0.42, 0.33, 0.28, 0.26, 0.25, 0.26, 0.27]

plt.figure(figsize=(7, 4))
plt.plot(epochs, train_loss, label='Training Loss', color='blue', linestyle='-', marker='o')
plt.plot(epochs, val_loss, label='Validation Loss', color='red', linestyle='--', marker='s')

plt.title('Model Training vs Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.grid(True)
plt.legend()
```

for Feature Inspection

Bivariate Feature Correlation & Clustering

`plt.scatter()` plots individual data points without connecting lines. Essential for visualizing feature correlation, decision boundaries, and cluster assignments.

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Synthetic feature data for two classes
np.random.seed(42)
x1 = np.random.randn(100)
x2 = 2 * x1 + np.random.randn(100)
labels = np.random.choice([0, 1], size=100)

plt.figure(figsize=(7, 4))
scatter = plt.scatter(x1, x2, c=labels, cmap='coolwarm', s=50, alpha=0.8, edgecolors='k')

plt.title('2D Feature Relationship Scatter Plot')
plt.xlabel('Feature 1 ($x_1$)')
plt.ylabel('Feature 2 ($x_2$)')
plt.colorbar(scatter, label='Class Label')
plt.grid(True, linestyle=':')
```

and `plt.pie()`

Analyzing Discrete Features and Class Imbalance

- `plt.bar()`: Represents categorical variables with rectangular bars proportional to values.
- `plt.pie()`: Displays relative proportions of a whole as slices of a pie.

```
import matplotlib.pyplot as plt

classes = ['Class-A', 'Class-B', 'Class-C', 'Class-D']
counts = [450, 300, 150, 100]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4))

# Bar Chart for Class Counts
ax1.bar(classes, counts, color=['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728'])
```

```
ax1.set_title('Class-Frequency-Distribution')
ax1.set_ylabel('Count')

# Pie Chart for Class Proportions
ax2.pie(counts, labels=classes, autopct='%1.1f%%', startangle=90, explode=(0.1, 0, 0, 0))
ax2.set_title('Class-Proportion-Share')

plt.tight_layout()
```

Histogram Analysis in Feature Preprocessing

`plt.hist()` bins continuous values to estimate probability density distributions. Crucial for detecting skewed distributions, multimodality, and variance issues.

```
import matplotlib.pyplot as plt
import numpy as np

# Feature with normal distribution
feature_data = np.random.normal(loc=50, scale=10, size=1000)

plt.figure(figsize=(7, 4))
n, bins, patches = plt.hist(feature_data, bins=25, color='skyblue', edgecolor='black', alpha=0.7)

plt.title('Feature-Value-Distribution')
plt.xlabel('Feature-Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
```

Five-Number Summary & Outlier Identification

Box plots summarize statistical distributions via five key numbers:

- **Median (Q_2)**: Middle horizontal line.
- **Interquartile Range (IQR $= Q_3 - Q_1$)**: Box length from 25th to 75th percentile.
- **Whiskers**: Extend to 1.5 $\times$ IQR beyond Q_1 and Q_3.
- **Outliers**: Individual points plotted beyond whisker limits.

```
import matplotlib.pyplot as plt
import numpy as np

data1 = np.random.normal(100, 10, 200)
data2 = np.concatenate([np.random.normal(90, 20, 200), [160, 170, 20]]) # with outliers

plt.figure(figsize=(7, 3.5))
plt.boxplot([data1, data2], labels=['Feature-1', 'Feature-2-(Outliers)'], patch_artist=True)
plt.title('Feature-Comparison-Box-Plot')
plt.ylabel('Value-Scale')
plt.grid(axis='y')
```

and `plt.savefig()`

Workflow for Saving High-Resolution Figures

- `plt.grid()`: Toggles grid lines for easier numeric alignment.
- `plt.savefig()`: Exports plots to disk (PNG, PDF, SVG). Call **before** `plt.show()`!
- `plt.show()`: Displays the plot interface and flushes the active figure memory buffer.

```
import matplotlib.pyplot as plt
```

```python
import numpy as np

x = np.linspace(0, 10, 100)
plt.figure(figsize=(6, 3.5))
plt.plot(x, np.sin(x), label='sin(x)', color='purple')

plt.title('Sine-Wave-Signal')
plt.xlabel('Time-(t)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()

# Save figure to file BEFORE calling show()
plt.savefig('sine_wave.png', dpi=300, bbox_inches='tight')
plt.show()
```

Summary of Matplotlib Functions I

Function	Primary Machine Learning Use Case	Key Parameters
`plot()`	Loss curves, time-series, metric trends	`linestyle`, `marker`, `color`
`scatter()`	2D feature correlation, cluster visualization	`s`, `c`, `cmap`, `alpha`
`bar()`	Class imbalance, feature importance counts	`x`, `height`, `color`
`hist()`	Feature density distribution, skew detection	`bins`, `density`, `alpha`
`boxplot()`	Outlier detection, quantile analysis	`vert`, `patch_artist`
`pie()`	Class proportion breakdown	`autopct`, `explode`
`savefig()`	Exporting figures for research papers/reports	`dpi`, `bbox_inches`

Best Practices in ML Visualization

Always label axes (`xlabel`, `ylabel`), add a descriptive `title`, include a grid or legend when appropriate, and execute `savefig()` before `show()` to prevent saving blank figures!

Overview of Scikit-learn in Python ML I

What is Scikit-learn?

Scikit-learn (`sklearn`) is the industry-standard Python library for classical machine learning algorithms, built on top of NumPy, SciPy, and Matplotlib.

- **Unified Interface**: Provides a consistent, object-oriented API across disparate model families (linear models, trees, support vector machines, clustering).
- **Standardized Data Model**:
  - **Feature Matrix** X: A 2D array of shape $(n_{\text{samples}}, n_{\text{features}})$ containing input covariates.
  - **Target Vector** y: A 1D array of shape $(n_{\text{samples}},)$ containing target labels or continuous responses.
- **Scope**: Supervised learning (classification, regression), unsupervised learning (clustering, dimensionality reduction), and preprocessing/model selection.

Scikit-learn API Design Principles I

Core Architectural Principles (Buitinck et al., 2013)

- **Consistency**: All objects adhere to a minimalist, uniform interface.
- **Inspection**: Constructor parameters and learned model attributes are public.
- **Non-factoring of objects**: Data is represented as standard NumPy arrays or SciPy matrices rather than proprietary classes.

- **Estimators**: Any object that learns from data via `fit(X, y)`.
  - Learned parameters end with a trailing underscore (e.g., `model.coef_`, `scaler.mean_`).
- **Transformers**: Objects that modify or filter data via `transform(X)` or `fit_transform(X)`.
- **Predictors**: Objects capable of producing inferences on new data via `predict(X)` or `predict_proba(X)`.

Data Preprocessing and Leakage Prevention I

The Fit-Transform Paradigm

To prevent **data leakage**, transformation parameters (e.g., feature mean μ and standard deviation σ) must be computed *solely* on the training split, then applied to both train and test splits.

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Synthetic feature matrix (5 samples, 2 features) and target
X = np.array([[100.0, 0.1], [200.0, 0.4], [150.0, 0.2],
              [300.0, 0.8], [250.0, 0.5]])
y = np.array([0, 0, 0, 1, 1])

# Split dataset prior to scaling
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Fit scaler ON TRAINING DATA ONLY
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Transform test data using training parameters (mu, sigma)
X_test_scaled = scaler.transform(X_test)
```

Supervised Learning Workflow: Classification I

Model Lifecycle

Instantiate Model $\longrightarrow$ Fit on Training Data $\longrightarrow$ Predict on Unseen Data $\longrightarrow$ Evaluate Metrics

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load benchmark Iris dataset
iris = load_iris()
X_tr, X_te, y_tr, y_te = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=42
)

# Instantiate estimator with hyperparameters
clf = LogisticRegression(max_iter=200, C=1.0)
clf.fit(X_tr, y_tr)

# Inference and quantitative evaluation
y_pred = clf.predict(X_te)
print(f"Accuracy: {accuracy_score(y_te, y_pred):.4f}")
print("Learned Coefficients Shape:", clf.coef_.shape)
```

Model Selection: Cross-Validation & Hyperparameter Tuning I

Cross-Validation & Grid Search

`GridSearchCV` evaluates hyperparameter combinations using K-Fold cross-validation to optimize model capacity while preventing overfitting.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Base estimator
rf = RandomForestClassifier(random_state=42)

# Define hyperparameter search grid
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [3, 5, None]
}

# 5-Fold Cross-Validation Grid Search
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           cv=5, scoring='accuracy')
grid_search.fit(X_tr, y_tr)

print("Best Parameters:", grid_search.best_params_)
print(f"Best CV Accuracy: {grid_search.best_score_:.4f}")
```

Encapsulation with `sklearn.pipeline.Pipeline`

`Pipeline` chains preprocessing steps and an estimator into a single unified object, preventing data leakage during cross-validation and simplifying production deployment.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

# Define ordered sequence of transformers and final estimator
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
    ('classifier', SVC(kernel='rbf', C=1.0))
])

# Single fit call triggers sequential fit_transform and final fit
pipeline.fit(X_tr, y_tr)

# Predict automatically transforms test features before inference
y_pred_pipe = pipeline.predict(X_te)
print(f"Pipeline Test Accuracy: {pipeline.score(X_te, y_te):.4f}")
```

Summary of Key Scikit-learn Modules I

Module	Key Classes / Functions	Primary Purpose
`preprocessing`	`StandardScaler`, `OneHotEncoder`	Feature transformation & scaling
`model_selection`	`train_test_split`, `GridSearchCV`	Validation & tuning
`linear_model`	`LinearRegression`, `LogisticRegression`	Linear baseline estimators
`ensemble`	`RandomForestClassifier`, `HistGradientBoostingClassifier`	Tree ensemble models
`metrics`	`accuracy_score`, `confusion_matrix`	Evaluation & diagnostics
`pipeline`	`Pipeline`, `make_pipeline`	Workflow composition

Best Practice

Always construct full workflows using `Pipeline` objects to guarantee zero data leakage during validation and cross-validation!

Overview of Machine Learning Activities I

The Lifecycle of an ML Project

Developing a machine learning solution is an iterative engineering process involving structured steps from raw data to deployed models.

- **Problem Definition**: Identifying business or research goals, target variables, and success metrics.
- **Data Ingestion & Exploration**: Understanding distributions, anomalies, missingness, and feature relationships.
- **Data Preprocessing**: Data cleaning, missing value imputation, categorical encoding, and feature scaling.
- **Feature Engineering**: Extracting domain-specific representations to enhance model predictive power.
- **Model Building & Validation**: Selection of algorithms, validation strategy, metric calculation, and error diagnostics.
- **Deployment & Monitoring**: Productionizing model pipelines and tracking data/concept drift over time.

Data Preprocessing & Cleaning I

Key Data Cleaning Steps

Raw real-world data is uncurated and noisy. Systematic cleaning is essential to prevent biased or failing models.

1. **Handling Missing Values**:
   - Imputation via mean, median, mode, or model-based methods (e.g., K-NN imputation).
   - Dropping missing features/records when missingness threshold is excessively high ($> 50\%$).
2. **Outlier Identification & Remediation**:
   - Statistical detection using Interquartile Range ($\text{IQR} = Q_3 - Q_1$) or Z-score ($|Z| > 3$).
   - Winsorization (capping values) or truncation.
3. **Deduplication & Data Consistency**:
   - Removing duplicate observations and fixing non-standard string formats.

Feature Encoding and Feature Scaling I

Categorical Encoding

Transforms non-numeric attributes into quantitative representations:

- **One-Hot Encoding**: Creates binary indicator columns for nominal features (prevents artificial ordering).
- **Ordinal Encoding**: Maps ordered categories to sequential integers (e.g., Low $\rightarrow$ 0, Medium $\rightarrow$ 1, High $\rightarrow$ 2).

Feature Scaling

Prevents features with larger magnitudes from dominating distance-based algorithms and gradient updates:

- **Min-Max Normalization**: Rescales data to $[0, 1]$ range:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

- **Standardization (Z-score)**: Centers data to zero mean and unit variance:

$$x' = \frac{x - \mu}{\sigma} \sim \mathcal{N}(0, 1)$$

Feature Engineering & Feature Selection I

Feature Engineering

Creating informative variables using domain knowledge to simplify learning:

- Interaction terms ($X_1 \times X_2$), feature ratios, and polynomial expansions.
- Time-based features (e.g., day of week, hour, elapsed time).
- Domain transformations (e.g., TF-IDF for text, Fourier transform for signal data).

Feature Selection Techniques

Selecting an optimal subset of attributes to improve model interpretability and reduce variance:

- **Filter Methods**: Variance thresholding, Pearson correlation, ANOVA F-test, Mutual Information.
- **Wrapper Methods**: Recursive Feature Elimination (RFE), Forward/Backward selection.
- **Embedded Methods**: L_1 Regularization (Lasso), Tree-based feature importance scores.

Validation Strategies & Data Splitting I

Data Partitioning Scheme

To accurately estimate model generalization to unseen data:

- **Training Set (**$60 - 80\%$**)**: Used to estimate parameters (weights and biases).
- **Validation Set (**$10 - 20\%$**)**: Used for hyperparameter tuning and model selection.
- **Test Set (**$10 - 20\%$**)**: Held out for unbiased final performance assessment.

Cross-Validation (CV)

Mitigates sample bias inherent to a single train-test split:

- K**-Fold Cross-Validation**: Partitions dataset into K equal subsets; iterates K times training on $K - 1$ folds and evaluating on the remaining fold.
- **Stratified** K**-Fold**: Preserves class ratio distributions across every fold (critical for imbalanced tasks).

Python ML Workflow Example I

```python
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier

# Define numerical and categorical features
num_cols = ['age', 'income', 'credit_score']
cat_cols = ['education', 'occupation']

# Preprocessing pipeline
preprocessor = ColumnTransformer(transformers=[
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first'), cat_cols)
])

# Full Machine Learning Pipeline
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Cross-Validation Evaluation
# X_train, y_train defined beforehand
scores = cross_val_score(pipeline, X_train, y_train, cv=5, scoring='accuracy')
print(f"5-Fold CV Accuracy: {scores.mean():.4f} +/- {scores.std():.4f}")
```

Summary of Machine Learning Activities I

Key Lessons

- Data preparation and feature engineering comprise up to 80% of practical machine learning effort.
- **Data Leakage Caution**: Always fit scalers, encoders, and feature selectors *only* on training data folds.
- Continuous diagnostic feedback loops guide whether to gather more data, engineer new features, or alter hyperparameter space.

Next Steps in Unit 3

- **Lecture 3.2**: Model Evaluation Metrics (Confusion Matrix, Precision-Recall, ROC-AUC, MSE, MAE).
- **Lecture 3.3**: Bias-Variance Tradeoff & Hyperparameter Optimization.

Taxonomy of Data in Machine Learning I

The Fundamental Structure of Datasets

Machine learning models operate on feature vectors $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$. Understanding the mathematical nature of each attribute x_i is essential for selecting appropriate algorithms, preprocessing techniques, and distance metrics.

- **Numerical (Quantitative) Data**: Expresses quantities, measurements, or counts. Mathematical operations (addition, multiplication) are inherently meaningful.
  - **Continuous**: Real-valued measurements ($x \in \mathbb{R}$).
  - **Discrete**: Integer-valued counts ($x \in \mathbb{Z}_{\geq 0}$).
- **Categorical (Qualitative) Data**: Expresses characteristics, labels, or group memberships. Represents qualitative attributes rather than numerical magnitudes.
  - **Nominal**: Distinct categories without natural ordering.
  - **Ordinal**: Categories with a well-defined sequential order or rank.
- **Specialized Types**: Datetime series, text/embeddings, images/tensors, and graph structures.

Numerical (Quantitative) Data I

Characteristics of Numerical Features

Numerical variables represent quantifiable amounts where distance and magnitude carry concrete mathematical meaning.

- **Continuous Data**:
  - Can take any real value within a continuous interval.
  - Examples: Height (175.4 cm), House Price ($\$450,250$), Temperature ($23.6^{\circ}$C).
  - Measured using floating-point representations in computational implementations.
- **Discrete Data**:
  - Takes countable, distinct values (typically integers).
  - Examples: Number of children (2), Website clicks per minute (142), Credit transactions per day (5).
  - Expresses finite or countably infinite step values.

Numerical (Quantitative) Data II

Stevens' Levels of Measurement for Numerical Data

- **Interval Scale**: Equal differences between values, but *no true zero point* (e.g., Temperature in $^\circ$C or $^\circ$F; 0°C does not mean zero heat). Ratios are invalid (40°C is not "twice as hot" as 20°C).
- **Ratio Scale**: Equal intervals *and* a meaningful absolute zero (e.g., Income, Weight, Kelvin temperature). Ratios are valid (100 kg is twice 50 kg).

Numerical Preprocessing: Scaling & Transformations I

Why Scaling Matters

Distance-based algorithms (K-NN, SVM, K-Means) and gradient descent updates are sensitive to feature scales. Unscaled features with large ranges dominate loss functions and distance computations.

1. **Standardization (Z-score Normalization)**: Centers features to zero mean ($\mu = 0$) and unit variance ($\sigma = 1$):

$$x' = \frac{x - \mu}{\sigma}$$

   Recommended for models assuming Gaussian-distributed data (e.g., Logistic Regression, Linear Regression, PCA).

2. **Min-Max Normalization**: Rescales features linearly into a fixed bounded range $[0, 1]$:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

   Sensitive to extreme outliers which compress the standard feature range.

3. **Log Transformations**: Applies $x' = \log(x + 1)$ to handle right-skewed numerical distributions (e.g., income, house prices) and stabilize variance.

Categorical (Qualitative) Data I

Characteristics of Categorical Features

Categorical variables represent qualitative values belonging to discrete classes or categories. Direct arithmetic operations ($+$, $\times$) on string labels are undefined.

- **Nominal Data (Unordered Categories)**:
  - Categories have no intrinsic order, rank, or hierarchy.
  - Examples: Eye Color (Blue, Green, Brown), Country (USA, India, Germany), Marital Status (Single, Married).
  - Allowed operations: Equality testing ($=$ vs $\neq$). Relational comparisons ($>$, $<$) are invalid.
- **Ordinal Data (Ordered Categories)**:
  - Categories follow a clear, natural sequence or ranking, but differences between ranks are non-uniform or unquantifiable.
  - Examples: Education Level (High School ¡ Bachelor's ¡ Master's ¡ PhD), Customer Satisfaction (Low ¡ Medium ¡ High).
  - Allowed operations: Order relations ($>$, $<$, $=$). Exact mathematical differences ($X_1 - X_2$) are undefined.

Categorical Encoding Strategies I

Converting Categories to Machine Learning Input

Algorithms require numeric matrix input $\mathbf{X} \in \mathbb{R}^{n \times d}$. Categorical features must be encoded appropriately to reflect their underlying structure.

- **Ordinal Encoding**:
  - Maps each category to a unique integer preserving rank (e.g., Low $\rightarrow$ 0, Med $\rightarrow$ 1, High $\rightarrow$ 2).
  - *Warning*: Applying this to nominal data introduces false ordinal assumptions (e.g., Red= 0, Green= 1, Blue= 2 implies Blue $>$ Red).
- **One-Hot Encoding (OHE)**:
  - Expands a nominal feature with K categories into K binary columns (0 or 1).
  - **Dummy Variable Trap**: For linear models, drop one category ($K-1$ columns) to avoid perfect multicollinearity ($\sum_{i=1}^{K} D_i = 1$).
- **Target / Frequency Encoding**:
  - Replaces high-cardinality categories (e.g., ZIP codes) with target means or class frequencies to prevent extreme dimensionality explosion.

Summary Comparison of Data Types I

Comparative Matrix

Data Type	Key Feature	Valid Ops	Standard Preprocessing
Continuous	Real values $\mathbb{R}$	$+, -, \times, /$	Standardization / Min-Max
Discrete	Count integers $\mathbb{Z}_{\geq 0}$	$+, -, \times, /$	Scaling / Binning
Nominal	Unordered sets	$=, \neq$	One-Hot / Target Encoding
Ordinal	Ranked sequence	$=, \neq, <, >$	Ordinal Integer Encoding

Key ML Design Takeaways

- **Tree-based Models** (e.g., Decision Trees, Random Forests): Invariant to monotonic numeric scaling; natively handle ordinal features.
- **Linear & Distance Models** (e.g., SVM, Linear Regression, K-NN): Highly sensitive to unscaled numerical features and invalid nominal encoding.
- **Data Leakage Principle**: Compute encoding mappings and scaling parameters (μ, σ) using *only* the training set fold!

Python Implementation: Feature Preprocessing I

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder, OrdinalEncoder

# Sample dataset with mixed data types
df = pd.DataFrame({
    'age': [25, 45, 35, 50],                # Continuous numerical
    'income': [50000, 120000, 85000, 95000], # Continuous numerical
    'education': ['High-School', 'PhD', 'Bachelor', 'Master'], # Ordinal
    'city': ['NY', 'Paris', 'NY', 'Tokyo'] # Nominal
})

# 1. Numerical Scaling
scaler = StandardScaler()
df[['age_scaled', 'income_scaled']] = scaler.fit_transform(df[['age', 'income']])

# 2. Ordinal Encoding (Preserving Order)
edu_order = [['High-School', 'Bachelor', 'Master', 'PhD']]
ord_enc = OrdinalEncoder(categories=edu_order)
df['edu_encoded'] = ord_enc.fit_transform(df[['education']])

# 3. One-Hot Encoding for Nominal Data
ohe = OneHotEncoder(sparse_output=False, drop='first')
city_ohe = ohe.fit_transform(df[['city']])
print("Scaled-&-Encoded-Data-shape:", df.shape)
```

Dimensions of Data Quality in Machine Learning I

The Foundation of Robust Models

Data quality dictates the upper bound of model performance ("Garbage In, Garbage Out"). High-quality training datasets must satisfy six core dimensions before modeling.

- **Completeness**: Extent to which required values are present (lack of missing entries).
- **Accuracy**: Agreement between recorded data values and true ground-truth entities.
- **Consistency**: Equivalence of attributes across disparate data sources and schemas.
- **Validity**: Conformance to defined domain constraints, ranges, formats, and data types.
- **Uniqueness**: Absence of unintended duplicate records or redundant observations.
- **Timeliness**: Freshness and temporal relevance of the dataset relative to inference time.

Impact of Poor Data Quality

Systematic noise, unhandled missingness, and invalid attributes lead to biased parameter estimation, inflated variance, severe data leakage, and silent failures in deployment.

Taxonomy of Data Defects and Noise I

Common Data Anomalies in ML Pipelines

Real-world datasets suffer from heterogeneous defects originating during collection, logging, and storage.

1. **Structural Anomalies**:
   - Heterogeneous representations (e.g., `"NY"`, `"New York"`, `"ny"`).
   - Inconsistent temporal formats, mixed metric units, and trailing whitespace.
2. **Measurement & Sensor Noise**:
   - Gaussian/random noise added to continuous features during acquisition.
   - Miscalibrated hardware or transmission errors causing corrupted readings.
3. **Label Noise**:
   - Incorrect or ambiguous ground-truth labels assigned by human annotators.
   - Adversarial or systemic biases in historical target label assignment.

Missing Data Mechanisms: MCAR, MAR, and MNAR I

Statistical Taxonomy of Missingness

Let $Y = (Y_{\text{obs}}, Y_{\text{mis}})$ represent the dataset and M be the missingness indicator matrix. The mechanism determines the appropriate remediation strategy.

- **Missing Completely at Random (MCAR)**:

$$P(M \mid Y_{\text{obs}}, Y_{\text{mis}}) = P(M)$$

Missingness is independent of both observed and unobserved data. Unbiased under listwise deletion, but reduces statistical power.

- **Missing at Random (MAR)**:

$$P(M \mid Y_{\text{obs}}, Y_{\text{mis}}) = P(M \mid Y_{\text{obs}})$$

Missingness depends systematically on observed data, but not on the missing values themselves. Requires model-based imputation (e.g., K-NN, MICE).

- **Missing Not at Random (MNAR)**:

$$P(M \mid Y_{\text{obs}}, Y_{\text{mis}}) \text{ depends on } Y_{\text{mis}}$$

Missingness depends directly on the unobserved value. Requires modeling the missingness mechanism or explicit missing indicators (M).

Missing Value Remediation Techniques I

Imputation Strategies

- **Deletion Methods**:
  - *Listwise Deletion*: Drops rows containing missing values. Safe only under MCAR and low missingness ($< 5\%$).
  - *Feature Dropping*: Removes features exceeding high missingness thresholds ($> 40 - 50\%$).
- **Univariate Imputation**:
  - Replaces missing entries with global statistics (Mean for Gaussian, Median for skewed continuous, Mode for categorical).
- **Multivariate Imputation**:
  - *K-Nearest Neighbors (KNN Imputer)*: Imputes using weighted average of K nearest sample profiles.
  - *MICE (Iterative Imputer)*: Models each feature with missing values as a function of all other features in a round-robin chain.

Outlier Detection and Remediation Strategies I

Univariate vs. Multivariate Outlier Detection

Outliers represent extreme anomalies that distort distance computations, loss functions, and gradient steps.

- **Univariate Statistical Rules**:
  - *Z-Score Thresholding*: Identifies points with $|Z| = \left|\frac{x-\mu}{\sigma}\right| > 3$ (assumes Gaussian distribution).
  - *Interquartile Range (IQR) Rule*: Classifies values outside $[Q_1 - 1.5 \cdot \text{IQR}, Q_3 + 1.5 \cdot \text{IQR}]$ as outliers.
- **Multivariate Detection**:
  - *Isolation Forest*: Isolates anomalies by randomly selecting features and splitting values; outliers require fewer partition splits.
  - *Local Outlier Factor (LOF)*: Measures local density deviation relative to neighboring instances.

Remediation Methods

Winsorization (clipping extreme values to 1^{st} and 99^{th} percentiles), **Trimming** (removing outlier samples), or applying **Log/Box-Cox Transformations**.

Remediation for Class Imbalance I

Handling Severe Class Skew

In tasks such as fraud detection or medical diagnosis, target classes are highly imbalanced ($1:100$ or $1:1000$).

- **Resampling Techniques**:
  - *Random Undersampling*: Reduces majority class instances (risk of losing valuable information).
  - *SMOTE (Synthetic Minority Over-sampling Technique)*: Synthesizes new minority samples along feature-space lines connecting nearest minority neighbors:

$$x_{\text{new}} = x_i + \lambda(x_{\text{zi}} - x_i), \quad \lambda \sim \text{Uniform}(0, 1)$$

- **Cost-Sensitive Learning**:
  - Adjusts class weights w_c inversely proportional to class frequencies during loss calculation:

$$w_c = \frac{N}{K \cdot N_c}$$

Python Implementation: Data Remediation Pipeline I

```python
import numpy as np
import pandas as pd
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer, SimpleImputer
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import RobustScaler

# 1. Missing Value Remediation via MICE (IterativeImputer)
mice_imputer = IterativeImputer(max_iter=10, random_state=42)
X_imputed = mice_imputer.fit_transform(X_raw)

# 2. Outlier Detection via Isolation Forest
iso_forest = IsolationForest(contamination=0.05, random_state=42)
outlier_mask = iso_forest.fit_predict(X_imputed) # -1 for outliers, 1 for inliers

# Filter out identified anomalies
X_clean = X_imputed[outlier_mask == 1]
y_clean = y_raw[outlier_mask == 1]

# 3. Robust Scaling to attenuate remaining heavy tails
scaler = RobustScaler()
X_scaled = scaler.fit_transform(X_clean)

print(f"Original shape: {X_raw.shape}, Cleaned shape: {X_scaled.shape}")
```

Summary and Best Practices I

Key Takeaways

- Data quality assessment must precede feature engineering and model training.
- Match the missingness mechanism (MCAR, MAR, MNAR) to the appropriate remediation strategy.
- Combine statistical outlier detection with robust scaling techniques to guard distance metrics.

Critical Rules for ML Pipelines

- **Prevent Data Leakage**: Compute all imputation parameters (mean, median, MICE models) and scaling factors *strictly on training sets*, then apply them to validation/test sets.
- **Continuous Monitoring**: Track data quality metrics continuously post-deployment to identify data drift and pipeline failures early.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 3**: Preparing to Model and Evaluation
- **Topic**: 3.4 Data Pre-Processing: Dimensionality reduction, Feature subset selection

Learning Objectives:

1. Understand the **Curse of Dimensionality** and its computational and mathematical impact on ML models.
2. Master **Feature Extraction** techniques (PCA, LDA) to project high-dimensional data into lower-dimensional sub-spaces.
3. Evaluate **Feature Subset Selection** paradigms: Filter, Wrapper, and Embedded methods.
4. Implement dimensionality reduction and feature selection pipelines using Python and `scikit-learn`.

The Curse of Dimensionality I

What is the Curse of Dimensionality?

As the number of features (dimensions d) increases, the volume of the feature space grows exponentially, causing the available data points to become extremely sparse.

- **Geometric Sparsity**: In high dimensions, almost all volume of a hypercube is concentrated in its corners, and data points lie near the boundary.
- **Distance Concentration Phenomenon**:

$$\lim_{d \to \infty} \frac{\text{Dist}_{\text{max}} - \text{Dist}_{\text{min}}}{\text{Dist}_{\text{min}}} \to 0$$

  Euclidean distance loses its discriminative power because all points become nearly equidistant.
- **Overfitting Risk**: High capacity for noise fitting when $d \gg N$ (d features relative to N samples).
- **Computational Burden**: Matrix inversions and distance calculations scale as $O(d^3)$ or $O(N \cdot d^2)$.

Dimensionality Reduction: Extraction vs. Selection I

Taxonomy of Dimensionality Reduction

Reducing feature space dimension $d \to k$ ($k \ll d$) can be achieved via two fundamental paradigms:

1. Feature Extraction (Projection)

Creates a new set of k composite features by combining original variables:

$$z_j = f(x_1, x_2, \ldots, x_d)$$

- **Pros**: Captures global structure, compresses information efficiently.
- **Cons**: Loss of original feature interpretability.
- **Examples**: PCA, LDA, t-SNE, UMAP.

2. Feature Subset Selection

Selects a subset of k features directly from the original d attributes:

$$S \subset \{x_1, x_2, \ldots, x_d\}, \quad |S| = k$$

- **Pros**: Retains original domain semantics and interpretability.
- **Cons**: Ignores complex non-linear feature combinations.
- **Examples**: Filter, Wrapper, Embedded.

Principal Component Analysis (PCA): Theory I

Unsupervised Linear Feature Extraction

PCA finds orthogonal axes (Principal Components) that maximize data variance or, equivalently, minimize reconstruction mean squared error.

1. **Mean Centering**: Center dataset $X \in \mathbb{R}^{N \times d}$ such that $\mu = 0$.
2. **Covariance Matrix Computation**:

$$\mathbf{\Sigma} = \frac{1}{N-1} X^T X \in \mathbb{R}^{d \times d}$$

3. **Eigen-Decomposition**: Compute eigenvectors v_i and eigenvalues λ_i:

$$\mathbf{\Sigma} v_i = \lambda_i v_i, \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$$

4. **Projection**: Matrix transformation to top k principal directions $W_k = [v_1, \ldots, v_k]$:

$$Z = XW_k \in \mathbb{R}^{N \times k}$$

PCA: Explained Variance & Singular Value Decomposition I

Explained Variance Ratio (EVR)

The proportion of total dataset variance retained by the i-th principal component:

$$\text{EVR}_i = \frac{\lambda_i}{\sum_{j=1}^{d} \lambda_j}, \quad \text{Cumulative Variance} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{d} \lambda_j}$$

Rule of Thumb: Retain enough components k to preserve $85\% - 95\%$ cumulative variance.

SVD Implementation (Numerical Efficiency)

In practice, PCA uses Singular Value Decomposition on centered matrix X:

$$X = U\mathbf{\Sigma}V^T$$

- Right singular vectors V are the principal components (eigenvectors of $X^T X$).
- Singular values s_i relate to eigenvalues via $\lambda_i = \frac{s_i^2}{N-1}$.
- Avoids computing explicit $d \times d$ covariance matrix $\mathbf{\Sigma}$.

Linear Discriminant Analysis (LDA) I

Supervised Linear Dimension Reduction

Unlike PCA (unsupervised variance maximization), LDA projects data to maximize class separability by finding directions that maximize the ratio of between-class variance to within-class variance.

- **Within-Class Scatter Matrix (S_W)**:

$$S_W = \sum_{c=1}^{C} \sum_{x \in C_c} (x - \mu_c)(x - \mu_c)^T$$

- **Between-Class Scatter Matrix (S_B)**:

$$S_B = \sum_{c=1}^{C} N_c (\mu_c - \mu)(\mu_c - \mu)^T$$

- **Fisher's Criterion Optimization**:

$$J(w) = \arg\max_w \frac{w^T S_B w}{w^T S_W w} \implies S_W^{-1} S_B w = \lambda w$$

- **Dimensional Constraint**: Maximum $k = \min(d, C - 1)$ components, where C is the number of classes.

Python Example: PCA and LDA with Scikit-Learn I

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.preprocessing import StandardScaler

# 1. Load and standard scale features
X, y = load_iris(return_X_y=True)
X_scaled = StandardScaler().fit_transform(X)

# 2. PCA: Unsupervised (Target y is NOT used)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
print(f"PCA Explained Variance Ratio: {pca.explained_variance_ratio_}")
# Output: [0.72962145 0.22850762] -> Preserves 95.8% variance

# 3. LDA: Supervised (Target y IS required)
lda = LDA(n_components=2)
X_lda = lda.fit_transform(X_scaled, y)
print(f"LDA Explained Variance Ratio: {lda.explained_variance_ratio_}")
# Output: [0.9912126  0.0087874 ] -> Maximizes class separation
```

Feature Subset Selection: Filter Methods I

Core Principle

Filter methods evaluate individual features based on statistical properties independent of any machine learning model.

- **Variance Thresholding**: Removes features with variance below a specified threshold ($\sigma^2 < \epsilon$). Extremely useful for removing constant or near-constant features.
- **Pearson Correlation Coefficient (r)**:

$$r(X_j, Y) = \frac{\text{Cov}(X_j, Y)}{\sigma_{X_j} \sigma_Y}$$

  Filters features highly correlated with target Y while eliminating collinear features ($r(X_i, X_j) > 0.9$).

- **Statistical Tests**:
  - **ANOVA F-test**: Measures linear dependency between numerical features and categorical target.
  - **Chi-Square (χ^2)**: Evaluates independence between categorical features and categorical target.
  - **Mutual Information (MI)**: Non-parametric score capturing non-linear relationships:

$$I(X; Y) = \iint p(x, y) \log \frac{p(x, y)}{p(x)p(y)} dxdy$$

Feature Subset Selection: Wrapper Methods I

Core Principle

Wrapper methods use a specific ML algorithm as a black-box evaluator to search the space of feature subsets based on predictive performance.

1. **Sequential Forward Selection (SFS)**:
   - Starts with empty feature set $\emptyset$. Iteratively adds feature that maximizes model score.
2. **Sequential Backward Elimination (SBE)**:
   - Starts with full feature set F. Iteratively removes feature whose removal causes smallest drop in performance.
3. **Recursive Feature Elimination (RFE)**:
   - Trains model on current features, ranks feature importance (e.g. coefficients w_j or tree importances), prunes lowest rank features, and repeats.

Trade-offs

Pros: Evaluates feature interactions; tailored to specific model.
Cons: High computational complexity $O(2^d)$; prone to overfitting.

Feature Subset Selection: Embedded Methods I

Core Principle

Embedded methods perform feature selection naturally as part of the model learning/optimization process.

- L_1 **Regularization (Lasso Regression)**: Adds penalty term proportional to absolute sum of weights:

$$\min_w \frac{1}{2N} \|y - Xw\|_2^2 + \alpha \|w\|_1$$

Due to geometry of L_1 norm diamond constraint, coefficients of irrelevant features are driven to **exactly zero**, performing automated variable selection.

- **Tree-Based Feature Importance**:
  - **Mean Decrease in Impurity (MDI)**: Measures cumulative reduction of Gini impurity or Entropy brought by splitting on feature X_j across all trees in Random Forest/GBDT.
  - Thresholding feature importance scores selects informative features.

Python Example: Feature Subset Selection I

```python
from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectKBest, f_classif, RFE
from sklearn.linear_model import LogisticRegression, LassoCV

X, y = make_classification(n_samples=200, n_features=20, n_informative=5, random_state=42)

# 1. Filter Method: Select top 5 features using ANOVA F-test
selector_filter = SelectKBest(score_func=f_classif, k=5)
X_filter = selector_filter.fit_transform(X, y)

# 2. Wrapper Method: Recursive Feature Elimination (RFE)
model = LogisticRegression()
rfe = RFE(estimator=model, n_features_to_select=5)
X_rfe = rfe.fit_transform(X, y)

# 3. Embedded Method: Lasso (L1 regularization)
lasso = LassoCV(cv=5).fit(X, y)
selected_mask = (lasso.coef_ != 0)
print("Non-zero coefficients count:", selected_mask.sum())
```

Comparison of Dimensionality Reduction Methods I

Method	Type	Supervised?	Preserves Semantics?	Computational Cost
PCA	Extraction	No	No (Linear combinations)	Low ($O(d^3)$ or SVD)
LDA	Extraction	Yes	No (Linear combinations)	Low ($O(d^3)$)
Filter	Selection	Varies	Yes (Original features)	Very Low ($O(d)$)
Wrapper	Selection	Yes	Yes (Original features)	High ($O(k \cdot \text{Train})$)
Embedded	Selection	Yes	Yes (Original features)	Medium (Integrated)

Practical Recommendation Matrix

- High d, linear model, tabular interpretability required $\rightarrow$ **Lasso / Filter (MI)**.
- Multi-class classification, low-dim representation $\rightarrow$ **LDA**.
- Multicollinearity reduction, dense continuous features $\rightarrow$ **PCA**.
- Maximum predictive power, moderate $d \leq 50$ $\rightarrow$ **RFE (Wrapper)**.

Predictive vs. Descriptive Modeling Overview I

Taxonomy of Machine Learning Models

Model selection begins by identifying whether the primary operational objective is **prediction** of unobserved outcomes or **description** of underlying data structures.

Predictive Modeling

- **Objective**: Forecast target label Y given feature vector X.
- **Supervision**: Requires labelled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.
- **Focus**: Generalization performance on unseen data.
- **Examples**: Spam detection, price forecasting, disease diagnosis.

Descriptive Modeling

- **Objective**: Discover patterns, groupings, or representations in X.
- **Supervision**: Operates on unlabelled data $\mathcal{D} = \{x_i\}_{i=1}^{N}$.
- **Focus**: Interpretability, actionable insights, structure.
- **Examples**: Customer segmentation, topic modeling, anomaly detection.

Mathematical Foundations of Predictive Modeling I

Formal Problem Setup

Given an input space $\mathcal{X} \subset \mathbb{R}^d$ and output space $\mathcal{Y}$, learn a mapping function $f : \mathcal{X} \to \mathcal{Y}$ minimizing expected risk:

$$R(f) = \mathbb{E}_{(X,Y)\sim P}[L(Y, f(X))] = \int_{\mathcal{X}\times\mathcal{Y}} L(y, f(x))\, dP(x, y)$$

where $L(y, \hat{y})$ is a task-specific loss function (e.g., Mean Squared Error or Cross-Entropy).

Key Characteristics

- **Empirical Risk Minimization (ERM)**: Approximates $R(f)$ using training dataset $\mathcal{D}_{\text{train}}$:

$$\hat{f} = \arg\min_{f\in\mathcal{H}} \frac{1}{N}\sum_{i=1}^{N} L(y_i, f(x_i)) + \lambda\Omega(f)$$

- **Out-of-Sample Validation**: Evaluated rigorously on holdout/test sets to detect overfitting.
- **Trade-off**: Often trades mathematical interpretability for superior predictive accuracy (e.g., Deep Learning, Gradient Boosting).

Structure Discovery in Descriptive Modeling I

Formal Problem Setup

Given unlabelled sample vectors $\{x_1, x_2, \ldots, x_N\} \subset \mathbb{R}^d$, fit a model $g(X)$ to estimate joint probability density $p(X)$, project dimensions, or partition space into K latent clusters:

$$\mathcal{C} = \{C_1, C_2, \ldots, C_K\}, \quad \text{where} \bigcup_{k=1}^{K} C_k = \mathcal{D} \quad \text{and} \quad C_i \cap C_j = \emptyset \ (\forall i \neq j)$$

Primary Tasks & Objectives

- **Clustering**: Partitioning instances based on similarity metrics (e.g., K-Means, Hierarchical, DBSCAN).
- **Dimensionality Reduction**: Projecting high-dimensional data onto lower-dimensional manifolds while preserving variance or distance (e.g., PCA, t-SNE, UMAP).
- **Association Rule Mining**: Uncovering co-occurrence relationships ($A \implies B$) evaluated via Support, Confidence, and Lift.
- **Density Estimation**: Modeling $p(x)$ directly (e.g., Gaussian Mixture Models, Kernel Density Estimation).

Predictive vs. Descriptive Modeling Comparison I

Side-by-Side Comparison

Dimension	Predictive Modeling	Descriptive Modeling
Primary Goal	Predict target outcome Y	Describe latent patterns in X
Target Label	Mandatory ($y_i \in \mathcal{Y}$)	Absent / Unsupervised
Validation	Ground-truth metrics (R^2, F1, ROC-AUC)	Internal metrics (Silhouette, Inertia, AIC/BIC)
Overfitting Risk	High risk; guarded by test splits & regularization	Evaluated via stability & domain plausibility
Algorithms	Linear/Logistic Reg., Random Forest, XGBoost, Neural Nets	K-Means, PCA, GMM, Apriori, Hierarchical Clustering

Model Selection Framework & Complexity Penalties I

Occam's Razor in Model Selection

When multiple candidate models perform similarly, prefer the simplest model with fewer parameters to maximize interpretability and reduce generalization error.

Akaike Information Criterion (AIC)

Penalizes model complexity based on parameter count k and log-likelihood $\hat{L}$:

$$\text{AIC} = 2k - 2\ln(\hat{L})$$

- Lower AIC indicates a superior balance between fit and parsimony.

Bayesian Information Criterion (BIC)

Applies a stronger penalty for large sample size N:

$$\text{BIC} = k\ln(N) - 2\ln(\hat{L})$$

- Heavily penalizes complex models as N increases.

Hybrid Paradigms: Synergy of Both Approaches I

Integrated Workflows in Practical Data Science

In real-world engineering, predictive and descriptive modeling are frequently combined to form hybrid pipelines:

1. **Descriptive Preprocessing for Predictive Tasks**:
   - Using *Principal Component Analysis (PCA)* to reduce multicollinearity before training a regression model.
   - Applying *K-Means Clustering* to generate cluster membership IDs as categorical features for a gradient boosted tree.
2. **Predictive Models for Descriptive Insights**:
   - Inspecting *Feature Importance* or *SHAP (SHapley Additive exPlanations)* values from a random forest to understand physical relationships.
   - Autoencoders (Predictive reconstruction loss) used for *Anomaly Detection* (Descriptive structure analysis).

Python Implementation: Predictive vs. Descriptive Pipelines I

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, silhouette_score

# Generate synthetic dataset
X, y = make_classification(n_samples=500, n_features=10,
                           n_informative=5, random_state=42)

# --- 1. Predictive Model (Supervised Classification) ---
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=42)
pred_model = RandomForestClassifier(n_estimators=50, random_state=42)
pred_model.fit(X_tr, y_tr)
y_pred = pred_model.predict(X_te)
print(f"Predictive Model Accuracy: {accuracy_score(y_te, y_pred):.4f}")

# --- 2. Descriptive Model (Unsupervised Clustering) ---
desc_model = KMeans(n_clusters=2, random_state=42, n_init=10)
cluster_labels = desc_model.fit_predict(X)
sil_score = silhouette_score(X, cluster_labels)
print(f"Descriptive Silhouette Score: {sil_score:.4f}")
```

Summary: Model Selection Guidelines I

Decision Matrix for Model Selection

- **Choose Predictive Modeling when**:
  - Ground-truth labels Y are available and target accuracy is paramount.
  - The primary goal is automated decision-making on incoming unseen instances.
- **Choose Descriptive Modeling when**:
  - Data is unlabelled and objective is exploratory analysis or pattern discovery.
  - Understanding underlying domain relationships takes precedence over precise prediction.

Key Takeaway

Model selection should be dictated by the **business/scientific question**, **data availability**, and the required balance between **interpretability** and **generalization performance**.

Supervised Model Training Workflow I

The Fundamental Goal

In supervised learning, the objective is to learn a mapping function $f : \mathcal{X} \to \mathcal{Y}$ using empirical data such that it generalizes accurately to unseen samples.

The Pitfall of Resubstitution Error

Evaluating a model on the exact same data used to train it leads to **optimistic bias** and fails to detect **overfitting**.

- **Training Error (E_{train}):** Empirical loss computed on training samples.
- **Generalization Error (E_{test}):** Expected loss on new, unseen samples drawn from the true underlying distribution $P(X, Y)$.
- **Data Partitioning:** Disjoint splitting of datasets to simulate evaluation on unseen data.

The Holdout Method I

Data Partitioning Strategy

The dataset $\mathcal{D}$ is randomly partitioned into two or three non-overlapping subsets:

- **Training Set ($\mathcal{D}_{\text{train}}$):** Used to learn parameters θ (typically 60% – 80%).
- **Validation Set ($\mathcal{D}_{\text{val}}$):** Used for hyperparameter tuning and model selection (10% – 20%).
- **Test Set ($\mathcal{D}_{\text{test}}$):** Reserved strictly for final, unbiased evaluation (10% – 20%).

Advantages

- Computationally efficient ($O(1)$ training runs).
- Simple and intuitive implementation.

Disadvantages

- High variance depending on split.
- Reduces data available for training.

Holdout Method in Python I

Implementation using Scikit-Learn

Demonstrating train-validation-test split with target stratification:

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

# Load dataset
X, y = load_iris(return_X_y=True)

# First split: Train+Val (80%) and Test (20%)
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42, stratify=y
)

# Second split: Train (60% total) and Val (20% total)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.25, random_state=42, stratify=y_temp
)

print(f"Train: {X_train.shape[0]}, Val: {X_val.shape[0]}, Test: {X_test.shape[0]}")
```

K-Fold Cross-Validation I

Core Mechanism

K-Fold Cross-Validation mitigates split dependency by evaluating model performance across K disjoint validation folds.

1. Randomly partition dataset $\mathcal{D}$ into K equal-sized subsets: $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$.
2. For each fold $k \in \{1, 2, \ldots, K\}$:
   - Train model on $\mathcal{D} \setminus \mathcal{D}_k$.
   - Compute validation error metric E_k on fold $\mathcal{D}_k$.
3. Calculate aggregated metric score:

$$\bar{E} = \frac{1}{K} \sum_{k=1}^{K} E_k$$

Key Benefit

Every data point is used exactly once for validation and $K - 1$ times for model training.

Variations of Cross-Validation I

Stratified K-Fold Cross-Validation

- Preserves target class distribution proportions across all K folds.
- Essential for imbalanced classification to avoid fold bias or empty class representation.

Leave-One-Out Cross-Validation (LOOCV)

- Extreme case of K-Fold CV where $K = N$ (N is dataset size).
- In each fold, $N - 1$ samples are used for training, 1 sample for validation.
- **Pros:** Low estimation bias; deterministic (no random seed variance).
- **Cons:** High computational overhead (N model fits); high error variance.

Scikit-Learn Implementation

Comparing standard K-Fold and Stratified K-Fold cross-validation:

```
import numpy as np
from sklearn.model_selection import KFold, StratifiedKFold, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_breast_cancer

X, y = load_breast_cancer(return_X_y=True)
clf = LogisticRegression(max_iter=10000)

# 5-Fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores_kf = cross_val_score(clf, X, y, cv=kf, scoring='accuracy')

# 5-Fold Stratified Cross Validation
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores_skf = cross_val_score(clf, X, y, cv=skf, scoring='accuracy')

print(f"K-Fold Accuracy: {np.mean(scores_kf):.4f} +/- {np.std(scores_kf):.4f}")
print(f"Stratified K-Fold: {np.mean(scores_skf):.4f} +/- {np.std(scores_skf):.4f}")
```

Comparison & Practical Guidelines I

Trade-off Summary

Method	Compute Cost	Bias	Variance
Holdout Split	Low ($O(1)$)	High	High
K-Fold CV ($K = 5, 10$)	Moderate ($O(K)$)	Low	Moderate
Stratified K-Fold	Moderate ($O(K)$)	Low	Low-Moderate
LOOCV ($K = N$)	High ($O(N)$)	Very Low	High

Best Practices

- Use **Holdout** for massive datasets (e.g., Deep Learning) where retraining is expensive.
- Standardize on **Stratified 5- or 10-Fold CV** for small-to-medium tabular datasets.
- **Prevent Data Leakage:** Perform preprocessing (e.g., feature scaling, imputation) strictly inside each cross-validation fold using pipelines.

Introduction to Performance Evaluation I

Why Simple Accuracy is Often Not Enough

- Classification accuracy measures the proportion of correct predictions:

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

- **Class Imbalance Problem:** In fraud detection (99% legitimate, 1% fraud), a naive model predicting all transactions as "legitimate" achieves 99% accuracy while failing completely.
- **Asymmetric Misclassification Costs:** Failing to detect a disease (False Negative) is often far more critical than a false alarm (False Positive).

The Solution

A **Confusion Matrix** breaks down predictions into correct and incorrect classifications for each specific class, revealing hidden performance flaws.

Structure of a Binary Confusion Matrix I

A 2×2 table comparing **Actual Ground Truth** against **Predicted Classes**:

		Predicted Class	
		Positive (1)	**Negative (0)**
2*Actual Class	**Positive (1)**	**True Positive (TP)** *(Hit / Correct)*	**False Negative (FN)** *(Type II Error / Miss)*
	Negative (0)	**False Positive (FP)** *(Type I Error / False Alarm)*	**True Negative (TN)** *(Correct Rejection)*

- **Actual Condition:** Rows represent true labels.
- **Predicted Condition:** Columns represent model output predictions.
- **Total Samples:** $N = \text{TP} + \text{TN} + \text{FP} + \text{FN}$.

Understanding the Four Core Outcomes I

- **True Positive (TP):**
  - Actual = Positive, Predicted = Positive.
  - *Example:* A sick patient correctly identified as having the illness.
- **True Negative (TN):**
  - Actual = Negative, Predicted = Negative.
  - *Example:* A healthy individual correctly identified as healthy.
- **False Positive (FP) — Type I Error:**
  - Actual = Negative, Predicted = Positive.
  - *Example:* Healthy patient falsely diagnosed with illness (False Alarm).
- **False Negative (FN) — Type II Error:**
  - Actual = Positive, Predicted = Negative.
  - *Example:* Sick patient incorrectly declared healthy (Missed Detection).

Basic Derived Metrics: Accuracy & Error Rate I

Classification Accuracy

Measures the overall fraction of correct predictions across all classes:

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

Classification Error Rate (Misclassification Rate)

Measures the overall proportion of incorrect predictions:

$$\text{Error Rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 1 - \text{Accuracy}$$

Caution

Accuracy is reliable ONLY when class distributions are balanced and error costs are symmetric across classes.

Precision and Recall (Sensitivity) I

Precision (Positive Predictive Value)

Out of all instances predicted as Positive, how many were actually Positive?

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

- Goal: Minimize **False Positives**.
- Critical in spam filtering, search results.

Recall (Sensitivity / True Positive Rate)

Out of all actual Positive instances, how many did the model correctly catch?

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

- Goal: Minimize **False Negatives**.
- Critical in medical diagnosis, fraud detection.

Specificity and F1-Score I

Specificity (True Negative Rate)

Out of all actual Negative instances, how many were correctly identified as Negative?

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

- False Positive Rate (FPR) $= 1 - \text{Specificity} = \frac{\text{FP}}{\text{TN}+\text{FP}}$.

F1-Score (Harmonic Mean)

Combines Precision and Recall into a single metric:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}$$

- Penalizes extreme imbalances between Precision and Recall.
- Essential metric for evaluating imbalanced datasets.

Multi-Class Confusion Matrix Extension I

For a classification problem with $K > 2$ classes (e.g., $K = 3$: Class A, B, C):

		Predicted Class		
		Class A	**Class B**	**Class C**
3***Actual Class**	**Class A**	C_{AA} (TP_A)	C_{AB}	C_{AC}
	Class B	C_{BA}	C_{BB} (TP_B)	C_{BC}
	Class C	C_{CA}	C_{CB}	C_{CC} (TP_C)

- **Diagonal Elements (C_{ii}):** Correct classifications for each class.
- **Off-Diagonal Elements ($C_{ij}, i \neq j$):** Misclassifications where actual class i is predicted as class j.
- **Per-Class Evaluation:** Precision and Recall can be computed using One-vs-Rest strategy.

Python Implementation: Confusion Matrix I

```python
import matplotlib.pyplot as plt
from sklearn.metrics import (
    confusion_matrix, ConfusionMatrixDisplay,
    classification_report, accuracy_score
)

# Ground truth labels and predictions
y_true = [1, 0, 1, 1, 0, 1, 0, 0, 1, 0]
y_pred = [1, 0, 1, 0, 0, 1, 1, 0, 1, 0]

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred)
tn, fp, fn, tp = cm.ravel()

print(f"TP:{tp},TN:{tn},FP:{fp},FN:{fn}")

# Detailed metrics summary report
print(classification_report(y_true, y_pred, target_names=['Neg', 'Pos']))

# Visual plot of the Confusion Matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Neg', 'Pos'])
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix Evaluation")
```

Summary: Choosing the Right Metric I

Scenario / Objective	Key Metric	Focus Area
Balanced classes, equal costs	Accuracy	Overall Correctness
Minimize False Alarms (FP)	Precision	Quality of Positives
Minimize Missed Detections (FN)	Recall	Quantity of Positives
Imbalanced datasets	F1-Score	Harmonic Balance of P & R

Key Takeaways

- Never rely on Accuracy alone on imbalanced datasets.
- Always analyze the **Confusion Matrix** to understand specific error distributions.
- Select evaluation metrics based on the real-world business/domain cost of errors.

Improving Model Performance: Overview I

Why Isn't the Baseline Model Enough?

Initial baseline models often suffer from underfitting (high bias), overfitting (high variance), or data-related issues such as class imbalance and uninformative features.

Model-Centric Strategies

- **Hyperparameter Tuning**: Grid Search, Random Search, Bayesian Optimization.
- **Regularization**: L_1/L_2 penalties, Dropout, Early Stopping.
- **Ensembling**: Bagging, Boosting, Stacking.

Data-Centric Strategies

- **Class Imbalance**: SMOTE, undersampling, cost-sensitive learning.
- **Feature Engineering**: Selection, transformations, interactions.
- **Data Cleaning**: Outlier handling, missing data imputations.

Hyperparameter Optimization I

Optimization Strategies

Grid Search Exhaustively evaluates all parameter combinations over a defined grid. Guarantees finding the best grid point but scales poorly: $O(\prod k_i)$.

Random Search Samples combinations randomly from specified statistical distributions. Highly efficient in high-dimensional hyperparameter spaces.

Bayesian Optimization Constructs a probabilistic surrogate model (e.g., Gaussian Process) to balance exploration and exploitation of the search space.

Validation Leakage Warning

Over-tuning hyperparameters on the validation set can cause **validation set overfitting**. Always preserve a separate test set for final evaluation.

Code: Hyperparameter Tuning in Scikit-Learn I

```python
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

# 1. Define hyperparameter distribution search space
param_dist = {
    'n_estimators': randint(50, 300),
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': randint(2, 11)
}

# 2. Configure Randomized Search Cross-Validation
rf = RandomForestClassifier(random_state=42)
rand_search = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist,
    n_iter=20, cv=5, scoring='f1_macro', n_jobs=-1
)
rand_search.fit(X_train, y_train)

print(f"Best Hyperparameters: {rand_search.best_params_}")
print(f"Best Validation Score: {rand_search.best_score_:.4f}")
```

Handling Class Imbalance I

The Imbalance Challenge

When target classes are skewed (e.g., 99% negative vs. 1% positive), standard loss functions default to predicting majority classes, yielding high accuracy but near-zero recall.

Resampling Strategies

- **Oversampling**: Duplicates minority instances (risk of overfitting).
- **Undersampling**: Discards majority instances (loss of information).
- **SMOTE**: Synthetic Minority Over-sampling Technique via k-NN interpolation.

Algorithmic Adjustments

- **Cost-Sensitive Learning**: Scale loss via inverse class frequencies $w_c = \frac{N}{K \cdot N_c}$.
- **Threshold Tuning**: Shift decision threshold off 0.5 using Precision-Recall curves.

Code: Addressing Class Imbalance I

```python
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Method A: Cost-Sensitive Learning via Class Weights
model_weighted = LogisticRegression(class_weight='balanced')
model_weighted.fit(X_train, y_train)

# Method B: SMOTE Resampling (fit only on training set!)
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_train, y_train)

model_smote = LogisticRegression()
model_smote.fit(X_res, y_res)

# Evaluation on pristine validation set
y_pred = model_smote.predict(X_val)
print(classification_report(y_val, y_pred))
```

Ensemble Learning Methods I

Combining Base Learners for Superior Performance

1. **Bagging (Bootstrap Aggregating)**: Trains independent parallel estimators on bootstrap samples and averages predictions. Primary goal: **Reduce Variance**. *Example*: Random Forest.
2. **Boosting**: Sequential ensemble where subsequent models fit residual errors of prior estimators. Primary goal: **Reduce Bias**. *Examples*: AdaBoost, Gradient Boosting, XGBoost, LightGBM.
3. **Stacking**: Trains heterogeneous base models and feeds their predictions into a meta-learner.

Code: Boosting and Stacking Ensembles I

```
from sklearn.ensemble import GradientBoostingClassifier, StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

# 1. Gradient Boosting Classifier
gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1)
gb_model.fit(X_train, y_train)

# 2. Heterogeneous Stacking Ensemble
base_estimators = [
    ('dt', DecisionTreeClassifier(max_depth=4)),
    ('svc', SVC(probability=True))
]
stack_model = StackingClassifier(
    estimators=base_estimators,
    final_estimator=LogisticRegression()
)
stack_model.fit(X_train, y_train)
```

Feature Selection & Refinement I

Feature Selection Taxonomy

Filter Methods	Evaluate feature subsets using statistical properties independent of the model (χ^2, Mutual Information, ANOVA). Fast and scalable.
Wrapper Methods	Use predictive performance of a target model to evaluate feature subsets (Recursive Feature Elimination - RFE). Computationally intensive.
Embedded Methods	Perform feature selection naturally during training (L_1 Lasso regularization, Tree feature importances).

Feature Transformation Best Practices

- Log/Power transformations for highly skewed feature distributions.
- Engineering domain-specific interactions and ratios ($X_i \cdot X_j, X_i / X_j$).

Summary: Systematic Performance Improvement Workflow I

Iterative Model Improvement Recipe

1. **Establish Baseline**: Simple model + default parameters + task-appropriate metric (F1, ROC-AUC).
2. **Diagnose Error Patterns**: Inspect confusion matrix, learning curves, and residuals (bias vs. variance).
3. **Data Feature Engineering**: Address imbalance (SMOTE/weights), handle outliers, select relevant features.
4. **Explore Models**: Benchmark diverse model families (Linear, Tree-based, Neural).
5. **Hyperparameter Tuning**: Apply Random/Bayesian Search to top-performing models.
6. **Ensemble Predictions**: Combine top models via Stacking/Blending for maximum predictive performance.

Comprehensive Framework for Model Evaluation I

Beyond Classification Accuracy

While accuracy is intuitive, it can be deeply misleading in real-world applications (e.g., imbalanced datasets where 99% of samples belong to the majority class).

- **Precision**: Measure of exactness — proportion of positive identifications that were correct:

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

- **Recall (Sensitivity)**: Measure of completeness — proportion of actual positives correctly identified:

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

- F_1-**Score**: Harmonic mean of precision and recall, balancing both metrics:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

ROC and Precision-Recall Curves I

Receiver Operating Characteristic (ROC) Curve

Plots the True Positive Rate (TPR) against the False Positive Rate (FPR) across varying classification thresholds:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

- **Area Under Curve (AUC-ROC)**: Quantifies overall ranking performance (0.5 = random guess, 1.0 = perfect classifier).

Precision-Recall (PR) Curve

Recommended for highly skewed/imbalanced datasets where negative samples dominate:

- Focuses strictly on the positive class without being inflated by a large number of True Negatives.
- Area Under PR Curve (PR-AUC) evaluates performance under heavy class imbalance.

Model Diagnostics: Bias vs. Variance I

The Bias-Variance Decomposition

Total expected generalization error decomposes into three fundamental components:

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

- **High Bias (Underfitting)**:
  - Model is overly simple and fails to capture underlying patterns.
  - *Symptoms*: High training error and high validation error.
  - *Remedies*: Add more features, decrease regularization, use a more complex model.
- **High Variance (Overfitting)**:
  - Model memorizes training noise and fails to generalize.
  - *Symptoms*: Low training error but high validation error.
  - *Remedies*: Gather more data, select feature subset, increase regularization.

Hyperparameter Optimization Strategies I

Automated Model Tuning

Hyperparameters govern algorithm behavior and model capacity. They are set prior to training rather than learned from data.

1. **Grid Search CV**:
   - Exhaustively evaluates all Cartesian combinations of specified hyperparameter values.
   - Guaranteed to find the best configuration in grid, but computationally expensive.
2. **Randomized Search CV**:
   - Samples a fixed number of parameter combinations randomly from specified distributions.
   - Drastically reduces runtime while exploring high-dimensional search spaces effectively.
3. **Bayesian Optimization**:
   - Uses probabilistic surrogate models (e.g., Gaussian Processes) to sample promising hyperparameter regions sequentially.

Ensemble Techniques for Performance Improvement I

Combining Weak Learners into Strong Models

Ensembling aggregates predictions from multiple base models to reduce variance, bias, or both.

- **Bagging (Bootstrap Aggregating)**:
  - Trains base models independently in parallel on bootstrap samples of training data.
  - *Primary Goal*: Variance reduction (e.g., Random Forests).
- **Boosting**:
  - Trains base models sequentially, each focusing on correcting errors made by previous iterations.
  - *Primary Goal*: Bias and variance reduction (e.g., AdaBoost, XGBoost, LightGBM).
- **Stacking**:
  - Combines heterogeneous base classifiers via a meta-learner trained on out-of-fold predictions.

Python Pipeline: Grid Search & Evaluation I

```python
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold
from sklearn.metrics import classification_report, roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# 1. Pipeline Definition
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestClassifier(random_state=42))
])

# 2. Hyperparameter Search Grid
param_grid = {
    'rf__n_estimators': [50, 100, 200],
    'rf__max_depth': [None, 10, 20],
    'rf__min_samples_split': [2, 5]
}

# 3. Grid Search with Stratified Cross-Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(pipe, param_grid, cv=cv, scoring='roc_auc', n_jobs=-1)
grid_search.fit(X_train, y_train)

# 4. Evaluation on Held-Out Test Set
best_model = grid_search.best_estimator_
y_pred_proba = best_model.predict_proba(X_test)[:, 1]
```

Python Pipeline: Grid Search & Evaluation II

```
print(f"Best ROC-AUC: {roc_auc_score(y_test, y_pred_proba):.4f}")
```

Unit 3 Review & End-to-End Workflow Checklist I

Preparing to Model & Evaluation Checklist

1. **Data Cleaning & Remediation**: Impute missingness, mitigate outliers, handle noise.
2. **Preprocessing**: Scale numerical attributes, encode categorical variables properly.
3. **Validation Protocol**: Establish strict train/validation/test split or Stratified K-Fold CV.
4. **Metric Alignment**: Choose evaluation metrics aligned with business objectives (F_1, ROC-AUC, MAE).
5. **Diagnostic Check**: Plot learning curves to distinguish underfitting vs. overfitting.
6. **Optimization**: Apply hyperparameter tuning and ensemble techniques for performance boosts.

Golden Rule of Model Building

Prevent Data Leakage: Always fit transformers, scalers, and encoders strictly on the training partition within cross-validation loops!

Introduction to Supervised Learning I

What is Supervised Learning?

Supervised learning is a machine learning paradigm where an algorithm learns a mapping function $f : \mathcal{X} \to \mathcal{Y}$ from a labeled training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$.

Core Goal

Given a new, unseen input vector $\mathbf{x}^* \in \mathcal{X}$, predict the corresponding true target label $y^* \in \mathcal{Y}$ with high accuracy and generalization.

Fundamental Task Types

- **Regression**: Continuous target space ($\mathcal{Y} \subseteq \mathbb{R}^k$).
- **Classification**: Discrete categorical space ($\mathcal{Y} \in \{1, \ldots, C\}$).

Mathematical Setup & Empirical Risk Minimization I

Supervised Learning Framework

Let $\mathbf{X} \in \mathbb{R}^{N \times d}$ be the feature design matrix and $\mathbf{y} \in \mathbb{R}^N$ be the vector of ground-truth target outputs.

- **Hypothesis Space** $\mathcal{H}$: The set of candidate functions $f_{\mathbf{w}}(\mathbf{x})$ parameterized by $\mathbf{w}$.
- **Loss Function** $\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}), y)$: Measures discrepancy between prediction and ground truth.

Empirical Risk Minimization (ERM)

Since the true underlying distribution $P(X, Y)$ is unknown, we minimize the empirical risk (training loss):

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} R_{\text{emp}}(\mathbf{w}) = \arg\min_{\mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(f_{\mathbf{w}}(\mathbf{x}^{(i)}), y^{(i)}\right)$$

Overview of Regression Problems I

Definition of Regression

Regression models model the relationship between a continuous outcome variable $y \in \mathbb{R}$ and one or more predictor variables $\mathbf{x} \in \mathbb{R}^d$.

Real-World Applications

- **Finance**: Stock price estimation, credit scoring.
- **Real Estate**: Property valuation based on square footage and location.
- **Healthcare**: Predicting patient response doses and survival duration.

Probabilistic View

Assuming target is generated by deterministic model plus additive Gaussian noise:

$$y = f(\mathbf{x}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Predicting $f(\mathbf{x})$ corresponds to modeling the conditional expectation $\mathbb{E}[Y \mid X = \mathbf{x}]$.

Simple Linear Regression & Ordinary Least Squares (OLS) I

Simple Linear Regression Model

With a single feature $x \in \mathbb{R}$, we assume a linear functional form:

$$\hat{y} = f(x) = w_1 x + w_0$$

where w_1 is the slope coefficient and w_0 is the bias (intercept).

Mean Squared Error (MSE) Objective

We find parameters (w_0, w_1) by minimizing the sum of squared residuals:

$$J(w_0, w_1) = \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - (w_1 x^{(i)} + w_0) \right)^2$$

Analytical Closed-Form Solution

Setting derivatives $\frac{\partial J}{\partial w_0} = 0$ and $\frac{\partial J}{\partial w_1} = 0$ yields:

$$w_1 = \frac{\sum_{i=1}^{N}(x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^{N}(x^{(i)} - \bar{x})^2} = \frac{\mathrm{Cov}(x, y)}{\mathrm{Var}(x)}, \qquad w_0 = \bar{y} - w_1 \bar{x}$$

Multiple Linear Regression & Normal Equation I

Matrix Formulation

For d input features, represent inputs as augmented vectors $\mathbf{x} = [1, x_1, x_2, \ldots, x_d]^T \in \mathbb{R}^{d+1}$ and weights $\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}, \quad \text{where } \mathbf{X} \in \mathbb{R}^{N \times (d+1)}$$

Matrix Loss Minimization

$$J(\mathbf{w}) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \frac{1}{2N} (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})$$

Gradient with respect to $\mathbf{w}$: $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{N} \mathbf{X}^T (\mathbf{X}\mathbf{w} - \mathbf{y})$.

The Normal Equation

Setting $\nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbf{0}$ yields the analytical solution:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Computational complexity is $O(d^3)$ due to matrix inversion.

Optimization via Gradient Descent I

Why Gradient Descent?

When feature count d is very large ($d > 10^4$), computing $(\mathbf{X}^T\mathbf{X})^{-1}$ becomes computationally intractable or non-invertible due to multicollinearity.

Iterative Parameter Update

Initialize $\mathbf{w}^{(0)}$ randomly or with zeros. Update iteratively using learning rate $\alpha > 0$:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(t)})$$

For MSE loss:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\alpha}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w}^{(t)} - \mathbf{y})$$

Gradient Descent Variants

- **Batch GD**: Uses all N samples per iteration (exact gradient).
- **Stochastic GD (SGD)**: Uses 1 random sample per step (fast, noisy).
- **Mini-batch GD**: Uses batch size $B \in [32, 256]$ (optimal balance).

Code: Linear Regression from Scratch & Scikit-Learn I

```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Generate synthetic dataset: y = 3*x1 + 2*x2 + 4 + noise
np.random.seed(42)
X = np.random.randn(100, 2)
y = 3 * X[:, 0] + 2 * X[:, 1] + 4 + np.random.randn(100) * 0.1

# Method 1: Closed-form via Normal Equation
X_b = np.c_[np.ones((100, 1)), X]  # Add bias term column
w_analytical = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y

# Method 2: Scikit-Learn Estimator
model = LinearRegression(fit_intercept=True)
model.fit(X, y)

print(f"Analytical Weights: {w_analytical}")
print(f"Sklearn Intercept: {model.intercept_:.4f}, Coeffs: {model.coef_}")
y_pred = model.predict(X)
print(f"Training MSE: {mean_squared_error(y, y_pred):.4f}")
```

Polynomial Regression & Non-linear Features I

Extending Linear Models to Non-linear Data

If relationship between x and y is non-linear, we can apply a non-linear feature transformation $\phi(x)$:

$$\phi(x) = [1, x, x^2, x^3, \ldots, x^p]^T$$

The model remains linear in its parameters $\mathbf{w}$:

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \cdots + w_p x^p = \mathbf{w}^T \phi(x)$$

Overfitting Risk with High-Degree Polynomials

- Low degree p (e.g., $p = 1$): High bias / underfitting (fails to capture trend).
- High degree p (e.g., $p = 15$): High variance / overfitting (fits noise).
- **Solution**: Cross-validation to select p, combined with feature scaling.

Code: Polynomial Regression Pipeline I

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

# 1. Create a scikit-learn Pipeline
poly_regression = Pipeline([
    ('poly_features', PolynomialFeatures(degree=3, include_bias=False)),
    ('scaler', StandardScaler()),
    ('linear_reg', LinearRegression())
])

# 2. Fit pipeline on training data
poly_regression.fit(X, y)

# 3. Predict and evaluate
y_poly_pred = poly_regression.predict(X)
mse = mean_squared_error(y, y_poly_pred)
print(f"3rd-Degree-Polynomial-MSE:-{mse:.4f}")
```

Evaluation Metrics for Regression Models I

Quantitative Metrics Overview

Let y_i be true targets, $\hat{y}_i$ be predictions, and $\bar{y}$ be target mean.

Absolute & Squared Errors

- **MAE** (Mean Absolute Error): $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$ *Robust to outliers.*
- **MSE** (Mean Squared Error): $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ *Penalizes large errors heavily.*
- **RMSE** (Root MSE): $\sqrt{\text{MSE}}$ *Interpretable in target units.*

R-Squared (R^2) Score

$$R^2 = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

- $R^2 = 1.0$: Perfect predictions.
- $R^2 = 0.0$: Performance equals baseline mean predictor $\bar{y}$.
- $R^2 < 0.0$: Predictor performs worse than mean predictor.

Summary: Supervised Learning & Regression Fundamentals I

Key Takeaways

1. **Supervised Paradigm**: Mapping feature inputs to targets via labeled pairs $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}$.
2. **Linear Regression**: Fundamental baseline predicting continuous outcomes by minimizing squared errors.
3. **Optimization**: Closed-form Normal Equation $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ vs. iterative Gradient Descent.
4. **Non-linearity**: Polynomial feature maps $\phi(x)$ extend linear capacity without changing linear parameterization.
5. **Evaluation**: Utilize RMSE, MAE, and R^2 to assess generalization quality.

Brief Overview of Supervised Machine Learning I

What is Supervised Machine Learning?

Supervised learning is an algorithm design paradigm where a model learns a mapping function $f : \mathcal{X} \to \mathcal{Y}$ from a dataset of labeled training examples.

- **Dataset Structure**: $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$, where:
  - $x^{(i)} \in \mathbb{R}^d$ represents the d-dimensional input feature vector.
  - $y^{(i)}$ represents the ground-truth target label.
- **Primary Objective**: Generalize to accurately predict target outputs $\hat{y}$ for novel, unseen input samples x_{new}.
- **Core Paradigms**:
  - **Classification**: Target space $\mathcal{Y}$ is discrete (e.g., categorical labels $\{0, 1\}$).
  - **Regression**: Target space $\mathcal{Y}$ is continuous (e.g., real numbers $\mathbb{R}$).

Core Components of the Supervised Learning Pipeline I

1. Data Representation

- **Feature Matrix**: $X \in \mathbb{R}^{N \times d}$ containing N samples and d features.
- **Target Vector**: $Y \in \mathbb{R}^{N}$ containing ground-truth values.

2. Model Hypothesis Class

Select a parameterized function family $h_\theta(x)$ (e.g., linear, polynomial, neural net).

3. Empirical Risk Minimization

Define a loss metric $\mathcal{L}(y, h_\theta(x))$ measuring prediction discrepancy.

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(y^{(i)}, h_\theta(x^{(i)})\right)$$

4. Evaluation & Generalization

Assess model performance on a separate validation/test set to ensure no overfitting.

Defining Regression Analysis I

Definition of Regression Analysis

Regression analysis is a fundamental statistical and machine learning method used to model the relationship between a continuous dependent target variable $y \in \mathbb{R}$ and one or more independent feature variables $x \in \mathbb{R}^d$.

- **Generative Assumption**:

$$y = f(x) + \epsilon$$

  where $f(x) = \mathbb{E}[Y \mid X = x]$ is the true systematic function, and ϵ represents random irreducible noise with $\mathbb{E}[\epsilon] = 0$ and $\text{Var}(\epsilon) = \sigma^2$.

- **Goal of Regression**: Construct an estimator $\hat{f}(x)$ such that the expected risk or deviation between y and $\hat{y} = \hat{f}(x)$ is minimized.

Taxonomy & Real-World Applications of Regression I

Types of Regression Models

- **Simple Linear Regression**: Single predictor $x \in \mathbb{R}$, linear mapping $y = \beta_0 + \beta_1 x$.
- **Multiple Linear Regression**: Multiple features $x \in \mathbb{R}^d$, $y = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$.
- **Non-linear / Polynomial Regression**: Models non-linear interactions via basis function expansions (e.g., x^2, $\sin(x)$).

Practical Applications

- **Real Estate Valuation**: Estimating house prices based on size, location, and age.
- **Finance & Economics**: Forecasting stock returns, inflation rates, or credit risk metrics.
- **Healthcare**: Predicting patient blood pressure or drug response levels based on clinical features.

Measuring Prediction Error in Regression I

- **Residual**: The difference between observed outcome $y^{(i)}$ and predicted value $\hat{y}^{(i)}$:

$$e^{(i)} = y^{(i)} - \hat{y}^{(i)}$$

Common Regression Loss Functions

- **Mean Squared Error (MSE)**:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

  Penalizes larger errors disproportionately due to squaring.
- **Root Mean Squared Error (RMSE)**: $\text{RMSE} = \sqrt{\text{MSE}}$ (interpretable in original target units).
- **Mean Absolute Error (MAE)**:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} \left| y^{(i)} - \hat{y}^{(i)} \right|$$

  More robust to extreme outliers compared to MSE.

Python Code Example: Supervised Regression I

```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# 1. Generate synthetic dataset: y = 2.5 * X + 4.0 + noise
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 2.5 * X + 4.0 + np.random.randn(100, 1) * 0.5

# 2. Split dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Initialize and fit Supervised Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# 4. Predict on test set and evaluate performance
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)

print(f"Slope (beta_1): {model.coef_[0][0]:.4f}")
print(f"Intercept (beta_0): {model.intercept_[0]:.4f}")
print(f"Test Set MSE: {mse:.4f}")
```

Lecture Summary & Next Steps I

Key Takeaways

1. **Supervised Learning** uses paired feature-target observations (x, y) to construct predictive mapping functions.
2. **Regression Analysis** addresses continuous outcome targets by modeling conditional expected values $\mathbb{E}[Y \mid X = x]$.
3. Optimization algorithms solve regression problems by minimizing loss metrics like **MSE** or **MAE**.

Looking Ahead

In subsequent lectures, we will explore:

- Closed-form solutions via Ordinary Least Squares (OLS) Normal Equations.
- Iterative optimization via Gradient Descent.
- Regularization techniques (Ridge, Lasso) to combat overfitting.

Lecture 4.3: Overview & Learning Objectives I

Unit 4: Supervised Machine Learning

Supervised Learning learns a mapping function $f : \mathcal{X} \to \mathcal{Y}$ from labeled training pairs $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i \in \mathcal{X}$ are input features and $y_i \in \mathcal{Y}$ are target outputs.

4.1.2 Learning Steps

- Problem Formulation & Target Definition
- Data Acquisition & Preprocessing
- Model Selection & Hypothesis Space
- Empirical Risk Minimization
- Validation, Testing & Deployment

4.3.2 Types of Regression

- Simple & Multiple Linear Regression
- Polynomial Regression
- Regularized Regression (L_1, L_2, ElasticNet)
- Tree-based & Non-Parametric Regression
- Support Vector Regression (SVR)

Learning Steps in Supervised Machine Learning (Part 1) I

Step-by-Step Execution Pipeline

1. **Problem Formulation & Target Definition**:
   - Determine the target output domain: Continuous ($y \in \mathbb{R}$ for Regression) or Categorical ($y \in \{C_1, \ldots, C_k\}$ for Classification).
   - Establish task metrics (e.g., MSE, R^2, Accuracy, F1-score) and business constraints.
2. **Data Collection & Preprocessing**:
   - Gather historical data samples $\{(x_i, y_i)\}$. Clean noise, deal with missing values and outliers.
   - Perform feature engineering: scaling (StandardScaler, MinMaxScaler), encoding, and dimensionality selection.
3. **Dataset Partitioning**:
   - Split data into **Train** ($60 - 80\%$), **Validation** ($10 - 20\%$), and **Test** ($10 - 20\%$) sets to evaluate out-of-sample generalization.

Learning Steps in Supervised Machine Learning (Part 2) I

Model Selection, Optimization, and Evaluation

1. **Hypothesis Space & Model Selection**:
   - Choose candidate functional family $f_\theta(x) \in \mathcal{H}$ parameterized by weights θ (e.g., linear combination, decision trees, neural network).
2. **Model Training & Optimization**:
   - Fit parameter set θ via optimization algorithm (e.g., OLS closed-form, Gradient Descent) by minimizing training loss:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) + \lambda \Omega(\theta)$$

3. **Validation, Hyperparameter Tuning & Testing**:
   - Tune structural parameters using Cross-Validation on the validation set.
   - Evaluate final model generalization capability on the isolated test dataset.

Mathematical Framework: Empirical Risk Minimization I

Formal Setup of Supervised Learning

Given training dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ sampled i.i.d. from joint distribution $P(X, Y)$:

Loss Function $\mathcal{L}(y, \hat{y})$

Measures prediction discrepancy:

- **Squared Loss (Regression):**

$$\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$$

- **Absolute Loss (Regression):**

$$\mathcal{L}(y, \hat{y}) = |y - \hat{y}|$$

Empirical Risk Minimization (ERM)

Minimizes average training error:

$$\mathcal{R}_{\text{emp}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, f_\theta(x_i))$$

Goal: Minimize true generalization risk $\mathcal{R}(\theta) = \mathbb{E}_{(x,y)\sim P}[\mathcal{L}(y, f_\theta(x))]$.

Introduction to Regression Analysis I

What is Regression?

Regression is a primary subfield of supervised learning where the objective is to predict a **continuous numerical target** $y \in \mathbb{R}$ given input vector $x \in \mathbb{R}^p$.

Core Objectives

- **Prediction**: Estimate unknown continuous values for unseen inputs x^*.
- **Inference**: Analyze direct relationships, coefficient magnitude, and feature impact on y.

Typical Applications

- Housing valuation based on physical features.
- Temperature & climate forecasting.
- Customer Lifetime Value (CLV) estimation.

Taxonomy & Types of Regression Models I

Regression Family	Specific Model Type	Key Operational Trait
Linear Models	Simple / Multiple Linear	Linear mapping, OLS analytical solution
Polynomial Basis	Polynomial Regression	Non-linear curve fitting via power features
Regularized Linear	Ridge (L_2 Penalty)	Weight shrinkage, handles multicollinearity
Regularized Linear	Lasso (L_1 Penalty)	Sparse coefficients, feature selection
Regularized Linear	ElasticNet ($L_1 + L_2$)	Balanced sparsity & feature grouping
Non-Parametric	Decision Tree Regressor	Piecewise step-constant spatial partitions
Ensemble Trees	Random Forest Regressor	Aggregates trees to reduce variance
Kernel Methods	Support Vector Regression	Insensitive margin tube (ϵ), kernel trick

Model Selection Trade-off

High interpretability (Linear/Ridge) vs. high flexibility & expressive power (Trees/SVR).

Linear and Polynomial Regression I

1. Simple & Multiple Linear Regression

Assumes linear relationship between predictor variables X and continuous outcome y:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon = X\beta + \epsilon$$

- **Ordinary Least Squares (OLS)** estimates: $\hat{\beta} = (X^T X)^{-1} X^T y$.
- Highly interpretable; vulnerable to outliers and strong feature correlation (multicollinearity).

2. Polynomial Regression

Models non-linear relationships by creating degree-d polynomial combinations:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d + \epsilon$$

- Remains linear with respect to parameters β, allowing standard OLS fitting.
- High degree polynomials ($d \gg 3$) risk severe overfitting and boundary instability.

Regularized Regression: Ridge, Lasso, and ElasticNet I

Regularization penalizes large weight coefficients to prevent overfitting and control model complexity:

$$\mathcal{L}_{\text{reg}}(\beta) = \frac{1}{2N}\|y - X\beta\|_2^2 + \Omega(\beta)$$

Ridge (L_2 Penalty)

$$\Omega(\beta) = \lambda\|\beta\|_2^2 = \lambda\sum_{j=1}^{p}\beta_j^2$$

- Shrinks weights smoothly toward zero.
- Stabilizes estimates under multicollinearity.

Lasso (L_1 Penalty)

$$\Omega(\beta) = \lambda\|\beta\|_1 = \lambda\sum_{j=1}^{p}|\beta_j|$$

- Drives irrelevant weights exactly to zero.
- Performs automatic feature selection.

ElasticNet

$$\Omega(\beta) = \lambda_1\|\beta\|_1 + \lambda_2\|\beta\|_2^2$$

- Convex combination of L_1 and L_2.
- Retains correlated feature groups while sparse.

Non-Parametric & Advanced Regression Methods I

Decision Tree & Ensemble Regressors

- **Decision Tree Regressor**: Divides feature space into orthogonal regions $\mathcal{R}_m$ and predicts mean regional response $\bar{y}_m$.
- **Random Forest Regressor**: Averages predictions across an ensemble of decorrelated trees to reduce variance.
- Captures complex non-linear feature interactions naturally.

Support Vector Regression (SVR)

- Uses an ϵ-insensitive loss function: ignores errors smaller than threshold ϵ.
- Formulates objective:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$$

- Applies kernel trick (RBF, Polynomial) for non-linear mappings into higher-dimensional feature space.

Python Implementation: Comparing Regression Models I

```
import numpy as np
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error, r2_score

# 1. Generate Synthetic Data
np.random.seed(42)
X = np.linspace(-3, 3, 100).reshape(-1, 1)
y = 0.5 * X.squeeze()**2 + X.squeeze() + 2 + np.random.normal(0, 0.5, 100)

# 2. Fit Simple Linear Regression
lin_reg = LinearRegression().fit(X, y)

# 3. Fit Polynomial Regression (Degree 2)
poly_reg = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
poly_reg.fit(X, y)

# 4. Fit Regularized Ridge & Lasso Regression
ridge_reg = Ridge(alpha=1.0).fit(X, y)
lasso_reg = Lasso(alpha=0.1).fit(X, y)

# 5. Evaluate Performance
y_pred_poly = poly_reg.predict(X)
print(f"Poly-MSE: {mean_squared_error(y, y_pred_poly):.4f}")
print(f"Poly-R2 Score: {r2_score(y, y_pred_poly):.4f}")
```

Summary: Model Selection Guide for Regression I

Practical Model Selection Guidelines

1. **Linear baseline & high interpretability needed**: → *Simple / Multiple Linear Regression* (OLS).
2. **Multicollinearity present or high-dimensional features**: → *Ridge Regression* (L_2 regularization).
3. **High-dimensional space requiring sparse feature selection**: → *Lasso Regression* (L_1) or *ElasticNet*.
4. **Non-linear continuous curve with single/few features**: → *Polynomial Regression*.
5. **Complex tabular dataset with strong non-linear interactions**: → *Random Forest / Gradient Boosted Trees*.
6. **Complex non-linear boundary with low sample density**: → *Support Vector Regression* (SVR with RBF kernel).

Best Practice

Always evaluate regression models on out-of-sample test datasets using metrics like MSE, RMSE, MAE, and R^2, and perform residual error analysis.

Lecture 4.4: Overview & Learning Objectives I

Unit 4: Supervised Machine Learning

Supervised Machine Learning algorithms map input features $x \in \mathcal{X}$ to labeled target outputs $y \in \mathcal{Y}$ using training pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.

4.3.2 Real World Applications

- Healthcare: Diagnosis & Risk Prediction
- Finance: Credit Scoring & Fraud Detection
- Industry: House Pricing & Demand Forecast
- Tech: Email Spam Filtering & NLP

4.3.3 Linear Regression

- Classification of Linear Models
- Scalar, Vector, Matrix Formulation
- Geometry of Positive & Negative Slopes
- Parameter Estimation via OLS & Code

4.1.3 Real-World Applications: Healthcare & Finance I

Healthcare & Medical Diagnostics

- **Disease Detection**: Classification algorithms analyze imaging (MRI, X-rays) to detect malignant tumors or diabetic retinopathy.
- **Patient Readmission**: Logistic regression predicts 30-day readmission likelihood based on patient EHR metrics.
- **Genomic Medicine**: Regression models predict therapeutic drug efficacy from patient genetic markers.

Finance & Banking Systems

- **Credit Risk Scoring**: Estimate loan default probability ($y \in \{0, 1\}$) using income, debt ratio, and credit history.
- **Fraud Detection**: Supervised classifiers detect unauthorized credit card transactions in real-time.
- **Algorithmic Trading**: Predict continuous asset prices ($\hat{y} \in \mathbb{R}$) using economic indicator vectors.

4.1.3 Real-World Applications: Industry & E-Commerce I

Real Estate & Supply Chain

- **Automated Valuation Models (AVM)**: Multiple linear regression estimates house market values based on area, rooms, and location.
- **Demand Forecasting**: Regression models forecast product demand for optimal warehouse inventory allocation.

Cybersecurity

- **Spam Filtering**: Classifies emails as `Spam` or `Ham` using text feature vectors.

E-Commerce & Marketing

- **Customer Lifetime Value (CLV)**: Models future revenue generated per user over a time horizon.
- **Churn Prediction**: Binary classification identifies customers at high risk of subscription cancellation.
- **Dynamic Pricing**: Predicts optimal pricing strategies dynamically based on competitor data and demand.

4.3.3 Linear Regression: Fundamentals & Concept I

Definition

Linear Regression is a parametric supervised learning model designed to establish a linear relationship between input features $\mathbf{x} \in \mathbb{R}^d$ and a continuous target variable $y \in \mathbb{R}$.

Core Characteristics

- **Target Domain**: Continuous real values ($y \in \mathbb{R}$).
- **Functional Form**: Linear combination of input parameters.
- **High Interpretability**: Weights quantify feature influence directly.
- **Efficiency**: Fast to compute analytical or numerical solutions.

Learning Goal

- Fit a line (2D) or hyperplane (d-D) that minimizes prediction residuals:

$$e_i = y_i - \hat{y}_i$$

- Generalize accurately to unseen test instances.

4.3.3 Types of Linear Regression I

Taxonomy of Linear Regression Models

Linear regression algorithms are categorized based on feature dimensionality, basis function transformations, and regularization constraints.

Dimensionality & Basis Functions

- **Simple Linear Regression**: One predictor variable ($x \in \mathbb{R}^1$). Fits a 2D line.
- **Multiple Linear Regression**: Multiple predictor variables ($\mathbf{x} \in \mathbb{R}^d$). Fits a hyperplane.
- **Polynomial Regression**: Incorporates non-linear terms (x^2, x^3). Linear in weights $\mathbf{w}$.

Regularization Variants

- **Ordinary Least Squares (OLS)**: Unconstrained squared error minimization.
- **Ridge Regression (L_2)**: Adds penalty $\lambda\|\mathbf{w}\|_2^2$ to address multicollinearity.
- **Lasso Regression (L_1)**: Adds penalty $\lambda\|\mathbf{w}\|_1$; drives weights to 0 for feature selection.
- **ElasticNet**: Combines L_1 and L_2 penalties.

4.3.3 Mathematical Formulation of Linear Regression I

1. Simple Linear Regression (Scalar Form)

For a single input x_i, the response variable is modeled as:

$$y_i = w_0 + w_1 x_i + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

where w_0 represents the **y-intercept**, w_1 is the **slope**, and ϵ_i is the random error term.

2. Multiple Linear Regression (Vector Form)

For d input features $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{id}]^T$:

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id} = w_0 + \sum_{j=1}^{d} w_j x_{ij}$$

4.3.3 Mathematical Formulation of Linear Regression II

3. Matrix Notation (Compact System)

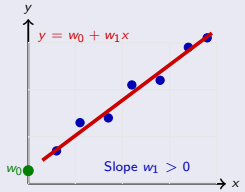
Augmenting feature vector with $x_{i0} = 1$, we write for all N data instances:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}, \quad \text{where } \mathbf{X} \in \mathbb{R}^{N \times (d+1)},\ \mathbf{w} \in \mathbb{R}^{d+1},\ \mathbf{y} \in \mathbb{R}^{N}$$

4.3.3 Geometric Interpretation: Positive vs. Negative Slope I

Positive Slope ($w_1 > 0$)

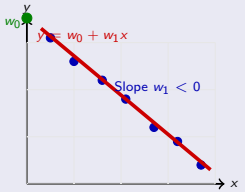
Direct linear relationship: y increases as x increases.



Example: House Price vs. Living Area.

Negative Slope ($w_1 < 0$)

Inverse linear relationship: y decreases as x increases.



Example: Used Car Price vs. Mileage.

4.3.3 Parameter Estimation: Loss Function & Optimization I

Mean Squared Error (MSE) Cost Function

Parameters $\mathbf{w}$ are estimated by minimizing the average empirical loss $J(\mathbf{w})$:

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 = \frac{1}{2N} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$$

Closed-Form Solution: Normal Equation

Solving $\nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbf{0}$ yields:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

- Exact global minimum.
- Computationally expensive for $d > 10^4$ due to $(\mathbf{X}^T \mathbf{X})^{-1}$.

Iterative Solution: Gradient Descent

Iterative parameter update rule:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(t)})$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \frac{\alpha}{N} \mathbf{X}^T (\mathbf{y} - \mathbf{X}\mathbf{w}^{(t)})$$

- Efficiently scales to large-scale data ($N, d \gg 10^5$).

Python Implementation: Linear Regression with Scikit-Learn I

Python Implementation: Linear Regression with Scikit-Learn II

Hands-on Python Example

```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# 1. Generate synthetic dataset (x: feature, y: target)
np.random.seed(42)
X = 2 * np.random.rand(100, 1)  # 100 samples, 1 feature
y = 4 + 3 * X + np.random.randn(100, 1)  # y = 4 + 3x + noise

# 2. Instantiate and train Linear Regression model
model = LinearRegression()
model.fit(X, y)

# 3. Extract fitted parameters (slope & intercept)
w0 = model.intercept_[0]  # Intercept (w_0)
w1 = model.coef_[0][0]    # Slope (w_1)
print(f"Fitted Line: y = {w0:.2f} + {w1:.2f}*x")

# 4. Predict and evaluate performance
y_pred = model.predict(X)
mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)
print(f"MSE: {mse:.4f} | R^2 Score: {r2:.4f}")
```

Lecture 4.4: Summary & Key Takeaways I

Core Concepts Summary

1. **Real-World Applications**: Supervised ML powers critical applications in Healthcare, Finance, Cybersecurity, Real Estate, and E-Commerce.
2. **Linear Regression Paradigm**: Parametric model fitting a line/hyperplane $y = \mathbf{w}^T\mathbf{x} + \epsilon$ to continuous targets.
3. **Model Taxonomy**: Includes Simple, Multiple, Polynomial, and Regularized variants (Ridge L_2, Lasso L_1, ElasticNet).
4. **Slope Geometry**: Positive slope ($w_1 > 0$) indicates direct relationship; negative slope ($w_1 < 0$) indicates inverse relationship.
5. **Parameter Fitting**: Solved via OLS Normal Equations $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ or iteratively via Gradient Descent.

Supervised Machine Learning: Core Paradigm I

Definition & Goal

Supervised Learning algorithms learn a mapping function $f : \mathcal{X} \to \mathcal{Y}$ from labeled training pairs $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, where $x_i \in \mathbb{R}^d$ are features and y_i are ground-truth targets.

Regression Tasks

- **Target**: Continuous variable ($y_i \in \mathbb{R}$).
- **Examples**: House price prediction, temperature forecasting, stock index estimation.

Classification Tasks

- **Target**: Discrete class label ($y_i \in \{1, \ldots, C\}$).
- **Examples**: Spam detection, medical diagnosis, image recognition.

Advantages of Supervised Machine Learning I

Key Strengths

High Predictive Accuracy Explicit ground-truth guidance enables optimization of precise loss functions tailored to target tasks.

Clear Performance Metrics Direct evaluation against ground truth using standard quantitative metrics (Accuracy, F1-Score, MSE, R^2).

Interpretability & Control Parametric models (e.g., Linear Regression, Decision Trees) allow direct analysis of feature importance and decision boundaries.

Task-Specific Optimization Models optimize directly for specific business objectives or clinical/engineering outcomes.

Disadvantages & Limitations of Supervised Machine Learning I

Key Challenges

High Labeling Cost	Requires large volumes of accurately annotated data, often demanding expensive domain expertise or human annotation.
Risk of Overfitting	Complex models may memorize training noise rather than generalizable patterns (Bias-Variance Tradeoff).
Closed-World Assumption	Models struggle with Out-Of-Distribution (OOD) data and unseen target categories at test time.
Data Bias Propagation	Historical biases present in training labels are directly encoded and amplified by the model.

Operational Bottleneck

Data annotation is often the single biggest operational bottleneck in real-world ML deployment pipelines.

Trade-off Matrix: Supervised Machine Learning I

Methodological Comparison

Dimension	Supervised ML	Unsupervised ML
Data Req.	Labeled pairs (X, y)	Unlabeled features X
Primary Goal	Map inputs to targets	Discover hidden structures
Evaluation	Objective ground-truth metrics	Heuristic / internal metrics
Complexity	High data prep, low ambiguity	Low data prep, high evaluation ambiguity

Introduction to Simple Linear Regression I

What is Simple Linear Regression?

Simple Linear Regression (SLR) models the relationship between a single independent continuous predictor variable X and a dependent target variable Y using a linear function.

Mathematical Formulation

For a dataset with N observations $\{(x_1, y_1), \ldots, (x_N, y_N)\}$:

$$y_i = w_0 + w_1 x_i + \epsilon_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

- w_0 (β_0): **y-intercept** — expected value of Y when $X = 0$.
- w_1 (β_1): **Slope coefficient** — expected change in Y per unit increase in X.
- ϵ_i: **Unobserved random error term** ($\epsilon_i \sim \mathcal{N}(0, \sigma^2)$).

Simple Linear Regression: Objective Function I

Predictive Model & Residuals

Given estimates $(\hat{w}_0, \hat{w}_1)$, the predicted target is $\hat{y}_i = \hat{w}_0 + \hat{w}_1 x_i$. The residual (error) for sample i is:

$$e_i = y_i - \hat{y}_i = y_i - (\hat{w}_0 + \hat{w}_1 x_i)$$

Ordinary Least Squares (OLS) Objective

Find parameters (w_0, w_1) that minimize the **Sum of Squared Errors (SSE)**:

$$J(w_0, w_1) = \text{SSE}(w_0, w_1) = \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} \left(y_i - (w_0 + w_1 x_i)\right)^2$$

Why Square the Errors?

1. Penalizes larger errors more heavily than small errors.
2. Prevents positive and negative residuals from canceling out.
3. Yields a convex, smooth, and twice-differentiable objective function.

Deriving the OLS Analytical Solution I

First-Order Conditions (Normal Equations)

Set partial derivatives of $J(w_0, w_1)$ with respect to w_0 and w_1 to zero:

$$\frac{\partial J}{\partial w_0} = -2\sum_{i=1}^{N}(y_i - w_0 - w_1 x_i) = 0 \implies \sum_{i=1}^{N}(y_i - w_0 - w_1 x_i) = 0$$

$$\frac{\partial J}{\partial w_1} = -2\sum_{i=1}^{N} x_i(y_i - w_0 - w_1 x_i) = 0 \implies \sum_{i=1}^{N} x_i(y_i - w_0 - w_1 x_i) = 0$$

Closed-Form Expressions

Solving the system of equations yields:

$$\hat{w}_1 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \bar{x})^2} = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(X)}, \qquad \hat{w}_0 = \bar{y} - \hat{w}_1 \bar{x}$$

where $\bar{x} = \frac{1}{N}\sum x_i$ and $\bar{y} = \frac{1}{N}\sum y_i$ are the sample means.

Code: Implementation of OLS from Scratch & Scikit-Learn I

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Synthetic dataset: y = 3 + 2.5 * x + noise
np.random.seed(42)
X = np.array([1, 2, 3, 4, 5], dtype=np.float64)
y = np.array([5.1, 8.2, 10.3, 13.1, 15.7], dtype=np.float64)

# 1. Closed-form OLS calculation from scratch
x_bar, y_bar = np.mean(X), np.mean(y)
w1_scratch = np.sum((X - x_bar) * (y - y_bar)) / np.sum((X - x_bar)**2)
w0_scratch = y_bar - w1_scratch * x_bar

print(f"Scratch -> Intercept (w0): {w0_scratch:.4f}, Slope (w1): {w1_scratch:.4f}")

# 2. Equivalent Scikit-Learn Implementation
reg = LinearRegression().fit(X.reshape(-1, 1), y)
print(f"Sklearn -> Intercept: {reg.intercept_:.4f}, Slope: {reg.coef_[0]:.4f}")
```

Assumptions of Simple Linear Regression I

Classical Gauss-Markov Assumptions

To guarantee that OLS estimators are **BLUE** (Best Linear Unbiased Estimators):

1. **Linearity**: The relationship between X and Y is linear in parameters.
2. **Independence**: Residuals ϵ_i are mutually independent ($\text{Cov}(\epsilon_i, \epsilon_j) = 0$).
3. **Homoscedasticity**: Constant error variance ($\text{Var}(\epsilon_i) = \sigma^2$ for all i).
4. **Normality of Errors**: $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ (required for hypothesis testing & confidence intervals).

Diagnostics

Always perform residual diagnostics (e.g., Residual vs. Fitted plot, Q-Q plot) to verify these assumptions before deploying predictions!

Evaluating Simple Linear Regression I

Goodness-of-Fit Metric: Coefficient of Determination (R^2)

R^2 measures the proportion of total variance in Y explained by feature X:

$$R^2 = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

Interpretation

- $R^2 = 1$: Perfect model fit.
- $R^2 = 0$: Model performs no better than baseline mean $\bar{y}$.
- $R^2 < 0$: Model performs worse than the sample mean baseline.

Additional Error Metrics

- **MAE**: $\frac{1}{N}\sum |y_i - \hat{y}_i|$
- **MSE**: $\frac{1}{N}\sum (y_i - \hat{y}_i)^2$
- **RMSE**: $\sqrt{\text{MSE}}$

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 4**: Supervised Machine Learning
- **Topics**:
  - 4.2 Classification Fundamentals & Taxonomy
  - 4.3.5 Real-World Applications of Regression Analysis

Learning Objectives:

1. Formalize the **Classification Paradigm** and distinguish binary, multiclass, and multi-label settings.
2. Differentiate between **Discriminative** and **Generative** classification approaches.
3. Master core classification algorithms and decision boundary mechanics.
4. Analyze real-world regression applications across finance, healthcare, and engineering.
5. Implement classification and regression pipelines using Python and `scikit-learn`.

Introduction to Classification I

What is Classification?

Classification is a supervised learning paradigm where the algorithm learns a mapping function $f : \mathcal{X} \to \mathcal{Y}$ to assign an input feature vector $\mathbf{x} \in \mathbb{R}^d$ to a discrete categorical label $y \in \mathcal{Y}$.

- **Target Space** $\mathcal{Y}$: A finite, discrete set of classes $\{c_1, c_2, \ldots, c_K\}$.
- **Decision Surface / Boundary**: The geometric partition in feature space $\mathbb{R}^d$ separating regions assigned to different classes:

$$\{\mathbf{x} \in \mathbb{R}^d \mid P(Y = c_i \mid \mathbf{x}) = P(Y = c_j \mid \mathbf{x})\}$$

- **Operational Goal**: Minimize expected misclassification loss or maximize classification accuracy on unseen test data.

Classification Problem Taxonomy I

1. Binary Classification

- Target set: $\mathcal{Y} = \{0, 1\}$ or $\{-1, +1\}$.
- Output: Single probability score $p = P(Y = 1 \mid \mathbf{x})$.
- **Real-World Examples**:
  - Email Spam Filtering (Spam vs. Legitimate).
  - Credit Card Fraud Detection.
  - Medical Diagnosis (Disease Positive/Negative).

2. Multiclass Classification

- Target set: $\mathcal{Y} = \{1, 2, \ldots, K\}$ $(K > 2)$.
- Output: Categorical distribution $\mathbf{p} \in [0, 1]^K$ with $\sum_{k=1}^{K} p_k = 1$.
- **Real-World Examples**:
  - Optical Character Recognition (MNIST digits 0–9).
  - Satellite Image Land Cover Classification.
  - Document Topic Categorization.

Multi-Label Classification

In **Multi-Label Classification**, an instance can simultaneously belong to multiple classes (e.g., an article tagged with both *Machine Learning* and *Healthcare*), where $\mathbf{y} \in \{0, 1\}^K$.

Discriminative vs. Generative Classification I

1. Discriminative Models

Directly model the conditional distribution $P(Y \mid \mathbf{X})$ or learn decision boundaries separating classes.

- **Focus**: Maximizing separation efficiency between classes.
- **Examples**: Logistic Regression, Support Vector Machines (SVM), Decision Trees, Neural Networks.
- **Advantage**: Often achieves higher accuracy when large labeled datasets are available.

2. Generative Models

Model the joint distribution $P(\mathbf{X}, Y) = P(\mathbf{X} \mid Y)P(Y)$ by estimating class-conditional likelihoods $P(\mathbf{X} \mid Y)$ and class priors $P(Y)$.

- **Inference**: Applies Bayes' Rule to compute posterior class probabilities:

$$P(Y = c_k \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid Y = c_k)P(Y = c_k)}{\sum_{j=1}^{K} P(\mathbf{x} \mid Y = c_j)P(Y = c_j)}$$

- **Examples**: Naive Bayes, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA).

Overview of Key Classification Algorithms I

- **Logistic Regression**: Uses the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ to model posterior probability:

$$P(Y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

  Trained by minimizing Binary Cross-Entropy loss.

- **Support Vector Machines (SVM)**: Finds the maximum-margin hyperplane separating classes. Utilizes kernel functions $K(\mathbf{x}, \mathbf{x}')$ to project data into higher-dimensional spaces for non-linear separation.
- **Decision Trees & Random Forests**: Partition feature space hierarchically based on impurity reduction (Gini Impurity, Information Gain). Ensembles aggregate multiple trees to reduce variance.
- **k-Nearest Neighbors (k-NN)**: Instance-based non-parametric classifier that assigns labels based on majority vote among k nearest neighbors under a distance metric (e.g., Euclidean distance).

Python Implementation: Classification Pipeline I

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# 1. Synthesize binary classification dataset
X, y = make_classification(n_samples=1000, n_features=10,
                           n_informative=5, random_state=42)

# 2. Split dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42
)

# 3. Initialize and train Logistic Regression model
clf = LogisticRegression(solver='lbfgs', C=1.0)
clf.fit(X_train, y_train)

# 4. Predict and evaluate performance
y_pred = clf.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print(f"Test Set Accuracy: {acc * 100:.2f}%")
print(classification_report(y_test, y_pred))
```

Real-World Regression: Finance & Economics I

1. Asset Pricing & Market Volatility Forecasting

- **Financial Volatility Modeling**: Regression models predict continuous asset volatility metrics based on macroeconomic factors, interest rates, and historical trading volumes.
- **Capital Asset Pricing Model (CAPM)**:

$$R_i - R_f = \alpha_i + \beta_i(R_m - R_f) + \epsilon_i$$

Quantifies market risk sensitivity (β_i) of security i.

2. Real Estate Valuation & Credit Risk

- **Hedonic Housing Models**: Estimates continuous home prices ($y \in \mathbb{R}^+$) using structural features (square footage, bedrooms), location indices, and local market trends.
- **Credit Limit Determination**: Regression algorithms forecast optimal credit line amounts for bank customers as a function of annual income, debt ratio, and credit history score.

Real-World Regression: Healthcare & Medicine I

1. Physiological Monitoring & Diagnostics

- **Continuous Blood Pressure Estimation**: Predicts continuous systolic and diastolic pressure (mmHg) using features derived from photoplethysmogram (PPG) and electrocardiogram (ECG) waveforms.
- **Pharmacokinetic Dosing Models**: Predicts drug plasma concentration over time to establish customized, optimal dosage amounts for patients.

2. Healthcare Management & Disease Trajectory

- **Hospital Length-of-Stay (LOS) Prediction**: Estimates total days a hospitalized patient will occupy a bed based on clinical vitals, age, and laboratory markers.
- **Cognitive Decline Scoring**: Fits continuous progression indices (e.g., MMSE scores in Alzheimer's patients) against genetic indicators and age.

Real-World Regression: Engineering & Smart Grids I

1. Predictive Maintenance & Industrial IoT

- **Remaining Useful Life (RUL) Estimation**: Models remaining operational cycles of industrial machinery and turbofan engines before maintenance is required:

$$\text{RUL} = f(\text{vibration}, \text{temperature}, \text{pressure}, \text{operating\_hours})$$

- **Manufacturing Yield Optimization**: Predicts chemical product output yield percentage based on reactor temperature, pressure, and catalyst concentration.

2. Smart Grid & Energy Demand Forecasting

- **Electrical Load Forecasting**: Predicts hourly power demand (MW) on regional power grids based on weather forecasts, time of day, and industrial activity.
- **Renewable Generation**: Forecasts solar photovoltaic power output based on solar irradiance, ambient temperature, and humidity.

Comparative Analysis: Classification vs. Regression I

Domain	Regression Task ($y \in \mathbb{R}$)	Classification Task ($y \in \mathcal{Y}$)
Real Estate	Predict home price in USD ($450,000).	Predict status: Sold vs. Unsold.
Healthcare	Predict blood glucose level (mg/dL).	Classify patient: Diabetic vs. Non-Diabetic.
Finance	Forecast exact stock return percentage (+3.4%).	Predict market direction: Up vs. Down.
Automotive	Estimate distance to obstacle (12.5 m).	Identify object type: Pedestrian, Car, Sign.
Retail	Predict customer lifetime spend ($1, 200).	Predict customer churn risk: High, Med, Low.

Design Principle

Formulate your ML problem as **regression** when output precision on a continuous scale is needed, and as **classification** when categorical decision boundaries are required.

Python Implementation: Real-World Regression Pipeline I

```python
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# 1. Load real-world continuous target dataset
housing = fetch_california_housing()
X, y = housing.data, housing.target  # Target: Median house value ($100k)

# 2. Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42
)

# 3. Instantiate and fit Random Forest Regressor
reg = RandomForestRegressor(n_estimators=100, random_state=42)
reg.fit(X_train, y_train)

# 4. Predict and evaluate continuous target performance
y_pred = reg.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Root Mean Squared Error (RMSE): ${rmse * 100000:.2f}")
print(f"Mean Absolute Error (MAE): ${mae * 100000:.2f}")
print(f"R-squared Score (R2): {r2:.4f}")
```

Python Implementation: Real-World Regression Pipeline II

Summary & Key Takeaways I

Core Concepts Covered

- **Classification Paradigm**: Supervised learning task for mapping continuous inputs $\mathbf{x} \in \mathbb{R}^d$ to discrete categorical targets $y \in \{c_1, \ldots, c_K\}$.
- **Model Taxonomy**:
  - **Discriminative** ($P(Y \mid X)$ directly) vs. **Generative** ($P(X \mid Y)P(Y)$ via Bayes' Rule).
  - Binary, Multiclass, and Multi-Label task formulations.
- **Real-World Regression Applications**: Continuous value estimation spans finance (CAPM, house pricing), medicine (blood pressure, hospital LOS), and engineering (RUL, load forecasting).
- **Practical Implementation**: Scikit-learn provides unified interfaces for training, evaluating, and deploying both classification and regression models.

Introduction to Classification I

What is Classification?

Classification is a core supervised machine learning task where the objective is to learn a decision mapping function $f : \mathcal{X} \to \mathcal{Y}$ from an input feature space $\mathcal{X} \subseteq \mathbb{R}^d$ to a discrete categorical label space $\mathcal{Y}$.

- **Labeled Training Dataset**: $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$, where:
  - $x^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}]^T \in \mathbb{R}^d$ denotes the d-dimensional feature vector.
  - $y^{(i)} \in \mathcal{C} = \{c_1, c_2, \ldots, c_K\}$ represents the discrete target class label.
- **Primary Goal**: Accurately predict the discrete class label $\hat{y} \in \mathcal{C}$ for novel, unseen input feature vectors x_{new}.
- **Classification vs. Regression**:
  - **Regression**: Predicts continuous outputs ($y \in \mathbb{R}$).
  - **Classification**: Predicts qualitative, discrete categories ($y \in \{c_1, \ldots, c_K\}$).

Taxonomy of Classification Problems I

Binary Classification

- **Target Space**: $\mathcal{Y} \in \{0, 1\}$ or $\{-1, +1\}$.
- Two mutually exclusive outcomes (Positive vs. Negative class).
- **Examples**:
  - Email classification (Spam / Non-Spam)
  - Medical diagnosis (Disease / Healthy)
  - Credit risk (Default / Non-Default)

Multi-class Classification

- **Target Space**: $\mathcal{Y} \in \{1, 2, \ldots, K\}$ with $K > 2$.
- Each instance belongs to exactly one category.
- **Examples**: Optical Character Recognition (digits $0 - 9$), Iris species classification.

Multi-label Classification

- **Target Space**: $\mathbf{y} \in \{0, 1\}^K$.
- Each instance can simultaneously belong to multiple classes.
- **Examples**: Image tag generation ("cat", "outdoor", "daylight"), Article topic categorization.

Structured Output Classification

- Target is a structured entity (sequence, tree, or graph).
- **Examples**: Part-of-Speech (POS) tagging in Natural Language Processing.

Decision Boundaries & Mathematical Formalization I

Decision Surface / Boundary

A decision boundary is a hyper-surface in feature space $\mathbb{R}^d$ partitioning regions assigned to different target classes.

$$\mathcal{S}_{ij} = \{x \in \mathbb{R}^d \mid P(Y = c_i \mid X = x) = P(Y = c_j \mid X = x)\}$$

- **Linear Decision Boundaries**: Separated by a linear hyperplane ($w^T x + b = 0$). Algorithms include Logistic Regression and Linear Support Vector Machines (SVM).
- **Non-linear Decision Boundaries**: Complex surfaces constructed by non-linear models such as Decision Trees, Kernel SVMs, and Neural Networks.
- **Decision Rule (Bayes Optimal Classifier)**:
$$h^*(x) = \arg\max_{c_k \in \mathcal{C}} P(Y = c_k \mid X = x)$$
- **0-1 Loss Metric**: $\mathcal{L}_{0-1}(y, \hat{y}) = \mathbb{I}(y \neq \hat{y})$ penalizes any misclassification with unit cost.

Key Performance Metrics for Classification I

The Binary Confusion Matrix

	Predicted Positive ($\hat{y} = 1$)	**Predicted Negative ($\hat{y} = 0$)**
Actual Positive ($y = 1$)	True Positive (TP)	False Negative (FN)
Actual Negative ($y = 0$)	False Positive (FP)	True Negative (TN)

Accuracy & Precision

- **Accuracy**: Proportion of correct predictions

$$\text{Acc} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- **Precision**: Fidelity of positive calls

$$\text{Prec} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall & F1-Score

- **Recall (Sensitivity)**: Positive coverage

$$\text{Rec} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F1-Score**: Harmonic mean

$$F_1 = 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}$$

Python Implementation: Building a Classifier I

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# 1. Synthesize binary classification dataset
X, y = make_classification(
    n_samples=500, n_features=4, n_classes=2, random_state=42
)

# 2. Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Initialize and train binary Logistic Regression classifier
model = LogisticRegression()
model.fit(X_train, y_train)

# 4. Inference and performance evaluation
y_pred = model.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred))
```

Real-World Applications & Core Takeaways I

Real-World Applications of Classification

- **Healthcare & Medicine**: Tumor malignancy diagnosis from diagnostic imaging.
- **Computer Vision**: Facial identification, autonomous vehicle traffic sign recognition.
- **Natural Language Processing**: Spam filtering, customer sentiment analysis.
- **Finance & Banking**: Credit card transaction fraud detection and credit score rating.

Summary of Key Concepts

- **Definition**: Learning a mapping $f : \mathcal{X} \to \mathcal{Y}$ from feature vectors to categorical labels.
- **Decision Boundary**: Geometry separating class regions in input space.
- **Metric Selection**: Tailor metrics (Accuracy vs. Precision/Recall/F1) to handle class imbalance and asymmetric error costs.

Overview of Classification Tasks I

Classification Definition

Classification is a supervised learning paradigm where the goal is to learn a mapping function $f : \mathcal{X} \to \mathcal{Y}$ from feature vectors $\boldsymbol{x} \in \mathbb{R}^d$ to categorical labels $y \in \mathcal{Y}$.

Binary Classification

- **Label Space**: $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{-1, +1\}$.
- **Goal**: Partition feature space into two mutually exclusive regions.
- **Examples**: Spam vs. Non-Spam, Tumor Benign vs. Malignant.

Multi-class Classification

- **Label Space**: $\mathcal{Y} = \{1, 2, \ldots, K\}$ where $K > 2$.
- **Goal**: Assign each sample to exactly one of K discrete categories.
- **Examples**: Handwritten digit recognition ($0 - 9$), Crop disease identification.

Key Distinction

In multi-class classification, classes are **mutually exclusive** (each instance belongs to exactly one class). This differs from multi-label classification where an instance can belong to multiple classes simultaneously.

Binary Classification: Mathematical Framework I

Probabilistic Output & Hypothesis Space

For a binary target $y \in \{0, 1\}$, a probabilistic binary classifier models the posterior distribution $P(Y = 1 \mid \boldsymbol{x})$.

- **Model Hypothesis**: $\hat{p}(\boldsymbol{x}) = \sigma(f(\boldsymbol{x})) = \frac{1}{1+e^{-f(\boldsymbol{x})}} \in [0, 1]$.
- **Decision Rule**:

$$\hat{y} = h(\boldsymbol{x}) = \begin{cases} 1 & \text{if } P(Y = 1 \mid \boldsymbol{x}) \geq \tau \\ 0 & \text{if } P(Y = 1 \mid \boldsymbol{x}) < \tau \end{cases} \quad \text{(default threshold } \tau = 0.5\text{)}$$

Binary Loss Function: Log Loss / Cross-Entropy

Given N samples, parameters are optimized by minimizing the Binary Cross-Entropy (BCE) loss:

$$\mathcal{L}_{\text{BCE}}(\boldsymbol{w}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

Binary Classification: Decision Boundaries I

Linear vs. Non-linear Boundaries

The decision boundary is the hypersurface in $\mathbb{R}^d$ where the classifier is indifferent between classes ($\hat{p}(\boldsymbol{x}) = 0.5 \iff f(\boldsymbol{x}) = 0$).

Linear Decision Boundary

- **Form**: $\boldsymbol{w}^T \boldsymbol{x} + b = 0$.
- Separates classes using a straight line ($d = 2$) or hyperplane ($d \geq 3$).
- **Algorithms**: Logistic Regression, Linear SVM, Perceptron.

Non-linear Decision Boundary

- **Form**: Complex surface in feature space.
- Captures non-linear dependencies without manual feature engineering.
- **Algorithms**: Kernel SVM, Decision Trees, Neural Networks, k-NN.

Multi-class Classification: Problem Setup I

Target Representation & Softmax Extension

For $K > 2$ classes, targets are represented using one-hot encoding: $\boldsymbol{y}_i \in \{0, 1\}^K$ where $\sum_{k=1}^{K} y_{ik} = 1$.

Softmax Activation Function

To output a valid probability distribution over K classes from raw logits $\boldsymbol{z} = [z_1, z_2, \ldots, z_K]^T$:

$$P(Y = k \mid \boldsymbol{x}) = \sigma(\boldsymbol{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{such that} \quad \sum_{k=1}^{K} P(Y = k \mid \boldsymbol{x}) = 1$$

Categorical Cross-Entropy Loss

Optimization objective across N training examples:

$$\mathcal{L}_{\text{CCE}}(\boldsymbol{W}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(\hat{p}_{ik})$$

Decomposing Multi-class: One-vs-Rest (OvR) I

Strategy Principle

Also known as **One-vs-All (OvA)**. Reduces a K-class problem into K independent binary classification tasks.

Training Phase

- Train K separate binary classifiers $f_1, f_2, \ldots, f_K$.
- For classifier f_k:
  - Class k samples $\rightarrow$ Positive $(+1)$.
  - All other $K-1$ classes $\rightarrow$ Negative (0).

Inference Phase

- Pass test sample $\boldsymbol{x}$ through all K classifiers.
- Assign label with the maximum confidence score:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} f_k(\boldsymbol{x})$$

Limitation of OvR

Can introduce severe class imbalance during binary training (1 vs. $K-1$ ratio) and scale calibration mismatches across unstandardized classifiers.

Decomposing Multi-class: One-vs-One (OvO) I

Strategy Principle

Constructs a dedicated binary classifier for every unique pair of classes, completely isolating pair-wise decision boundaries.

Model Complexity

- Number of binary classifiers:

$$M = \binom{K}{2} = \frac{K(K-1)}{2}$$

- For $K = 10$ classes $\Rightarrow$ 45 classifiers.
- For $K = 100$ classes $\Rightarrow$ 4,950 classifiers.

Voting & Decision

- Each classifier f_{ij} votes for either class i or class j.
- Final prediction uses majority voting:

$$\hat{y} = \arg\max_k \sum_{i<j} \mathbb{I}(f_{ij}(\boldsymbol{x}) = k)$$

Advantage

Each classifier is trained on a small subset of data (only instances belonging to classes i and j), making training computationally efficient per classifier.

Native Multi-class Classification Algorithms I

Direct Multi-class Modeling

Instead of binary decomposition wrappers (OvR/OvO), several algorithms natively handle $K > 2$ classes within their mathematical formulation.

Multinomial Logistic Regression Uses vector-valued linear outputs with Softmax normalization and cross-entropy optimization.

Decision Trees & Random Forests Calculates split criteria (Gini Impurity, Information Gain) over categorical target distribution across all K classes:

$$\text{Gini}(p) = 1 - \sum_{k=1}^{K} p_k^2$$

k-Nearest Neighbors (k-NN) Computes class frequencies among the k nearest neighbors and assigns the mode class.

Naive Bayes Computes joint class probabilities using Bayes' theorem:

$$P(Y = k \mid \boldsymbol{x}) \propto P(Y = k) \prod_{j=1}^{d} P(x_j \mid Y = k)$$

Evaluation Metrics: Multi-class Averaging I

Extending Binary Metrics (Precision, Recall, F1)

In multi-class settings, per-class metrics must be aggregated to provide a global performance summary.

Macro-Averaging

- Unweighted mean across all classes:

$$\text{F1}_{\text{macro}} = \frac{1}{K} \sum_{k=1}^{K} \text{F1}_k$$

- Treats all classes equally; highlights performance on minority classes.

Micro & Weighted Averaging

- **Micro**: Aggregates total TP, FP, FN across all classes first.
- **Weighted**: Class-frequency weighted mean:

$$\text{F1}_{\text{weighted}} = \sum_{k=1}^{K} \frac{N_k}{N} \text{F1}_k$$

Python Implementation: Scikit-Learn Pipelines I

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.metrics import classification_report

# Generate synthetic 4-class classification dataset
X, y = make_classification(n_samples=1000, n_features=10,
                           n_classes=4, n_informative=6, random_state=42)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=42)

# 1. Native Softmax Multi-class Regression
clf_softmax = LogisticRegression(multi_class='multinomial', solver='lbfgs')
clf_softmax.fit(X_tr, y_tr)

# 2. Explicit One-vs-Rest Meta-estimator
clf_ovr = OneVsRestClassifier(LogisticRegression())
clf_ovr.fit(X_tr, y_tr)

# 3. Explicit One-vs-One Meta-estimator
clf_ovo = OneVsOneClassifier(LogisticRegression())
clf_ovo.fit(X_tr, y_tr)

# Evaluation
print(classification_report(y_te, clf_softmax.predict(X_te)))
```

Comparative Summary: Multi-class Strategies I

Methodology Comparison Matrix

Property	One-vs-Rest (OvR)	One-vs-One (OvO)	Native (Softmax)
No. of Models	K	$\frac{K(K-1)}{2}$	1
Sub-dataset Size	Full dataset (N)	Subset ($N_i + N_j$)	Full dataset (N)
Train Time Complexity	$\mathcal{O}(K \cdot T_{\text{binary}})$	$\mathcal{O}(K^2 \cdot T_{\text{sub}})$	Directly optimized
Output Type	Independent scores	Pairwise votes	Calibrated probs
Imbalance Sensitivity	High (1 vs $K-1$)	Low (Pairwise)	Handled via Loss
Best Suited For	Fast baseline, $K \leq 10$	SVMs, Kernels	Neural Nets, Trees

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 4**: Supervised Machine Learning
- **Topic 4.2.3**: k-Nearest Neighbor (kNN) Algorithm

Learning Objectives:

1. Understand the core concepts of **Instance-Based Learning** and **Lazy Learning**.
2. Master the mathematical formulation of distance metrics (Euclidean, Manhattan, Minkowski, Cosine).
3. Understand the step-by-step working mechanism of k-NN for classification and regression.
4. Analyze hyperparameter selection (k) and its impact on the **Bias-Variance Tradeoff**.
5. Evaluate the main advantages, limitations, computational complexity, and mitigation strategies (KD-Tree, Ball-Tree).
6. Implement a complete k-NN pipeline in Python using `scikit-learn`.

Introduction to k-Nearest Neighbor (kNN) I

What is k-NN?

The **k-Nearest Neighbor (k-NN)** algorithm is a non-parametric, instance-based supervised learning method used for both **classification** and **regression**.

- **Core Intuition**: "Birds of a feather flock together." Data points that are close to each other in feature space are likely to belong to the same class or have similar target values.
- **Instance-Based Learning**: The algorithm does not construct an explicit internal model or target function during training; instead, it stores the entire training dataset.
- **Formal Setup**: Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y}$, for a query point $\boldsymbol{x}_0$, k-NN finds the set $\mathcal{N}_k(\boldsymbol{x}_0)$ of k closest training points.

Need for k-NN: Lazy vs. Eager Learning I

Eager Learning Paradigm

- **Examples**: Decision Trees, Logistic Regression, Neural Networks.
- **Training Phase**: Construct an explicit generalization model $f(\boldsymbol{x}; \boldsymbol{w})$ before receiving test queries. High training time complexity $\mathcal{O}(\text{epochs} \cdot N \cdot d)$.
- **Inference Phase**: Fast prediction $\mathcal{O}(d)$ by evaluating $f(\boldsymbol{x}_0)$. Training data can be discarded.

Lazy Learning (k-NN) Paradigm

- **Training Phase**: Zero training work ($\mathcal{O}(1)$ time). Simply store training instances $\mathcal{D}$ in memory.
- **Inference Phase**: Computation is deferred to prediction time. Evaluates distance to all stored points ($\mathcal{O}(N \cdot d)$).
- **Non-Parametric**: No prior assumptions about the probability distribution of data.

Distance Metrics in k-NN I

Measuring Proximity in Feature Space

The performance of k-NN depends heavily on the choice of distance metric $d(\boldsymbol{x}, \boldsymbol{z})$ between vectors $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^d$:

- **Euclidean Distance** (L_2 Norm): Most common for continuous variables.

$$d_2(\boldsymbol{x}, \boldsymbol{z}) = \sqrt{\sum_{j=1}^{d} (x_j - z_j)^2} = \|\boldsymbol{x} - \boldsymbol{z}\|_2$$

- **Manhattan Distance** (L_1 Norm / City-Block): Preferred for high-dimensional or grid-like data.

$$d_1(\boldsymbol{x}, \boldsymbol{z}) = \sum_{j=1}^{d} |x_j - z_j| = \|\boldsymbol{x} - \boldsymbol{z}\|_1$$

- **Minkowski Distance** (L_p Norm Generalized):

$$d_p(\boldsymbol{x}, \boldsymbol{z}) = \left(\sum_{j=1}^{d} |x_j - z_j|^p \right)^{\frac{1}{p}} \quad (p = 1 \Rightarrow \text{Manhattan}, p = 2 \Rightarrow \text{Euclidean})$$

Distance Metrics in k-NN II

- **Cosine Distance**: Measures orientation angle regardless of magnitude (useful for text vectors).

$$d_{\cos}(\boldsymbol{x}, \boldsymbol{z}) = 1 - \frac{\boldsymbol{x} \cdot \boldsymbol{z}}{\|\boldsymbol{x}\|_2 \|\boldsymbol{z}\|_2}$$

Working Mechanism of k-NN I

Algorithmic Workflow for Query Point x_0

1. **Parameter Choice**: Select integer $k \geq 1$ and distance metric $d(\cdot, \cdot)$.
2. **Distance Computation**: Compute distance $d(\boldsymbol{x}_0, \boldsymbol{x}_i)$ for all $i = 1, 2, \ldots, N$.
3. **Neighbor Identification**: Identify the set $\mathcal{N}_k(\boldsymbol{x}_0)$ containing the k training points with smallest distances.
4. **Target Prediction**:
   - **Majority Vote (Classification)**:

$$\hat{y}_0 = \arg\max_{c \in \mathcal{Y}} \sum_{i \in \mathcal{N}_k(\boldsymbol{x}_0)} \mathbb{I}(y_i = c)$$

   - **Distance-Weighted Majority Vote**: Assign higher weights $w_i = \frac{1}{d(\boldsymbol{x}_0, \boldsymbol{x}_i)^2 + \epsilon}$ to closer neighbors.
   - **Mean Average (Regression)**:

$$\hat{y}_0 = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\boldsymbol{x}_0)} y_i$$

Selecting the Value of k: Hyperparameter Tuning I

Impact of k on Model Capacity & Decision Surface

Small k (e.g., $k = 1$)

- Highly flexible, complex decision boundary.
- Low Bias, High Variance.
- Sensitive to noise, outliers, and mislabeled data.
- Risk of **Overfitting**.

Large k (e.g., $k \to N$)

- Smooth, overly simplified decision boundary.
- High Bias, Low Variance.
- Predicts majority class everywhere as $k \to N$.
- Risk of **Underfitting**.

Best Practices for Choosing k

- Use an **odd value for** k in binary classification to prevent tie votes.
- Rule of thumb starter value: $k = \lfloor \sqrt{N} \rfloor$.
- **Grid Search with Cross-Validation**: Select k that minimizes validation error across candidate values $k \in \{1, 3, 5, 7, \ldots, 25\}$.

Critical Requirement: Feature Scaling I

Sensitivity to Unscaled Features

Because k-NN relies strictly on geometric distances, features with larger numeric ranges dominate distance calculations, rendering small-scale features irrelevant.

Example Illustrating Feature Dominance

Consider two features: **Income** ($\$20,000 - \$150,000$) vs. **Age** ($18 - 65$ years). Without scaling, a difference of $\$1,000$ in income completely overwhelms a 40-year difference in age during Euclidean distance evaluation.

Scaling Solutions

- **Standardization (Z-Score Normalization)**: (Recommended when data is Gaussian)

$$x_{\text{std}} = \frac{x - \mu}{\sigma} \quad \Rightarrow \quad \mu = 0,\ \sigma = 1$$

- **Min-Max Normalization**: (Scales values strictly to $[0, 1]$)

$$x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

Advantages of k-NN Algorithm I

Key Strengths

1. **Simplicity & Intuition**: Very easy to understand, explain, and implement with minimal mathematical complexity.
2. **Zero Training Cost**: Instant training step ($\mathcal{O}(1)$) since data points are stored as-is without optimization loops.
3. **No Distributional Assumptions**: Non-parametric algorithm that naturally handles arbitrary multi-modal distributions and complex decision surfaces.
4. **Natural Multi-Class Extension**: Inherently supports multi-class classification and multi-output regression without requiring One-vs-Rest wrappers.
5. **Dynamic Data Updates**: New training samples can be continuously added to memory without retraining the model.

Disadvantages & Practical Challenges I

Key Limitations

1. **High Computational Overhead at Prediction**: Searching N points in d dimensions takes $\mathcal{O}(N \cdot d)$ time per test sample. Prohibitive for real-time applications on large datasets.
2. **High Memory Consumption**: Must store entire dataset in RAM ($\mathcal{O}(N \cdot d)$ space complexity).
3. **Curse of Dimensionality**: In high-dimensional spaces ($d \gg 10$), the distance between any pair of points converges to nearly the same value, causing loss of discriminative power.
4. **Sensitivity to Outliers & Class Imbalance**: Outliers can skew predictions; majority classes dominate random voting.
5. **Sensitivity to Irrelevant Features**: Uninformative features corrupt distance metrics unless pruned or scaled.

Computational Efficiency: Indexing Structures I

Accelerating Neighbor Search beyond Brute-Force $\mathcal{O}(N \cdot d)$

To avoid computing distance to all N samples, specialized tree structures partition space hierarchically:

KD-Tree (k-dimensional Tree)

- Recursively splits data along axis-aligned hyperplanes.
- Search Complexity: Average $\mathcal{O}(d \cdot \log N)$.
- Performance degrades when dimension $d > 20$ (falls back to brute-force speed).

Ball Tree

- Partitions data into nested hyperspheres ("balls").
- Search Complexity: Efficient in higher dimensions ($d > 20$) where axis-aligned splits fail.
- More expensive to construct than KD-Tree.

Python Implementation: Complete k-NN Pipeline I

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline

# 1. Generate Synthetic Classification Dataset
X, y = make_classification(n_samples=500, n_features=4,
                           n_informative=3, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 2. Build Pipeline (Scaling + k-NN Classifier)
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(metric='minkowski', p=2)) # Euclidean
])

# 3. Hyperparameter Tuning for optimal 'k' via 5-Fold Cross Validation
param_grid = {'knn__n_neighbors': list(range(1, 21, 2))} # Odd numbers 1 to 19
grid = GridSearchCV(pipeline, param_grid, cv=5, scoring='accuracy')
grid.fit(X_train, y_train)

# 4. Evaluate Best Model
print(f"Optimal k: {grid.best_params_['knn__n_neighbors']}")
print(f"Test Accuracy: {grid.score(X_test, y_test):.4f}")
```

Python Implementation: Complete k-NN Pipeline II

Summary & Practical Recommendations I

k-NN Algorithm Summary Table

Aspect	Characteristics / Recommendation
Model Type	Non-parametric, Instance-based, Lazy Learner
Training Complexity	$\mathcal{O}(1)$ time, $\mathcal{O}(N \cdot d)$ storage space
Prediction Complexity	$\mathcal{O}(N \cdot d)$ brute-force; $\mathcal{O}(d \log N)$ with KD-Tree
Primary Preprocessing	Mandatory Feature Standardization / Normalization
Hyperparameter k	Tune via Cross-Validation; use odd numbers for binary targets
Best Suited For	Low-dimensional ($d \leq 20$), small-to-medium dataset sizes

Golden Rule for Practical Deployment

Always pair k-NN with a feature scaling preprocessor (e.g. `StandardScaler`) and reduce dimensions (e.g. via PCA) if feature dimension d is large.

Introduction to Support Vector Machines (SVM) I

What is a Support Vector Machine?

Support Vector Machines (SVMs) are powerful supervised learning algorithms used primarily for classification and regression. The core philosophy of SVM is to construct an **optimal decision hyperplane** that maximizes the margin of separation between classes in feature space.

Key Characteristics

- **Maximum Margin Classifier**: Finds the decision boundary furthest from closest samples.
- **Sparsity**: Decision boundary depends strictly on a subset of samples called *Support Vectors*.
- **Convexity**: Solves a convex quadratic program guaranteed to reach global optimum.

Why Maximize Margin?

- **Generalization**: Larger margins provide maximum safety margin against noise and unseen test instances.
- **Robustness**: Implements structural risk minimization rather than purely empirical risk.
- **Stability**: Prevents boundary from overfitting to isolated training points.

Geometric Foundations: Hyperplanes & Distance I

Hyperplane Geometry

In a d-dimensional feature space $\mathbb{R}^d$, an affine hyperplane separating two classes is defined by:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b = 0$$

where $\boldsymbol{w} \in \mathbb{R}^d$ is the weight vector normal (orthogonal) to the hyperplane, and $b \in \mathbb{R}$ is the bias offset.

Perpendicular Distance of a Point to Hyperplane

The signed Euclidean distance r from any arbitrary vector $\boldsymbol{x}_i \in \mathbb{R}^d$ to the decision plane $\boldsymbol{w}^T \boldsymbol{x} + b = 0$ is:

$$r = \frac{\boldsymbol{w}^T \boldsymbol{x}_i + b}{\|\boldsymbol{w}\|_2}$$

Given binary targets $y_i \in \{-1, +1\}$, the correct geometric distance r_i from $\boldsymbol{x}_i$ to the boundary is:

$$r_i = \frac{y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2}$$

Functional vs. Geometric Margins I

Functional Margin

For a sample $(\boldsymbol{x}_i, y_i)$, the functional margin is:

$$\hat{\gamma}_i = y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)$$

- Measures algebraic correctness and confidence.
- **Scale Sensitivity**: Scaling $(\boldsymbol{w}, b) \to (\alpha \boldsymbol{w}, \alpha b)$ inflates $\hat{\gamma}_i$ arbitrarily without shifting geometry.

Geometric Margin

The geometric margin normalizes by the L2 norm of $\boldsymbol{w}$:

$$\gamma_i = \frac{y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} = \frac{\hat{\gamma}_i}{\|\boldsymbol{w}\|_2}$$

- Represents true geometric distance in feature space.
- Completely invariant to scaling parameters by $\alpha > 0$.

Canonical Hyperplane Convention

To resolve scale ambiguity, we set the functional margin of the closest point(s) to 1: $\min_i \hat{\gamma}_i = 1$. Consequently, the geometric margin of closest points becomes $\gamma = \frac{1}{\|\boldsymbol{w}\|_2}$.

Hard-Margin SVM: Optimization Formulation I

Primal Optimization Problem

For a linearly separable dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ with $y_i \in \{-1, +1\}$, maximizing the margin $\frac{1}{\|\boldsymbol{w}\|_2}$ is equivalent to minimizing $\frac{1}{2}\|\boldsymbol{w}\|_2^2$:

$$\begin{aligned} \min_{\boldsymbol{w}, b} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 \\ \text{subject to} \quad & y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, N \end{aligned}$$

Convex Quadratic Program

- Strictly convex quadratic objective function.
- N linear inequality constraints.
- Guaranteed single global optimum; solvable via QP solvers.

Total Margin Width

- Positive boundary: $\boldsymbol{w}^T \boldsymbol{x} + b = +1$.
- Negative boundary: $\boldsymbol{w}^T \boldsymbol{x} + b = -1$.
- Total margin distance between boundary boundaries:

$$M = \frac{2}{\|\boldsymbol{w}\|_2}$$

Support Vectors & Margin Geometry I

Definition of Support Vectors

Support Vectors are the subset of training examples $\boldsymbol{x}_i$ that lie exactly on the marginal hyperplanes where constraint equality is met:

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1$$

Properties of Support Vectors

- Non-support vector points can be removed or moved without altering $(\boldsymbol{w}^*, b^*)$.
- The model parameters $\boldsymbol{w}$ depend **only** on support vectors.
- Memory-efficient and fast decision boundary evaluations.

Limitations of Hard-Margin

- Extremely sensitive to single outliers or noise.
- Strictly fails (infeasible QP) if dataset has any non-linear overlap.
- Motivates soft-margin relaxation with slack variables.

Soft-Margin SVM: Handling Non-Separable Data I

Slack Variables (ξ_i)

To handle non-linearly separable data, we introduce non-negative **slack variables** $\xi_i \geq 0$ allowing samples to violate the strict margin constraint:

- $\xi_i = 0$: Point lies outside or directly on the correct margin ($y_i f(\boldsymbol{x}_i) \geq 1$).
- $0 < \xi_i \leq 1$: Point is within the margin but correctly classified ($0 \leq y_i f(\boldsymbol{x}_i) < 1$).
- $\xi_i > 1$: Point is misclassified on the wrong side of hyperplane ($y_i f(\boldsymbol{x}_i) < 0$).

Soft-Margin Primal Formulation

$$\min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \quad \frac{1}{2} \|\boldsymbol{w}\|_2^2 + C \sum_{i=1}^{N} \xi_i$$

$$\text{subject to} \quad y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \forall i = 1, \ldots, N$$

where $C > 0$ is the regularization trade-off hyperparameter.

Hinge Loss Formulation I

Unconstrained Risk Minimization

By setting $\xi_i = \max(0, 1 - y_i f(\boldsymbol{x}_i))$, the Soft-Margin SVM problem can be rewritten as unconstrained loss minimization with L2 regularization:

$$\min_{\boldsymbol{w},b} \underbrace{\sum_{i=1}^{N} \max\left(0, 1 - y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\right)}_{\text{Hinge Loss Sum}} + \frac{1}{2C}\|\boldsymbol{w}\|_2^2$$

Hinge Loss Function

$$\mathcal{L}_{\text{Hinge}}(z) = \max(0, 1 - z) \quad \text{where } z = y_i f(\boldsymbol{x}_i)$$

- Zero penalty for points with margin score $z \geq 1$.
- Linear penalty for margin score $z < 1$.

Sparsity vs. Log-Loss

- **Log Loss** (Logistic Regression) is strictly positive everywhere $\Rightarrow$ every sample affects $\boldsymbol{w}$.
- **Hinge Loss** is zero for $z \geq 1 \Rightarrow$ leads to *sparse* support vectors.

Hyperparameter C & Feature Scaling I

Role of Hyperparameter C

Hyperparameter C dictates the trade-off between maximizing margin width and penalizing slack errors $\sum \xi_i$:

Large C (Hard Margin Behavior)

- Heavy penalty on violations $\Rightarrow$ narrow margin.
- Fits training data aggressively.
- Low bias, high variance (risk of **overfitting**).

Small C (Soft Margin Behavior)

- Light penalty on violations $\Rightarrow$ wider margin.
- Allows more margin errors for better smoothness.
- High bias, low variance (risk of **underfitting**).

Crucial Necessity: Feature Standardization

Because SVM computes distance $\|\boldsymbol{w}\|_2$, unscaled features with large numerical ranges will dominate distance calculations and distort the margin. **Always apply StandardScaler** before fitting SVMs!

Python Implementation: Linear SVM Pipeline I

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# 1. Generate synthetic binary classification dataset
X, y = make_classification(n_samples=500, n_features=4, n_informative=2,
                           n_redundant=0, n_clusters_per_class=1,
                           class_sep=1.5, random_state=42)

# Map labels {0, 1} to {-1, +1} standard SVM format
y_svm = np.where(y == 0, -1, 1)

# 2. Train/Test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y_svm, test_size=0.2, random_state=42
)

# 3. Build pipeline: Feature Standardization + Linear SVC
svm_pipeline = make_pipeline(
    StandardScaler(),
    SVC(kernel='linear', C=1.0, random_state=42)
)
svm_pipeline.fit(X_train, y_train)
```

Python Implementation: Linear SVM Pipeline II

```
# 4. Model Evaluation
y_pred = svm_pipeline.predict(X_test)
print(f"Test Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred))
```

Extracting Decision Boundaries & Support Vectors I

```
# Extract pipeline components
scaler = svm_pipeline.named_steps['standardscaler']
model = svm_pipeline.named_steps['svc']

# Access hyperplane parameters: w^T x + b = 0
w = model.coef_[0]          # Weight vector normal to boundary
b = model.intercept_[0]     # Bias offset

# Extract support vector properties
sv_indices = model.support_
support_vectors = model.support_vectors_
n_sv_per_class = model.n_support_

print(f"Weight vector (w): {np.round(w, 4)}")
print(f"Bias (b): {b:.4f}")
print(f"Support Vector Counts (class -1 / +1): {n_sv_per_class}")

# Compute exact geometric margin M = 2 / ||w||
w_norm = np.linalg.norm(w)
margin_width = 2.0 / w_norm
print(f"Geometric Margin Width (M): {margin_width:.4f}")

# Calculate test functional margins y * (w^T x + b)
X_test_scaled = scaler.transform(X_test)
scores = model.decision_function(X_test_scaled)
functional_margins = y_test * scores
print(f"Min Test Functional Margin: {functional_margins.min():.4f}")
```

Linear SVM vs. Logistic Regression I

Methodological Comparison

Property	Linear SVM	Logistic Regression
Loss Function	Hinge Loss $\max(0, 1 - yf(\boldsymbol{x}))$	Log Loss $\log(1 + e^{-yf(\boldsymbol{x})})$
Optimization	Quadratic Programming (Convex)	Gradient Descent / L-BFGS
Sparsity	Sparse (SVs only)	Dense (All instances affect $\boldsymbol{w}$)
Prediction	Margin / Distance Score	Calibrated Probability
Outliers	Robust (distant points ignored)	Sensitive (all points pull boundary)
Feature Scaling	Essential	Highly Recommended

Key Takeaways

- Linear SVM maximizes geometric margin $M = \frac{2}{\|\boldsymbol{w}\|_2}$ to achieve strong generalization.
- Soft-margin SVM uses slack variables (ξ_i) and parameter C to handle non-separable data.
- Hinge loss creates exact zeros for points outside the margin, resulting in a sparse support vector model.

Introduction to Unsupervised Learning I

What is Unsupervised Learning?

Unlike Supervised Learning where dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ contains ground-truth labels $y^{(i)}$, **Unsupervised Learning** operates on unlabeled data:

$$\mathcal{D} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)}\}, \quad \boldsymbol{x}^{(i)} \in \mathbb{R}^d$$

The core objective is to uncover underlying patterns, structures, probability densities, or low-dimensional representations within $\mathcal{D}$.

Key Paradigms

- **Clustering**: Grouping similar instances into clusters $S_1, \ldots, S_K$.
- **Dimensionality Reduction**: Projecting d-dim space to k-dim space ($k \ll d$).
- **Density Estimation**: Modeling the data distribution $p(\boldsymbol{x})$.

Modern Frontier: Generative AI

- **Generative Modeling**: Learning $p_\theta(\boldsymbol{x})$ to sample novel synthetic data $\boldsymbol{x}_{\text{new}} \sim p_\theta(\boldsymbol{x})$.
- **Latent Representation**: Mapping complex inputs into structured latent spaces $\boldsymbol{z} \in \mathbb{R}^k$.

Taxonomy of Unsupervised & Generative Learning I

Mathematical Taxonomy

Unsupervised Learning problems can be broadly classified by their mathematical goals:

Partitioning & Grouping Minimize intra-cluster distance: $\arg\min_{\{S_k\}} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in S_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|^2$.

Manifold Learning Learn mapping $\boldsymbol{z} = f(\boldsymbol{x})$ preserving local/global geometry (e.g., PCA, t-SNE, Autoencoders).

Probabilistic Modeling Maximize log-likelihood $\max_\theta \sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}^{(i)})$ or minimize KL divergence $D_{\mathrm{KL}}(p_{\text{data}} \parallel p_\theta)$.

The Generative AI Paradigm Shift

Classical unsupervised learning focuses on **discriminating patterns** (e.g., cluster assignment). Generative AI extends this to **synthesis**: learning latent variable distribution $p(\boldsymbol{z})$ and conditional generator $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$ to model marginal density:

$$p(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) \, d\boldsymbol{z}$$

Overview of Classical vs. Generative Approaches I

2gray!10white

Attribute	Classical Unsupervised	Generative AI
Primary Goal	Cluster, reduce dimensionality, detect anomalies	Model data distribution $p(\boldsymbol{x})$ and synthesize samples
Output Type	Labels, feature scores, projection matrices	High-fidelity synthetic data, embeddings, text/image
Key Algorithms	K-Means, PCA, GMM, DBSCAN, t-SNE	VAEs, GANs, Diffusion Models, Autoregressive Transformers
Latent Space	Geometric subspace (e.g., principal axes)	Continuous probabilistic distribution $p(\boldsymbol{z})$

Case Study Context: Port Decarbonization & Green Hydrogen I

Problem Statement: Maritime Port Decarbonization

Maritime ports are major emission hubs (CO_2, NO_x, PM). Transitioning to **Green Hydrogen** (H_2) requires:

1. **Operational Profiling**: Unsupervised clustering of vessel duty cycles (tugs, ferries, cargo vessels) from raw sensor streams without labels.
2. **Generative Forecasting**: Simulating adoption trajectories and forecasting port-wide emission reductions under stochastic operational profiles.

Stage 1: Duty Profile Discovery

Apply unsupervised clustering (K-Means/GMM) on vessel operational metrics: dwell time, peak power demand (kW), daily fuel burn rate (L/h).

Stage 2: Generative Forecasting

Use generative scenario generation to forecast CO_2 abatement under varying adoption rates and green hydrogen production capacity.

Case Study: Unsupervised Vessel Duty Cycle Clustering I

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Simulated port telemetry: [Peak Power (kW), Daily Dwell (hrs), Idle Ratio]
X_raw = np.array([
    [4500, 14.2, 0.65], [4800, 15.1, 0.70], [4200, 13.8, 0.62], # Container
    [1200,  4.5, 0.20], [1350,  5.0, 0.25], [1100,  4.1, 0.18], # Tugs
    [ 800, 22.0, 0.85], [ 850, 21.5, 0.88], [ 790, 23.1, 0.82]  # Auxiliary
])

# Step 1: Feature Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_raw)

# Step 2: Unsupervised K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(X_scaled)

print("Discovered Vessel Duty Clusters:", cluster_labels)
print("Cluster Centroids (Scaled):\n", kmeans.cluster_centers_)
```

Case Study: Mathematical Formulation for Emission Abatement I

Baseline Emission Model

For vessel cluster $k \in \{1, \ldots, K\}$, baseline annual emissions E_k^0 (metric tons CO_2/yr) are computed from cluster centroids:

$$E_k^0 = N_k \cdot \left(P_k^{\text{peak}} \cdot \text{LF}_k \cdot h_k \cdot \text{EF}_{\text{diesel}} \right) \times 10^{-6}$$

where N_k is vessel count, P_k^{peak} is peak power, LF_k is load factor, h_k is operational hours, and $\text{EF}_{\text{diesel}} \approx 690$ g CO_2/kWh.

Generative Hydrogen Adoption & Emission Reduction

Let $\eta_k(t) \sim \text{Beta}(\alpha_k(t), \beta_k(t))$ be a generative stochastic adoption curve for cluster k at year t. The net port-wide emission reduction $\Delta E(t)$ is:

$$\Delta E(t) = \sum_{k=1}^{K} E_k^0 \cdot \eta_k(t) \cdot \left(1 - \frac{\text{EF}_{\text{H2}}}{\text{EF}_{\text{diesel}}} \right)$$

For green hydrogen produced via renewables, $\text{EF}_{\text{H2}} \approx 0$, yielding $\Delta E(t) = \sum_{k=1}^{K} E_k^0 \cdot \eta_k(t)$.

Case Study: Generative Forecasting of Emission Reductions I

```python
# Generative scenario simulation for Green H2 adoption and CO2 forecasting
np.random.seed(42)
years = np.arange(2025, 2036)
num_simulations = 1000

# Baseline annual emissions per cluster (Metric Tons CO2)
E_baseline = np.array([45000, 12000, 8000]) # Container, Tugs, Auxiliary

# Simulate stochastic adoption trajectories over 10 years
def generate_scenarios(n_sims, n_years):
    # Generative trajectory using sigmoid with random adoption rates
    t = np.linspace(-3, 3, n_years)
    base_sigmoid = 1 / (1 + np.exp(-t))
    # Add Gaussian latent noise to simulate adoption variance
    noise = np.random.normal(0, 0.05, size=(n_sims, n_years))
    trajectories = np.clip(base_sigmoid + noise, 0, 1)
    return trajectories

adoption_sims = generate_scenarios(num_simulations, len(years))
annual_reductions = adoption_sims * np.sum(E_baseline)

mean_reduction = np.mean(annual_reductions, axis=0)
ci_95_lower = np.percentile(annual_reductions, 2.5, axis=0)
ci_95_upper = np.percentile(annual_reductions, 97.5, axis=0)

print(f"Year-2030-Expected-CO2-Abatement:-{mean_reduction[5]:.2f}-Metric-Tons")
print(f"95%-CI:-[{ci_95_lower[5]:.2f},-{ci_95_upper[5]:.2f}]")
```

Summary & Key Takeaways I

Core Takeaways

- **Unsupervised Paradigm**: Extracts latent patterns, density estimations, and clusters without human labeling.
- **Generative AI Evolution**: Transitions from descriptive modeling to predictive scenario generation and probabilistic sample synthesis.
- **Real-World Impact**: Applied to port decarbonization by discovering vessel duty clusters (K-Means) and generating stochastic green hydrogen adoption trajectories for CO_2 abatement forecasting.

Next Steps in Unit 5

In upcoming lectures, we will explore:

- **Lecture 5.2**: Clustering Deep Dive (K-Means, Hierarchical Clustering, DBSCAN).
- **Lecture 5.3**: Dimensionality Reduction & PCA.
- **Lecture 5.4**: Generative AI Architectures (VAEs, GANs, Diffusion Models).

The Unlabeled Data Explosion & Annotation Bottleneck I

The Real-World Data Paradox

In modern enterprise and scientific domain applications, raw, unannotated data is generated at petabyte scale continuously. However, human annotation is exceptionally slow, expensive, and non-scalable.

Supervised Data Bottleneck

- **High Cost**: Requires expert annotators (e.g., radiologists, legal scholars, domain experts).
- **Human Fatigue & Error**: Inter-annotator disagreement skews label quality.
- **Scale Constraint**: Less than 1% of global digital data contains ground-truth labels y.

Unlabeled Data Abundance

- **Zero Marginal Cost**: Server logs, satellite imagery, web text, streaming sensor telemetry.
- **Continuous Ingestion**: Streaming real-time data without human intervention.
- **Massive Scale**: Enables statistical learning of complex distributions $p(\mathbf{x})$.

Core Motivating Imperative

Supervised learning models fail to utilize over 99% of available world data. Unsupervised learning unlocks value directly from unannotated observations $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$.

Mathematical Motivation: Density & Latent Modeling I

Shifting from Conditional $p(y|\mathbf{x})$ to Data Density $p(\mathbf{x})$

While supervised learning estimates conditional boundaries $p(y|\mathbf{x})$, unsupervised learning models the underlying data-generating distribution $p(\mathbf{x})$ or maps data into a structured latent space $\mathbf{Z}$.

Marginal Density Integration

Data points $\mathbf{x} \in \mathbb{R}^d$ are generated via hidden latent factors $\mathbf{z} \in \mathbb{R}^k$:

$$p(\mathbf{x}) = \int_{\mathcal{Z}} p(\mathbf{x} \mid \mathbf{z}; \boldsymbol{\theta}) p(\mathbf{z}) d\mathbf{z}$$

Learning $p(\mathbf{x})$ allows detection of low-density regions (anomalies) and high-density regions (clusters).

Optimization Comparison

- **Supervised Loss**:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

- **Unsupervised Likelihood**:

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \log p(\mathbf{x}_i; \boldsymbol{\theta})$$

Four Primary Drivers for Unsupervised Learning I

Why Unsupervised Techniques are Essential in Modern ML

Unsupervised learning addresses core computational, economic, and operational challenges that supervised methods cannot solve.

1. Label Scarcity & Acquisition Costs Annotating specialized datasets (e.g., gene sequencing, seismic scans, pathology slides) requires thousands of domain-expert hours, making supervised pipelines economically infeasible.
2. Discovery of Unknown Patterns Supervised models can only learn pre-defined class categories. Unsupervised models discover novel sub-populations, taxonomy shifts, and emerging behavior patterns without prior human hypotheses.
3. Novelty & Zero-Day Anomaly Detection Fraudulent transactions, cyber intrusion vectors, and equipment failure modes are inherently rare or completely unseen; supervised models lack target training labels y_{anomaly}.
4. Feature Compression & Manifold Learning Real-world data resides in ultra-high dimensions ($d \gg 10^3$). Unsupervised compression extracts compact latent codes $\mathbf{z} \in \mathbb{R}^k$ ($k \ll d$) to defeat the curse of dimensionality.

Use Case 1: Pattern Discovery & Customer Segmentation I

Unlocking Structure in Unlabeled Behavioral Data

Organizations collect user interaction logs without explicit tags describing behavioral personas. Unsupervised clustering groups instances based on geometric similarity in feature space.

E-Commerce & Fintech

- **Feature Inputs**: Recency, Frequency, Monetary value (RFM), dwell time, click paths.
- **Outcome**: Discover distinct buyer personas (e.g., impulse buyers, bargain hunters, high-value loyalists) for targeted campaigns.

Precision Medicine

- **Feature Inputs**: High-throughput RNA sequencing expression levels ($d > 20{,}000$).
- **Outcome**: Stratify patients into disease sub-types to tailor drug therapies without requiring manual disease labels.

Geometric Criterion

Optimal partitions maximize inter-cluster distance D_{between} while minimizing intra-cluster variance W_{within}.

Use Case 2: Anomaly & Outlier Detection I

The Extreme Class Imbalance Problem

In critical safety and security systems, anomalous events constitute less than 0.01% of total observations. Supervised classifiers suffer from catastrophic false-negative rates due to severe class imbalance.

Financial Fraud Monitoring

- Fraud tactics continuously adapt to bypass known supervised rules.
- Unsupervised density models identify transactions residing in low-density tails of $p(\mathbf{x})$.

Industrial IoT & Predictive Maintenance

- Machine breakdown data is rarely observed prior to catastrophic failure.
- Unsupervised models construct baseline normal operational boundaries and trigger alerts upon deviation.

Zero-Day Vulnerability Defense

Because unsupervised anomaly detection models normal baseline behavior $p(\mathbf{x})$, it successfully flags brand-new (zero-day) attack patterns that have never occurred in history.

Use Case 3: Dimensionality Reduction & Visualization I

Taming High-Dimensional Feature Spaces

High-dimensional datasets ($d > 100$) suffer from numerical instability, distance concentration effects, and severe computational overhead.

The Distance Concentration Effect

As dimensionality $d \to \infty$, the ratio of maximum to minimum Euclidean distance between any two random points approaches 1:

$$\lim_{d \to \infty} \frac{d_{\max} - d_{\min}}{d_{\min}} = 0$$

This renders nearest-neighbor metrics ineffective without dimensionality reduction.

Benefits of Low-Dimensional Mapping

- **Visualization**: Projecting d-dimensional features into $\mathbb{R}^2$ or $\mathbb{R}^3$ via PCA/t-SNE for human inspection.
- **Noise Filtering**: Discarding low-variance components removes measurement noise.
- **Downstream Acceleration**: Reduces train time for downstream ML models.

Use Case 4: Self-Supervised Learning & Foundation Models I

Unsupervised Pre-training: The Bedrock of Generative AI

Modern Foundation Models (LLMs, Vision Transformers, Generative Diffusion Models) are trained using unsupervised / self-supervised objectives over massive unannotated datasets.

Self-Supervised Pre-training

- Create pretext tasks from raw data (e.g., mask next token in text, mask image patches).
- Objective: Predict masked components using contextual representations $\mathbf{z}$.
- Learns universal domain representations without any manual labels.

Downstream Fine-Tuning

- Transfer pre-trained weights to specific downstream tasks.
- Requires only a small fraction (less than 1%) of labeled data (Few-Shot / Zero-Shot learning).
- Dramatically outperforms models trained strictly supervised from scratch.

Generative Modeling Core

Generative AI models (GANs, VAEs, Diffusion) learn the true underlying distribution $p_{\text{data}}(\mathbf{x})$ to sample completely new, realistic data instances $\mathbf{x}_{\text{new}} \sim p_{\text{model}}(\mathbf{x})$.

Comparative Analysis: Learning Paradigms I

Systematic Comparison of Core Machine Learning Paradigms

Understanding when unsupervised learning is required relative to supervised and self-supervised approaches.

Dimension	**Supervised**	**Unsupervised**	**Self-Supervised**
Target Input	Feature $\mathbf{X}$ + Label y	Raw Feature $\mathbf{X}$ only	Raw Feature $\mathbf{X}$ (Self-labeled)
Primary Goal	Map $\mathbf{X} \rightarrow y$	Discover $p(\mathbf{x})$, clusters, latent $\mathbf{Z}$	Pre-train representations $\mathbf{Z}$ via pre-text tasks
Data Scale	Limited by label cost	Unlimited raw data	Unlimited raw data
Human Overhead	Extremely High	Zero / Minimal	Zero for pre-training
Primary Failure	Overfitting / Label Noise	Evaluation ambiguity	Pretext-task misalignment

Python Implementation: Feature Reduction & Anomaly Detection I

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import IsolationForest

# 1. Simulate high-dimensional unlabeled operational data (1000 samples, 20 features)
X_raw, _ = make_blobs(n_samples=950, n_features=20, centers=3, cluster_std=1.2, random_state=42)
# Add 50 extreme unlabeled anomalous points (zero-day events)
anomalies = np.random.uniform(low=-10, high=10, size=(50, 20))
X_complete = np.vstack([X_raw, anomalies])

# 2. Preprocess & standardize features (Zero mean, unit variance)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_complete)

# 3. Unsupervised Dimensionality Reduction via PCA (Retain 90% variance)
pca = PCA(n_components=0.90)
X_pca = pca.fit_transform(X_scaled)
print(f"Original features: {X_scaled.shape[1]} -> Reduced features: {X_pca.shape[1]}")
print(f"Cumulative Variance Ratio: {np.sum(pca.explained_variance_ratio_):.4f}")

# 4. Unsupervised Anomaly Detection using Isolation Forest
iso_forest = IsolationForest(contamination=0.05, random_state=42)
outlier_predictions = iso_forest.fit_predict(X_pca)  # -1 for anomaly, 1 for normal

num_anomalies = np.sum(outlier_predictions == -1)
```

Python Implementation: Feature Reduction & Anomaly Detection II

```
print(f"Detected Unsupervised Outliers: {num_anomalies} out of {len(X_complete)} samples")
```

Python Implementation: Customer Segmentation Pipeline I

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Simulate unlabeled customer behavioral telemetry (Recency, Frequency, Monetary, Dwell)
np.random.seed(42)
telemetry = np.random.randn(600, 4) * np.array([15, 5, 200, 12]) + np.array([30, 10, 500, 25])

# Standardize feature matrix
scaler = StandardScaler()
X_norm = scaler.fit_transform(telemetry)

# Evaluate optimal cluster count (K) without ground truth labels
sil_scores = []
k_range = range(2, 7)

for k in k_range:
    km = KMeans(n_clusters=k, init='k-means++', n_init=10, random_state=42)
    labels = km.fit_predict(X_norm)
    score = silhouette_score(X_norm, labels)
    sil_scores.append(score)
    ch_score = calinski_harabasz_score(X_norm, labels)
    print(f"K={k} | Silhouette: {score:.4f} | Calinski-Harabasz: {ch_score:.2f}")

optimal_k = k_range[np.argmax(sil_scores)]
print(f"Optimal discovered customer segments (unsupervised): K={optimal_k}")
```

Strategic Framework: When to Deploy Unsupervised Learning I

Decision Protocol for ML Engineers

Use this workflow to determine whether unsupervised learning is the mandatory or optimal choice for a problem domain.

1. **Is target annotation y unavailable or prohibitively expensive?**
   - **Yes**: Deploy Unsupervised Clustering / Self-Supervised Representation Learning.
2. **Is the objective to detect rare, unseen, or zero-day anomalies?**
   - **Yes**: Deploy Unsupervised Density Estimation (Isolation Forest, GMMs, One-Class SVM).
3. **Is feature dimension d extremely large, causing model overfitting or high latency?**
   - **Yes**: Apply Unsupervised Dimensionality Reduction (PCA, UMAP, Autoencoders) as a preprocessing module.
4. **Is the objective to explore data structure without pre-conceived biases?**
   - **Yes**: Perform Unsupervised Clustering and Manifold Visualization.

End-to-End Operational Pipeline I

The Core Unsupervised Workflow

Unsupervised learning algorithms transform an unannotated raw feature matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ into structured patterns, latent representations, or clusters without target labels y_i.

Stage 1: Ingestion & Preprocessing

- **Data Normalization**: Apply **Z**-score or MinMax scaling to prevent high-magnitude features from dominating distance functions.
- **Imputation & Cleaning**: Handle missing attributes and remove severe noise points.

Stage 2: Distance & Structure Matrix

- Compute pairwise distance matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ or covariance matrix $\mathbf{C} \in \mathbb{R}^{d \times d}$.
- Construct similarity kernels or graph adjacency matrices for non-linear structures.

Stage 3: Optimization & Partitioning

- **Iterative Optimization**: Minimize Within-Cluster Sum of Squares (WCSS) or maximize expected log-likelihood via EM.
- **Matrix Factorization**: Extract top eigenvectors or singular values via SVD/PCA.

Stage 4: Latent Mapping & Validation

- Project instances into low-dimensional space $\mathbf{Z} \in \mathbb{R}^{N \times k}$ ($k \ll d$).
- Evaluate partition quality using internal metrics (Silhouette, Davies-Bouldin).

Data Preprocessing & Scale Sensitivity I

Why Preprocessing is Critical in Unsupervised Learning

Unlike supervised learning where decision trees or target loss functions can compensate for scale differences, unsupervised algorithms rely strictly on geometric metrics in $\mathbb{R}^d$ or covariance structures.

Standardization (**Z**-Score)

- Transforms features to zero mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$):

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

- **Use Case**: Essential for PCA, K-Means, Gaussian Mixture Models, and gradient-based learning.

Min-Max Normalization

- Scales feature range strictly into $[0, 1]$:

$$x'_{ij} = \frac{x_{ij} - x_{\min,j}}{x_{\max,j} - x_{\min,j}}$$

- **Use Case**: Preferred when distance bounds are fixed or preserving zero values in sparse matrices is required.

Sensitivity to Outliers

Unsupervised algorithms lack ground-truth supervision to mask or ignore extreme values. Outliers distort mean vectors $\boldsymbol{\mu}_k$, shrink main clusters, and skew principal eigenvectors.

Geometric Metrics in Feature Space $\mathbb{R}^d$

The choice of distance metric defines the topology of the feature space and governs how similarity between unannotated data points is evaluated.

Minkowski Metric (L_p-norm) Generalized metric for feature vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$:

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p}$$

- **Euclidean ($p = 2$)**: Isotropic distance metric; assumes spherical clusters.
- **Manhattan ($p = 1$)**: Grid-based distance; robust to moderate high-dimensional noise.

Cosine Similarity & Distance Evaluates angular orientation rather than spatial magnitude:

$$S_{\cos}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}, \quad d_{\cos}(\mathbf{x}, \mathbf{y}) = 1 - S_{\cos}(\mathbf{x}, \mathbf{y})$$

Used extensively in text mining and sparse document embeddings.

Mahalanobis Distance Scale-invariant metric accounting for feature correlation:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^T \mathbf{\Sigma}^{-1} (\mathbf{x} - \mathbf{y})}$$

Where $\mathbf{\Sigma}$ is the sample covariance matrix.

Mathematical Formulation of Unsupervised Objectives I

Optimization Objectives Across Key Paradigms

Unsupervised algorithms operate by minimizing an objective function $\mathcal{J}$ over parameters $\boldsymbol{\Theta}$ given data $\mathbf{X}$.

1. Clustering: Within-Cluster Sum of Squares (WCSS / Inertia)

Given K cluster centers $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$ and cluster assignments C:

$$\mathcal{J}_{\text{WCSS}}(C, \boldsymbol{\mu}) = \sum_{k=1}^{K} \sum_{i \in C_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2^2 = \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2^2$$

where $r_{ik} \in \{0, 1\}$ is a binary indicator of point i belonging to cluster k.

Mathematical Formulation of Unsupervised Objectives II

2. Dimensionality Reduction: Reconstruction Loss Minimization

Finding an orthogonal projection matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$ $(k < d)$:

$$\mathcal{J}_{\text{PCA}}(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{W}\mathbf{W}^T \mathbf{x}_i\|_2^2 = \text{Tr}(\mathbf{\Sigma_X}) - \text{Tr}(\mathbf{W}^T \mathbf{\Sigma_X} \mathbf{W})$$

Minimizing reconstruction loss is mathematically equivalent to maximizing projected variance $\text{Tr}(\mathbf{W}^T \mathbf{\Sigma_X} \mathbf{W})$.

The Expectation-Maximization (EM) Framework I

General Probabilistic Latent Variable Optimization

When data generation depends on unobserved latent variables Z, direct maximum marginal likelihood estimation $\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \boldsymbol{\theta})$ is analytically intractable.

Expectation Step (E-Step)

- Compute posterior probabilities over latent variable Z given current parameters $\boldsymbol{\theta}^{(t)}$:

$$q_i(z) = P(Z_i = z \mid \mathbf{x}_i; \boldsymbol{\theta}^{(t)})$$

- Construct the expected complete-data log-likelihood (lower bound Q):

$$Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}) = \sum_{i=1}^{N} \mathbb{E}_{q_i} \left[\log p(\mathbf{x}_i, Z_i \mid \boldsymbol{\theta})\right]$$

Maximization Step (M-Step)

- Update parameter vector $\boldsymbol{\theta}$ by maximizing the Q-function:

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)})$$

- Re-estimate cluster parameters (means $\boldsymbol{\mu}_k$, covariances $\boldsymbol{\Sigma}_k$, and mixture weights π_k).

The Expectation-Maximization (EM) Framework II

Monotonic Convergence Property

Every iteration of the EM algorithm guarantees $\mathcal{L}(\boldsymbol{\theta}^{(t+1)}) \geq \mathcal{L}(\boldsymbol{\theta}^{(t)})$, converging to a local log-likelihood maximum.

Step-by-Step Mechanics: Centroid & Density Updating I

Iterative Partitioning Dynamics: Hard vs. Soft Assignment

Unsupervised iterative optimization proceeds by updating assignment scores and cluster parameters in alternating turns.

Hard Assignment (K-Means)

- **Assign**: Each point belongs to exactly one cluster center:

$$c^{(i)} = \arg \min_{k \in \{1,\dots,K\}} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2^2$$

- **Update**: Centroid vector re-estimation:

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} \mathbb{I}(c^{(i)} = k)\mathbf{x}_i}{\sum_{i=1}^{N} \mathbb{I}(c^{(i)} = k)}$$

Soft Assignment (GMM - EM)

- **Assign**: Compute responsibility $\gamma_{ik} \in [0, 1]$:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

- **Update**: Soft mean update using $N_k = \sum_{i=1}^{N} \gamma_{ik}$:

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i$$

Working of Dimensionality Reduction: SVD & Eigen-Decomposition I

Spectral Mechanics of Linear Representation Learning

Dimensionality reduction transforms high-dimensional features into a lower-dimensional orthogonal subspace that preserves maximal variance.

Covariance Eigen-Decomposition

- Compute mean-centered matrix $\mathbf{X}_c = \mathbf{X} - \bar{\mathbf{x}}$.
- Form covariance matrix:

$$\mathbf{C} = \frac{1}{N-1}\mathbf{X}_c^T\mathbf{X}_c \in \mathbb{R}^{d\times d}$$

- Solve eigenvalue problem $\mathbf{C}\mathbf{v}_j = \lambda_j\mathbf{v}_j$.
- Sort eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ to form projection matrix $\mathbf{W}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$.

Singular Value Decomposition (SVD)

- Directly factorize mean-centered matrix $\mathbf{X}_c$:

$$\mathbf{X}_c = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

- $\mathbf{U} \in \mathbb{R}^{N\times N}$: Left singular vectors (principal components scaled).
- $\mathbf{\Sigma} \in \mathbb{R}^{N\times d}$: Singular values $\sigma_j = \sqrt{(N-1)\lambda_j}$.
- Transformed data: $\mathbf{Z}_k = \mathbf{X}_c\mathbf{V}_k = \mathbf{U}_k\mathbf{\Sigma}_k \in \mathbb{R}^{N\times k}$.

Python Implementation: Complete Unsupervised Pipeline I

```python
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, davies_bouldin_score

# 1. Generate synthetic high-dimensional unlabeled data
X_raw, _ = make_blobs(n_samples=500, n_features=10, centers=4, cluster_std=1.5, random_state=42)

# 2. Step 1: Preprocessing & Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_raw)

# 3. Step 2: Dimensionality Reduction via PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
print(f"Explained Variance Ratio: {pca.explained_variance_ratio_}")

# 4. Step 3: Clustering via K-Means Optimization
kmeans = KMeans(n_clusters=4, init='k-means++', n_init=10, random_state=42)
cluster_labels = kmeans.fit_predict(X_pca)

# 5. Step 4: Internal Model Evaluation
sil_score = silhouette_score(X_pca, cluster_labels)
db_score = davies_bouldin_score(X_pca, cluster_labels)

print(f"Silhouette Score: {sil_score:.3f} | Davies-Bouldin Index: {db_score:.3f}")
```

Python Implementation: Complete Unsupervised Pipeline II

Python Implementation: Custom K-Means Iteration Mechanics I

```python
import numpy as np

def custom_kmeans(X, k=3, max_iters=100, tol=1e-4):
    n_samples, n_features = X.shape
    # Step 1: Initialize centroids randomly from data points
    idx = np.random.choice(n_samples, k, replace=False)
    centroids = X[idx]

    for iteration in range(max_iters):
        # Step 2: Calculate Euclidean distances (N, K)
        distances = np.linalg.norm(X[:, np.newaxis, :] - centroids[np.newaxis, :, :], axis=2)

        # Step 3: Hard assignment step (E-step equivalent)
        labels = np.argmin(distances, axis=1)

        # Step 4: Recalculate centroids (M-step equivalent)
        new_centroids = np.array([X[labels == j].mean(axis=0) for j in range(k)])

        # Step 5: Check for convergence
        centroid_shift = np.linalg.norm(new_centroids - centroids)
        centroids = new_centroids
        if centroid_shift < tol:
            print(f"Converged at iteration {iteration + 1}")
            break

    return centroids, labels
```

Evaluating Unsupervised Learning Models (Without Ground Truth) I

Internal Validation Criteria

Because true labels y are unavailable, unsupervised model performance is evaluated using geometric compactness (intra-cluster variance) and separability (inter-cluster distance).

Silhouette Coefficient

- For sample i, let $a(i)$ be mean intra-cluster distance and $b(i)$ be mean nearest-cluster distance:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \in [-1, 1]$$

- $S = 1$: Ideal separation.
- $S = 0$: Overlapping clusters.
- $S < 0$: Incorrect assignment.

Davies-Bouldin & Calinski-Harabasz

- **Davies-Bouldin Index ($\mathcal{DB}$)**: Measures average similarity ratio between each cluster and its most similar one:

$$\mathcal{DB} = \frac{1}{K} \sum_{i=1}^{K} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)} \right)$$

  Lower scores indicate superior separation.
- **Calinski-Harabasz**: Ratio of between-cluster to within-cluster dispersion (higher is better).

Challenges, Pitfalls & Operational Best Practices I

Core Practical Challenges in Unsupervised Learning

Unsupervised workflows present unique operational hurdles due to the lack of explicit target supervision.

Curse of Dimensionality As feature dimensionality d increases, volume grows exponentially, making data sparse. Pairwise Euclidean distances converge to equal values ($d_{\max}/d_{\min} \to 1$), deteriorating clustering quality.

- *Best Practice*: Always apply dimensionality reduction (PCA, UMAP) or feature selection before clustering high-dimensional spaces.

Local Minima & Sensitivity to Initialization Objectives like WCSS in K-Means or non-convex log-likelihoods in EM are non-convex and sensitive to initial seeds.

- *Best Practice*: Use probabilistic initialization (e.g., K-Means++) and run multiple random restarts.

Determining Hyperparameters (K or Latent k) Deciding optimal cluster count K without ground-truth.

- *Best Practice*: Utilize Elbow curves (WCSS knee), Scree plots, and Silhouette profile analysis.

Overview: Unsupervised Learning Taxonomy I

Defining Unsupervised Machine Learning

Unlike supervised learning, unsupervised learning operates on unlabeled datasets $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ where $x_i \in \mathbb{R}^d$. The primary goal is to uncover underlying structures, patterns, latent variables, or joint probability distributions $p(x)$ without explicit target labels.

1. Clustering

- **Objective**: Partition data into homogeneous subgroups (clusters).
- **Core Principle**: Maximize intra-cluster similarity, minimize inter-cluster similarity.
- **Paradigms**: Partitioning (K-Means), Hierarchical (Agglomerative), Density-Based (DBSCAN).

2. Association Rule Mining

- **Objective**: Discover conditional dependencies ($X \implies Y$) in itemsets.
- **Core Principle**: Extract rules satisfying support and confidence thresholds.
- **Paradigms**: Level-wise Search (Apriori), Tree-based Mining (FP-Growth).

Clustering: Problem Formulation & Distance Metrics I

Mathematical Formulation of Clustering

Given dataset $\mathcal{D} = \{x_1, \ldots, x_N\}$, partition $\mathcal{D}$ into K disjoint clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$ such that:

$$\bigcup_{k=1}^{K} C_k = \mathcal{D} \quad \text{and} \quad C_i \cap C_j = \emptyset, \quad \forall i \neq j$$

Minkowski Distance Metric

General metric for $x, y \in \mathbb{R}^d$:

$$d_p(x, y) = \left(\sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p}$$

- $p = 1$: Manhattan Distance (L_1 norm).
- $p = 2$: Euclidean Distance (L_2 norm).

Clustering: Problem Formulation & Distance Metrics II

Specialized Distance Metrics

- **Cosine Similarity**:

$$\text{Sim}_{\cos}(x, y) = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

- **Mahalanobis Distance**:

$$d_{\text{Mah}}(x, y) = \sqrt{(x - y)^T \Sigma^{-1} (x - y)}$$

Accounts for correlation across features (Σ).

Partitioning Clustering: K-Means Algorithm I

Objective Function: Within-Cluster Sum of Squares (WCSS)

K-Means seeks cluster assignments $\mathcal{C}$ and centroids $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_K\}$ that minimize the Total Inertia:

$$J(\mathcal{C}, \boldsymbol{\mu}) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|_2^2$$

where $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$ is the mean (centroid) of cluster C_k.

Partitioning Clustering: K-Means Algorithm II

Lloyd's Iterative Optimization Algorithm

1. **Initialization**: Select K initial cluster centroids $\mu_1^{(0)}, \ldots, \mu_K^{(0)}$.
2. **Assignment Step**: Assign each data point to its nearest centroid:

$$C_k^{(t)} = \left\{ x_i : \|x_i - \mu_k^{(t)}\|_2^2 \leq \|x_i - \mu_j^{(t)}\|_2^2, \ \forall j \neq k \right\}$$

3. **Update Step**: Recompute centroids based on newly assigned points:

$$\mu_k^{(t+1)} = \frac{1}{|C_k^{(t)}|} \sum_{x_i \in C_k^{(t)}} x_i$$

4. **Convergence**: Repeat steps 2–3 until centroids stabilize ($\mu_k^{(t+1)} = \mu_k^{(t)}$).

K-Means++ Initialization & Methodological Limitations I

K-Means++ Initialization

Mitigates poor local minima by spreading initial centroids:

1. Choose first centroid μ_1 uniformly at random from dataset $\mathcal{D}$.
2. For each $x \in \mathcal{D}$, compute shortest distance to existing centroids: $D(x) = \min_k \|x - \mu_k\|_2$.
3. Select next centroid μ_j with probability proportional to $D(x)^2$:

$$P(x) = \frac{D(x)^2}{\sum_{x' \in \mathcal{D}} D(x')^2}$$

4. Repeat until K centroids are chosen. Guarantees an $\mathcal{O}(\log K)$ approximation bound.

Limitations of K-Means

- **Spherical Shape Assumption**: Struggles with non-globular or complex non-convex cluster topologies.
- **Pre-specified K**: Requires user to define cluster count K a priori.
- **Sensitivity to Outliers**: Centroid mean computation is susceptible to extreme values.
- **Scale Sensitivity**: Requires strict feature standardization (z-score normalization).

Hierarchical Clustering: Agglomerative vs. Divisive I

Hierarchical Taxonomy

Hierarchical clustering constructs a nested tree of clusters called a **Dendrogram**.

- **Agglomerative (Bottom-Up)**: Starts with N singletons; iteratively merges nearest pair.
- **Divisive (Top-Down)**: Starts with one root cluster containing all points; recursively splits.

Cluster Linkage Criteria $d(A, B)$

- **Single Linkage** (Minimum distance):

$$d_{\text{min}}(A, B) = \min_{x \in A, y \in B} d(x, y)$$

- **Complete Linkage** (Maximum distance):

$$d_{\text{max}}(A, B) = \max_{x \in A, y \in B} d(x, y)$$

- **Average Linkage** (Mean pairwise distance):

$$d_{\text{avg}}(A, B) = \frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} d(x, y)$$

Hierarchical Clustering: Agglomerative vs. Divisive II

Ward's Minimum Variance

Minimizes total within-cluster variance growth when merging clusters A and B:

$$\Delta \text{ESS}(A, B) = \frac{|A||B|}{|A| + |B|} \|\mu_A - \mu_B\|_2^2$$

- Tends to create balanced, spherical clusters.
- Most widely used linkage in practical data science pipelines.

Density-Based Clustering: DBSCAN I

Density-Based Spatial Clustering of Applications with Noise

DBSCAN identifies clusters as dense continuous regions separated by regions of low density. It effectively discovers arbitrarily shaped clusters and identifies noise points.

Core Definitions (ε, MinPts)

- ε-**Neighborhood**: $N_\varepsilon(x) = \{y \in \mathcal{D} : d(x, y) \leq \varepsilon\}$.
- **Core Point**: Point x with $|N_\varepsilon(x)| \geq$ MinPts.
- **Border Point**: Point $y \notin$ Core, but $y \in N_\varepsilon(p)$ for some core point p.
- **Noise Point**: Any point that is neither a core point nor a border point.

Density-Reachability

- **Directly Density-Reachable**: $y \in N_\varepsilon(x)$ where x is a Core Point.
- **Density-Reachable**: Chain of core points connecting x to y.
- **Density-Connected**: Points x and y are both density-reachable from a common core point o.

Validation Metrics for Clustering I

Internal Validation (No Ground Truth)

- **Silhouette Coefficient**:

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

where $a(i)$ is mean intra-cluster distance, $b(i)$ is mean distance to nearest cluster. $s(i) \in [-1, +1]$.

- **Davies-Bouldin Index**:

$$DB = \frac{1}{K} \sum_{k=1}^{K} \max_{j \neq k} \left(\frac{\sigma_k + \sigma_j}{d(\mu_k, \mu_j)} \right)$$

Lower values indicate better clustering compactness and separation.

Validation Metrics for Clustering II

External Validation (With Ground Truth)

- **Adjusted Rand Index (ARI)**: Measures agreement between true class labels Y and cluster assignments C, adjusted for random chance:

$$\text{ARI} = \frac{\text{RI} - \mathbb{E}[\text{RI}]}{\max(\text{RI}) - \mathbb{E}[\text{RI}]}$$

- **Normalized Mutual Information (NMI)**:

$$\text{NMI}(Y, C) = \frac{2I(Y; C)}{H(Y) + H(C)}$$

Ranges from 0 (uncorrelated) to 1 (perfect recovery).

Association Rule Mining: Problem Formulation I

Transactional Database Framework

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a universe of items. Let $T = \{t_1, t_2, \ldots, t_N\}$ be a database of transactions, where each transaction $t_k \subseteq I$. An **Association Rule** is an implication of the form:

$$X \implies Y \quad \text{where } X, Y \subset I \text{ and } X \cap Y = \emptyset$$

X is the *Antecedent* (Left-Hand Side), Y is the *Consequent* (Right-Hand Side).

1. Support

Frequency of itemset in dataset:

$$\text{Supp}(X) = \frac{|\{t_k \in T : X \subseteq t_k\}|}{|T|}$$

2. Confidence

Conditional probability $P(Y|X)$:

$$\text{Conf}(X \implies Y) = \frac{\text{Supp}(X \cup Y)}{\text{Supp}(X)}$$

Association Rule Mining: Problem Formulation II

3. Lift

Ratio of observed to expected co-occurrence:

$$\text{Lift}(X \implies Y) = \frac{\text{Supp}(X \cup Y)}{\text{Supp}(X) \cdot \text{Supp}(Y)}$$

The Apriori Algorithm & Monotonicity Principle I

The Apriori Property (Downward Closure / Anti-Monotonicity)

Theorem: If an itemset X is frequent (Supp(X) $\geq$ min_sup), then all of its non-empty subsets $S \subset X$ must also be frequent.
Pruning Rule: If an itemset S is infrequent, all of its supersets $X \supset S$ are guaranteed to be infrequent and can be pruned immediately without scanning the database!

Level-Wise Search Strategy

1. **Frequent 1-Itemsets (L_1)**: Scan database to identify all 1-itemsets satisfying min_sup.
2. **Candidate Generation (C_{k+1})**: Join L_k with L_k ($L_k \bowtie L_k$) to construct candidate $(k+1)$-itemsets.
3. **Candidate Pruning**: Remove any candidate $c \in C_{k+1}$ if any k-subset of c is not in L_k.
4. **Database Scan**: Count actual support of remaining candidates in C_{k+1} to form L_{k+1}.
5. **Termination**: Stop when no new frequent itemsets are generated ($L_{k+1} = \emptyset$).

FP-Growth: Candidate-Free Pattern Mining I

Motivation: Overcoming Apriori Bottlenecks

Apriori suffers from candidate generation explosion ($2^{|I|}$ search space) and requires multiple full database scans (k passes for length-k itemsets).

FP-Tree Data Structure

Encodes database into a compact tree using 2 passes:

- **Pass 1**: Calculate item support; discard infrequent items; sort items in descending support order.
- **Pass 2**: Read transactions sequentially and insert sorted items into the FP-Tree, sharing common prefixes.

Divide-and-Conquer Mining

- Builds a **Header Table** linking item nodes via pointers.
- Constructs **Conditional Pattern Bases** for each item starting from suffix nodes.
- Mines conditional FP-Trees recursively.
- Avoids candidate generation entirely!

Comparative Analysis: Clustering vs. Association Rules I

Methodological Comparison

Dimension	Clustering Analysis	Association Rule Mining
Input Format	Feature vectors $x \in \mathbb{R}^d$	Transactional item sets $t \subseteq I$
Primary Goal	Group similar observations	Discover co-occurrence rules
Key Metrics	Euclidean distance, WCSS, Silhouette	Support, Confidence, Lift
Primary Output	Partition assignments $(C_1, \ldots, C_K)$	Rules of form $X \implies Y$
Key Algorithms	K-Means, Hierarchical, DBSCAN	Apriori, FP-Growth, ECLAT
Search Space	Partitioning space K^N	Combinatorial itemset space $2^{\lvert I \rvert}$

Python Implementation: Clustering Techniques I

```python
import numpy as np
from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics import silhouette_score
from sklearn.datasets import make_blobs

# 1. Generate synthetic dataset
X, y_true = make_blobs(n_samples=500, centers=4,
                       cluster_std=0.60, random_state=42)

# 2. Partitioning Clustering: K-Means++
kmeans = KMeans(n_clusters=4, init='k-means++',
                n_init=10, random_state=42)
labels_km = kmeans.fit_predict(X)
score_km = silhouette_score(X, labels_km)

# 3. Density-Based Clustering: DBSCAN
dbscan = DBSCAN(eps=0.4, min_samples=5)
labels_db = dbscan.fit_predict(X)

print(f"K-Means Silhouette Score: {score_km:.4f}")
print(f"DBSCAN Detected Clusters: {len(set(labels_db)) - (1 if -1 in labels_db else 0)}")
print(f"DBSCAN Outlier Points: {list(labels_db).count(-1)}")
```

Python Implementation: Association Rule Mining I

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules

# 1. Sample Transactional Dataset
dataset = [['Milk', 'Onion', 'Nutmeg', 'Eggs', 'Yogurt'],
           ['Dill', 'Onion', 'Nutmeg', 'Eggs', 'Yogurt'],
           ['Milk', 'Apple', 'Eggs'],
           ['Milk', 'Corn', 'Yogurt'],
           ['Corn', 'Onion', 'Eggs', 'Ice-cream']]

# 2. One-hot Encode Transactions
te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)

# 3. Mine Frequent Itemsets (Min Support = 60%)
frequent_itemsets = apriori(df, min_support=0.6, use_colnames=True)

# 4. Generate Association Rules (Min Confidence = 70%)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)
print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

Summary & Key Takeaways I

Core Conceptual Takeaways

- **Unsupervised Learning Objectives**: Focuses on discovering intrinsic geometry (Clustering) or item correlation structure (Association Rules) without target supervision.
- **Clustering Selection Guide**:
  - Use K-**Means** for large-scale datasets with compact, convex clusters.
  - Use **Hierarchical** when tree structure/dendrogram is required.
  - Use **DBSCAN** when noise handling and arbitrary shape discovery are needed.
- **Association Rule Mining Strategy**: Utilize **Lift** > 1.0 to filter out uninformative rules resulting from high marginal probabilities. Prefer **FP-Growth** over Apriori for high transaction density.

Next Lecture Preview

Lecture 5.5: Dimensionality Reduction — Principal Component Analysis (PCA), Singular Value Decomposition (SVD), and Manifold Learning (t-SNE, UMAP).

Landscape of Unsupervised Learning in Industry I

The Industry Reality: Abundance of Unlabeled Data

Over 90% of real-world enterprise data (system logs, sensor telemetry, customer interactions, unstructured text) is unlabeled. Unsupervised learning extracts actionable insights without expensive manual annotation.

Core Application Domains

- **Customer Analytics**: RFM segmentation, behavioral profiling.
- **Cybersecurity & Finance**: Fraud detection, network intrusion alerts.
- **Bioinformatics**: Genomic subgrouping, disease discovery.

Strategic Value Drivers

- **Exploratory Data Analysis**: Discovering hidden data structures.
- **Feature Preprocessing**: Dimensionality reduction for downstream models.
- **Automated Alerting**: Detecting rare events and operational anomalies.

Domain 1: Customer Segmentation & Market Basket Analysis I

RFM Customer Segmentation

Clustering customers by **R**ecency, **F**requency, and **M**onetary value using K-Means or Gaussian Mixture Models (GMMs) to tailor marketing strategies.

Market Basket Analysis

Identifies product co-occurrence patterns in retail transactions using Association Rule Mining (Apriori / FP-Growth).

Association Metrics

For itemsets $A \implies B$:

- **Support**: $P(A \cap B) = \frac{\text{freq}(A,B)}{N}$
- **Confidence**: $P(B|A) = \frac{\text{Support}(A \cap B)}{\text{Support}(A)}$
- **Lift**: $\frac{P(A \cap B)}{P(A)P(B)} = \frac{\text{Confidence}(A \implies B)}{\text{Support}(B)}$

Domain 2: Anomaly & Fraud Detection Systems I

Unsupervised Outlier Detection Paradigm

In financial transactions and network traffic, anomalies are extremely rare and dynamic, making supervised classification prone to severe class imbalance and out-of-date rules.

Primary Algorithms

- **Isolation Forest**: Isolates anomalies by randomly partitioning feature space; outliers require fewer splits.
- **Local Outlier Factor (LOF)**: Measures local density deviation relative to k-nearest neighbors.
- **Autoencoders**: Reconstruction error $\|x - \hat{x}\|_2^2 > \tau$ signals abnormal behavior.

Industrial Applications

- Credit card fraud detection in real-time stream processing.
- Predictive maintenance in IoT sensor telemetry (e.g., turbine failure).
- Zero-day network intrusion detection.

Domain 3: High-Dimensional Visualization & Feature Extraction I

Curse of Dimensionality Mitigation

High-dimensional datasets (e.g., gene expression profiles with 20,000+ features or image embeddings) suffer from data sparsity and heavy computational overhead.

Linear vs. Non-linear Reduction

- **PCA (Principal Component Analysis)**: Orthogonal linear projection preserving global variance.
- **t-SNE / UMAP**: Non-linear manifold learning preserving local neighborhood distances.

Genomics & Image Compression

- Single-cell RNA sequencing (scRNA-seq) visualization via UMAP.
- Eigenfaces for facial recognition preprocessing via SVD:

$$X = U\Sigma V^T$$

Domain 4: Topic Modeling & Unstructured Text Clustering I

Latent Dirichlet Allocation (LDA)

LDA is a generative probabilistic model that represents documents as random mixtures over latent topics, where each topic is characterized by a distribution over words.

Generative Process

1. For each document d, sample topic proportions $\theta_d \sim \text{Dir}(\alpha)$.
2. For each word $w_{d,n}$, sample topic assignment $z_{d,n} \sim \text{Multinomial}(\theta_d)$.
3. Sample word $w_{d,n} \sim \text{Multinomial}(\beta_{z_{d,n}})$.

Modern Text Applications

- **Legal Discovery**: Clustering millions of contract clauses.
- **Customer Feedback**: Grouping support tickets automatically.
- **BERTopic**: Combining Transformer embeddings (SBERT) with HDBSCAN.

Domain 5: Image Segmentation & Computer Vision I

Pixel-Level Unsupervised Grouping

Partitioning images into semantically meaningful regions without pixel-level ground truth masks.

Techniques

- **Color Quantization**: K-Means clustering on RGB/Lab color channels to compress color palettes.
- **Normalized Cuts**: Graph-based spectral partitioning of pixel similarity matrices.
- **SLIC Superpixels**: Localized K-Means in 5D space (R, G, B, x, y).

Real-World Scenarios

- Medical MRI tumor boundary candidate generation.
- Satellite imagery land-cover classification (forest vs. urban).
- Image compression for embedded devices.

Python Implementation: Customer Segmentation & Outlier Detection I

End-to-End Pipeline in Python

Demonstrates feature scaling, dimensionality reduction, K-Means clustering, and Isolation Forest anomaly filtering.

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.ensemble import IsolationForest

# 1. Feature Engineering & Scaling
X_raw = np.random.rand(1000, 5) # Recency, Frequency, Monetary, etc.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_raw)

# 2. Dimensionality Reduction (PCA)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# 3. Customer Segmentation (K-Means)
kmeans = KMeans(n_clusters=4, random_state=42)
cluster_labels = kmeans.fit_predict(X_pca)

# 4. Anomaly / Fraud Detection (Isolation Forest)
iso_forest = IsolationForest(contamination=0.05, random_state=42)
anomalies = iso_forest.fit_predict(X_scaled) # -1: Outlier, 1: Normal
```

Python Implementation: Customer Segmentation & Outlier Detection II

Evaluation & Operational Challenges in Production I

The Evaluation Dilemma: No Ground Truth Labels

Without true labels y, model validation relies on internal cluster validation indices and domain-specific downstream metrics.

Validation Metrics

- **Silhouette Coefficient**:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \in [-1, 1]$$

- **Davies-Bouldin Index**: Ratio of intra-cluster scatter to inter-cluster separation (lower is better).
- **Calinski-Harabasz Index**: Ratio of between-cluster to within-cluster dispersion.

Production Challenges

- **Concept Drift**: Cluster centers shift over time as user behavior changes.
- **Scalability**: Distance matrix computation $O(N^2 d)$ requires mini-batch or approximate nearest neighbors (e.g., Faiss, Annoy).
- **Interpretability**: Translating high-dimensional latent clusters into business logic.

Summary: Real-World Unsupervised Learning Taxonomy I

Comparative Industry Map

Application	Core Algorithm	Key Metric	Industry Impact
Customer Clustering	K-Means / GMM	Silhouette Score	Personalized Marketing
Fraud Detection	Isolation Forest	Precision@K / Recall	Financial Loss Reduction
Genomic Analysis	PCA / UMAP	Variance Explained	Disease Subtype Discovery
Topic Extraction	LDA / BERTopic	Topic Coherence	Automated Document Routing
Color Quantization	MiniBatch K-Means	Mean Squared Error	Image Compression

Bridge to Generative AI

Unsupervised representation learning forms the foundation of modern Generative AI: Autoencoders (VAEs) and Self-Supervised Learning (Contrastive Learning, Masked Autoencoders) map unstructured data into continuous latent spaces.

Learning Paradigms: Overview I

The Fundamental Divide in Machine Learning

Machine Learning paradigms are primarily categorized by the nature of the training data and the presence or absence of explicit target supervisory signals.

Supervised Learning

- **Dataset**: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ containing feature-label pairs.
- **Goal**: Learn functional mapping $f : \mathcal{X} \to \mathcal{Y}$ to predict target y from features $\mathbf{x}$.
- **Feedback**: Direct error signal computed against ground truth targets y_i.

Unsupervised Learning

- **Dataset**: $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N}$ containing unannotated feature vectors.
- **Goal**: Uncover intrinsic data structure, latent patterns, or underlying distribution $p(\mathbf{x})$.
- **Feedback**: Indirect signal (e.g., reconstruction error, density maximization).

Core Distinction

Supervised learning answers *"What is the correct output for this input?"*, whereas unsupervised learning answers *"What is the underlying structure of this data?"*.

Mathematical & Probabilistic Formulations I

Supervised Formulation: Empirical Risk Minimization (ERM)

Given N samples, supervised learning estimates parameters $\boldsymbol{\theta}$ by minimizing empirical risk over conditional distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i; \boldsymbol{\theta})) \quad \Longleftrightarrow \quad \boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta})$$

Unsupervised Formulation: Density Estimation & Latent Modeling

Unsupervised learning models marginal likelihood $p(\mathbf{x}; \boldsymbol{\theta})$, often introducing latent variables $\mathbf{z} \in \mathbb{R}^k$:

$$p(\mathbf{x}; \boldsymbol{\theta}) = \int_{\mathcal{Z}} p(\mathbf{x} \mid \mathbf{z}; \boldsymbol{\theta}) p(\mathbf{z}) d\mathbf{z} \quad \Longrightarrow \quad \boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(\mathbf{x}_i; \boldsymbol{\theta})$$

Reconstruction Formalism (Autoencoders / PCA)

Learn encoder $f_{\boldsymbol{\phi}} : \mathcal{X} \to \mathcal{Z}$ and decoder $g_{\boldsymbol{\theta}} : \mathcal{Z} \to \mathcal{X}$:

$$\min_{\boldsymbol{\theta},\boldsymbol{\phi}} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{x}_i - g_{\boldsymbol{\theta}} \left(f_{\boldsymbol{\phi}}(\mathbf{x}_i) \right) \right\|_2^2$$

Taxonomy of Tasks I

Supervised Tasks

- **Classification**: Categorical target $\mathcal{Y} = \{1, \ldots, K\}$.
  - Logistic Regression, SVM, Random Forest, Neural Networks.
- **Regression**: Continuous target $\mathcal{Y} = \mathbb{R}^m$.
  - Ridge, Lasso, Gradient Boosting Regressors.

Unsupervised Tasks

- **Clustering**: Partitioning $\mathcal{X}$ into K discrete subsets.
  - K-Means, Hierarchical, DBSCAN, GMM.
- **Dimensionality Reduction**: Projection to $\mathbb{R}^k$ ($k \ll d$).
  - PCA, t-SNE, UMAP, Autoencoders.
- **Generative Modeling**: Learning $p(\mathbf{x})$.
  - VAEs, GANs, Diffusion Models.

Task Divergence

Supervised models map fixed inputs to pre-defined outputs; unsupervised models explore unlabelled spaces to infer representations, structures, or generation mechanisms.

Systematic Paradigm Comparison I

Feature Comparison Matrix

Dimension	Supervised Learning	Unsupervised Learning
Input Data	Labeled pairs $(\mathbf{x}_i, y_i)$	Unlabeled features $\mathbf{x}_i$
Primary Goal	Predict target y for unseen $\mathbf{x}$	Discover structural patterns in $\mathcal{X}$
Target Labels	Required ($y_i \in \mathcal{Y}$)	Absent ($\emptyset$)
Evaluation	Objective metrics (Accuracy, MSE)	Heuristic / Internal metrics (Silhouette)
Data Cost	High (Manual annotation needed)	Low (Raw data collected abundantly)
Algorithmic Complexity	Driven by loss convergence	Driven by space/density exploration

Evaluation Paradigms & Validation Metrics I

Supervised Validation

Ground truth labels enable direct loss and metric computation:

- **Classification**: Accuracy, Precision, Recall, F_1-score, ROC-AUC.
- **Regression**: Mean Squared Error (MSE), R^2 coefficient of determination.

Unsupervised Validation (No Ground Truth)

Performance must be assessed via data-intrinsic structural properties:

- **Silhouette Coefficient**: Evaluates cluster cohesion $a(i)$ vs. separation $b(i)$:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \in [-1, 1]$$

- **Reconstruction Error**: Mean distance between original and reconstructed vectors.
- **External Benchmarking**: Adjusted Rand Index (ARI), Normalized Mutual Information (NMI) (when synthetic ground truth exists).

Bridging the Gap: Semi-Supervised & Self-Supervised Learning I

Semi-Supervised Learning (SSL)

Leverages small labeled dataset $\mathcal{D}_L = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_L}$ alongside large unlabeled dataset $\mathcal{D}_U = \{\mathbf{x}_j\}_{j=1}^{N_U}$ ($N_U \gg N_L$).

- **Combined Objective**: $\mathcal{L} = \mathcal{L}_{\text{supervised}}(\mathcal{D}_L) + \lambda \mathcal{L}_{\text{unsupervised}}(\mathcal{D}_U)$.
- **Techniques**: Pseudo-labeling, consistency regularization, graph-based label propagation.

Self-Supervised Learning (SSL)

Transforms unsupervised data into supervised learning tasks by creating **pretext tasks** directly from raw data without human labeling.

- **Masked Pretext Tasks**: Predict masked words (BERT) or masked image patches (MAE).
- **Contrastive Learning**: Pull augmented views of same image together, push different images apart (SimCLR, CLIP).

Modern Workflow Standard

Self-supervised pre-training on massive unlabeled data followed by supervised fine-tuning on small target datasets.

Python Implementation: Workflow Comparison I

Comparing Supervised (Random Forest) vs. Unsupervised (K-Means)

```python
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, silhouette_score

# Generate synthetic dataset: X (features), y (ground truth labels)
X, y = make_blobs(n_samples=500, centers=3, n_features=4, random_state=42)

# --- 1. SUPERVISED LEARNING WORKFLOW ---
# Uses BOTH features X and labels y
rf_model = RandomForestClassifier(n_estimators=50, random_state=42)
rf_model.fit(X[:400], y[:400])                    # Training with targets y
y_pred = rf_model.predict(X[400:])
acc = accuracy_score(y[400:], y_pred)             # Exact metric against y
print(f"Supervised Test Accuracy: {acc:.4f}")

# --- 2. UNSUPERVISED LEARNING WORKFLOW ---
# Uses ONLY features X (no ground truth labels y!)
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(X)            # Discovery of structure
sil_score = silhouette_score(X, cluster_labels) # Internal validation
print(f"Unsupervised Silhouette Score: {sil_score:.4f}")
```

Python Implementation: Workflow Comparison II

Real-World Applications & Practical Decision Guide I

Clinical Healthcare Application

- **Supervised**: Train classifier on labeled MRI scans (Tumor vs. Benign) to assist automated diagnostic triage.
- **Unsupervised**: Cluster multi-omic genomic sequences to discover previously unknown disease sub-phenotypes.

E-Commerce & Retail Application

- **Supervised**: Predict customer churn probability $y \in [0, 1]$ based on historical transaction features.
- **Unsupervised**: Segment customer base into behavioral personas using purchasing pattern clustering.

Decision Tree for Method Selection

1. **High-quality labeled target available?** $\rightarrow$ Choose **Supervised Learning**.
2. **No target labels available, goal is exploration / grouping?** $\rightarrow$ Choose **Unsupervised Learning**.
3. **Abundant raw data, scarce labels?** $\rightarrow$ Choose **Self-Supervised Pre-training** + **Supervised Fine-tuning**.

Discriminative vs. Generative Modeling I

Discriminative Models

- Estimate conditional probability distribution $P(Y \mid X)$ or decision boundary $f(X) \rightarrow Y$.
- **Goal:** Distinguish between categories given input features (Classification / Regression).
- **Examples:** Logistic Regression, Support Vector Machines, Random Forests, Standard CNNs.

Generative Models

- Estimate joint distribution $P(X, Y)$ or marginal data distribution $P(X)$.
- **Goal:** Learn underlying probability structure to generate new synthetic samples $\hat{x} \sim P(X)$.
- **Examples:** Naive Bayes, GMMs, Variational Autoencoders (VAEs), GANs, Diffusion Models.

Taxonomy of Generative Modeling I

Problem Formulation

Given dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ sampled independently from unknown data distribution $p_{\text{data}}(x)$, fit a parametric model $p_\theta(x)$.

Density Estimation Paradigms

- **Explicit Density Estimation:** Define parametric model $p_\theta(x)$ and directly maximize log-likelihood $\sum_{i=1}^{N} \log p_\theta(x_i)$.
  - *Tractable Density:* Normalizing Flows, Autoregressive Models (e.g., PixelCNN, GPT).
  - *Approximate Density:* Variational Autoencoders (VAEs, maximizing ELBO).
- **Implicit Density Estimation:** Train generator $G_\theta(z)$ mapping noise $z \sim p(z)$ to realistic samples without explicit calculation of $p_\theta(x)$ (e.g., GANs).

Variational Autoencoders (VAEs): Architecture I

Latent Variable Model

Assume observed data x is generated by unobserved continuous latent variable $z \sim p(z) = \mathcal{N}(0, I)$ via conditional likelihood $p_\theta(x \mid z)$.

Intractability of Direct Maximum Likelihood

Marginalizing over latent space $p_\theta(x) = \int p_\theta(x \mid z)p(z)\,dz$ and computing exact posterior $p_\theta(z \mid x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)}$ are computationally intractable.

Encoder-Decoder Structure

- **Probabilistic Encoder** $q_\phi(z \mid x)$: Approximates intractable posterior $p_\theta(z \mid x)$, mapping input x to parameters $(\mu_\phi(x), \sigma^2_\phi(x))$.
- **Probabilistic Decoder** $p_\theta(x \mid z)$: Reconstructs data x given latent sample z.

Evidence Lower Bound (ELBO) & Reparameterization I

ELBO Objective

Using Jensen's Inequality / KL divergence, log marginal likelihood is bounded below:

$$\log p_\theta(x) \geq \mathcal{L}_{\text{ELBO}}(\theta, \phi; x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)]}_{\text{Reconstruction Loss}} - \underbrace{D_{\text{KL}}(q_\phi(z \mid x) \parallel p(z))}_{\text{Latent Regularization (KL Divergence)}}$$

The Reparameterization Trick

Sampling $z \sim \mathcal{N}(\mu_\phi, \Sigma_\phi)$ breaks backpropagation (non-differentiable).

- Isolate stochasticity by sampling standard normal noise $\epsilon \sim \mathcal{N}(0, I)$.
- Compute continuous mapping:

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$

- Allows gradient computation via chain rule: $\frac{\partial \mathcal{L}}{\partial \phi} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial \phi}$.

Generative Adversarial Networks (GANs) I

Minimax Two-Player Game

Simultaneously train two competing neural networks:

- **Generator** $G_\theta(z)$: Maps random noise $z \sim p_z(z)$ to synthetic sample $G_\theta(z)$.
- **Discriminator** $D_\phi(x)$: Classifies whether input x comes from real data distribution $p_{\text{data}}(x)$ ($D \approx 1$) or generated sample ($D \approx 0$).

Minimax Objective Function

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D_\phi(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D_\phi(G_\theta(z)))]$$

- Optimal Discriminator: $D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$.
- At global equilibrium ($p_g = p_{\text{data}}$): $D^*(x) = \frac{1}{2}$ and $V(D^*, G) = -\log 4$.

GAN Training Challenges & Extensions I

Training Dynamics & Failure Modes

- **Mode Collapse:** Generator outputs low variety (collapses to generating single convincing sample).
- **Vanishing Gradients:** Early in training, D easily rejects generated samples ($D(G(z)) \approx 0$), leading to zero gradient for G.
- **Non-Saturating Loss Fix:** Train G to maximize $\mathbb{E}_z[\log D(G(z))]$ instead of minimizing $\mathbb{E}_z[\log(1 - D(G(z)))]$.

Wasserstein GAN (WGAN)

Replaces Jensen-Shannon Divergence with Earth Mover's (Wasserstein-1) Distance:

$$\min_G \max_{D \in \mathcal{D}_L} \mathbb{E}_{x \sim p_{\text{data}}}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))]$$

Requires 1-Lipschitz continuity enforced via weight clipping or gradient penalty (WGAN-GP).

Diffusion & Autoregressive Models I

Denoising Diffusion Probabilistic Models (DDPM)

- **Forward Process (Noising):** Slowly add Gaussian noise to sample $x_0 \to x_1 \to \cdots \to x_T$.
- **Reverse Process (Denoising):** Train network $\epsilon_\theta(x_t, t)$ to predict noise added at step t:

$$\mathcal{L}_{\text{simple}}(\theta) = \mathbb{E}_{t, x_0, \epsilon}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2\right]$$

- Powers state-of-the-art image generators (Stable Diffusion, Midjourney).

Autoregressive Models (Transformers)

- Factorize joint sequence distribution using chain rule:

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1})$$

- Dominant paradigm in Large Language Models (LLMs like GPT-4, LLaMA).

Python Implementation: PyTorch VAE Module I

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class VAE(nn.Module):
    def __init__(self, input_dim=784, latent_dim=20):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 400)
        self.fc_mu = nn.Linear(400, latent_dim)
        self.fc_logvar = nn.Linear(400, latent_dim)
        self.fc3 = nn.Linear(latent_dim, 400)
        self.fc4 = nn.Linear(400, input_dim)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        h = F.relu(self.fc1(x))
        mu, logvar = self.fc_mu(h), self.fc_logvar(h)
        z = self.reparameterize(mu, logvar)
        recon_x = torch.sigmoid(self.fc4(F.relu(self.fc3(z))))
        return recon_x, mu, logvar

def vae_loss(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
```

Python Implementation: PyTorch VAE Module II

```
return BCE + KLD
```

Python Implementation: PyTorch GAN Step I

```python
def train_gan_step(G, D, opt_G, opt_D, real_imgs, latent_dim=100):
    batch_size = real_imgs.size(0)
    device = real_imgs.device
    criterion = torch.nn.BCELoss()

    real_labels = torch.ones(batch_size, 1, device=device)
    fake_labels = torch.zeros(batch_size, 1, device=device)

    # 1. Train Discriminator: max log(D(x)) + log(1 - D(G(z)))
    opt_D.zero_grad()
    z = torch.randn(batch_size, latent_dim, device=device)
    fake_imgs = G(z)

    loss_D_real = criterion(D(real_imgs), real_labels)
    loss_D_fake = criterion(D(fake_imgs.detach()), fake_labels)
    loss_D = loss_D_real + loss_D_fake
    loss_D.backward()
    opt_D.step()

    # 2. Train Generator: max log(D(G(z)))
    opt_G.zero_grad()
    loss_G = criterion(D(fake_imgs), real_labels)
    loss_G.backward()
    opt_G.step()

    return loss_D.item(), loss_G.item()
```

Comparison of Generative AI Paradigms I

Model Class	Sampling Speed	Sample Quality	Likelihood Access
VAEs	Fast	Moderate (Blurry)	Approximate (ELBO)
GANs	Fast	High (Sharp)	Implicit (None)
Diffusion	Slow (Multi-step)	State-of-the-Art	Exact / Lower Bound
Autoregressive	Sequential	High	Exact Likelihood

Summary Takeaways

- Generative models learn $P(X)$ or $P(X, Y)$ to synthesize novel data samples.
- VAEs leverage variational inference and reparameterization trick for stable latent representations.
- GANs use zero-sum game dynamics for high-frequency perceptual fidelity.
- Diffusion & Autoregressive Transformers currently dominate multimodal AI synthesis.

Lecture Overview & Learning Objectives I

- **Course**: DI04000061 (Introduction Machine Learning)
- **Unit 5**: Unsupervised Machine Learning and Generative AI
- **Topic**: 5.2.1 Define Generative AI

Learning Objectives:

1. Define **Generative AI** and contrast it with traditional Discriminative AI models.
2. Formalize the mathematical objectives of generative modeling ($P(X)$ vs $P(Y \mid X)$).
3. Classify generative approaches into **Explicit** vs. **Implicit** density estimation.
4. Survey foundational model paradigms: VAEs, GANs, Diffusion Models, and Autoregressive LLMs.
5. Understand evaluation metrics (e.g., FID, Perplexity) and challenges like Mode Collapse.
6. Implement a fundamental generative density sampling pipeline in Python.

What is Generative AI? I

Formal Definition

Generative AI refers to a class of machine learning models that learn the underlying joint probability distribution $P(X, Y)$ or marginal distribution $P(X)$ of a dataset in order to generate new, synthetic data instances $\hat{\boldsymbol{x}} \sim P_{\text{model}}(X)$ that resemble real data $\boldsymbol{x} \sim P_{\text{data}}(X)$.

Discriminative AI

- Learns conditional probability $P(Y \mid X)$.
- Maps inputs to labels (decision boundary).
- **Task**: Classification, Detection, Regression.
- **Question**: *"Is this image a cat or a dog?"*

Generative AI

- Learns distribution $P(X)$ or $P(X, Y)$.
- Models data density to sample new data.
- **Task**: Synthesis, Inpainting, Translation.
- **Question**: *"Generate a new realistic image of a cat."*

Mathematical Foundations: Discriminative vs. Generative I

Probabilistic Formulations

Given input data $\boldsymbol{x} \in \mathcal{X}$ and target labels $y \in \mathcal{Y}$:

- **Discriminative Objective**: Maximizes conditional log-likelihood:

$$\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log P_{\boldsymbol{\theta}}(y_i \mid \boldsymbol{x}_i)$$

- **Generative Objective (Joint Likelihood)**: Maximizes joint likelihood:

$$\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log P_{\boldsymbol{\theta}}(\boldsymbol{x}_i, y_i) = \max_{\boldsymbol{\theta}} \sum_{i=1}^{N} [\log P_{\boldsymbol{\theta}}(\boldsymbol{x}_i \mid y_i) + \log P(y_i)]$$

- **Unsupervised Generative Objective**: Models raw marginal density $P(X)$:

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x} \sim P_{\text{data}}} [\log P_{\boldsymbol{\theta}}(\boldsymbol{x})]$$

Mathematical Foundations: Discriminative vs. Generative II

Core Challenge of Generative Modeling

Real-world data (e.g., 1024×1024 images) live in extremely high-dimensional spaces ($\mathbb{R}^{3,145,728}$). Explicitly computing high-dimensional density $P(\boldsymbol{x})$ is intractable; generative models rely on deep neural networks to approximate or sample from this manifold.

Taxonomy of Generative Models I

Generative models are broadly categorized based on how they handle the data density $P(X)$:

1. Explicit Density Models

Model exact or approximate density function $P_{\boldsymbol{\theta}}(\boldsymbol{x})$ explicitly.

- **Tractable Density**:
  - *Autoregressive Models*: Factorize $P(\boldsymbol{x}) = \prod_{j=1}^{d} P(x_j \mid \boldsymbol{x}_{<j})$ (e.g., GPT, PixelCNN).
- **Approximate Density**:
  - *Variational Autoencoders (VAEs)*: Maximize Evidence Lower Bound (ELBO).
  - *Normalizing Flows*: Exact transformation via invertible maps.

2. Implicit Density Models

Do not define explicit density $P(\boldsymbol{x})$; instead, provide a sampling mechanism $\boldsymbol{x} = G(\boldsymbol{z})$ from latent noise $\boldsymbol{z}$.

- **Generative Adversarial Networks (GANs)**: Game-theoretic optimization between Generator and Discriminator.
- **Diffusion / Score-Based Models**: Denoising iterative stochastic processes (DDPMs, SGM).

Foundational Generative Architectures I

1. Variational Autoencoders (VAEs)

Encoder maps data to latent distribution parameters $(\boldsymbol{\mu}, \boldsymbol{\sigma})$; Decoder reconstructs data. Optimizes the Evidence Lower Bound (ELBO):

$$\mathcal{L}_{\text{ELBO}}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})] - D_{\text{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,||\, p(\boldsymbol{z})\right)$$

2. Generative Adversarial Networks (GANs)

Minimax game between Discriminator $D_{\boldsymbol{\phi}}$ and Generator $G_{\boldsymbol{\theta}}$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim P_{\text{data}}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}}[\log(1 - D(G(\boldsymbol{z})))]$$

3. Diffusion Models (DDPM)

Progressive noise addition $q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1})$ and learned reverse denoising $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$:

$$\mathcal{L}_{\text{simple}}(\boldsymbol{\theta}) = \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\|^2\right]$$

Generative AI Modalities & Applications I

- **Text & Code Generation**:
  - *Architectures*: Transformer Decoders (Autoregressive LLMs: GPT-4, LLaMA).
  - *Applications*: Conversational agents, automated code generation, summarization.
- **Computer Vision (Image & Video Synthesis)**:
  - *Architectures*: Latent Diffusion Models (Stable Diffusion), StyleGAN.
  - *Applications*: Text-to-image synthesis, photorealistic video rendering, image inpainting.
- **Audio & Speech Generation**:
  - *Architectures*: WaveNet, Audio Spectrogram Transformers.
  - *Applications*: Text-to-speech (TTS), voice cloning, automated music composition.
- **Science & Tabular Data**:
  - *Applications*: De novo molecular design, protein folding prediction, synthetic data creation for privacy compliance.

Python Implementation: Generative Density Sampling I

Fitting a Gaussian Mixture Model (GMM) as a Generative Model

A Gaussian Mixture Model models $P(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ and enables sampling new data.

```
import numpy as np
from sklearn.mixture import GaussianMixture

# 1. Generate synthetic multimodal training data
np.random.seed(42)
cluster1 = np.random.normal(loc=-3.0, scale=0.8, size=(250, 2))
cluster2 = np.random.normal(loc=3.0, scale=1.2, size=(250, 2))
X_train = np.vstack([cluster1, cluster2])

# 2. Train Generative Model (Explicit Density Estimator)
gmm = GaussianMixture(n_components=2, covariance_type='full', random_state=42)
gmm.fit(X_train)

# 3. Sample new synthetic data points from learned P(X)
X_synthetic, y_synthetic_components = gmm.sample(n_samples=100)

print(f"Learned Component Means:\n{gmm.means_}")
print(f"Generated Synthetic Samples Shape: {X_synthetic.shape}")
```

Evaluating Generative Models & Key Challenges I

Evaluation Metrics

- **Fréchet Inception Distance (FID)**: Quantifies distance between feature representations of real and generated samples:

$$\text{FID} = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|^2 + \text{Tr}(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{1/2})$$

- **Perplexity**: Evaluates language model prediction likelihood.
- **Inception Score (IS)**: Evaluates sample quality and diversity.

Key Technical Challenges

- **Mode Collapse**: GAN generator produces limited variety of samples, missing modes of P_{data}.
- **Training Instability**: Vanishing gradients or non-convergence in min-max optimizations.
- **Hallucination**: LLMs generating factually incorrect yet confident content.
- **Computational Demands**: High memory and throughput requirements for inference/training.

Summary & Paradigm Comparison I

Feature	Discriminative AI	Generative AI
Primary Goal	Predict label Y given X	Model data distribution $P(X)$
Output	Scalar / Category Class	New Complex Sample (Text/Image)
Math Formulation	$P(Y \mid X)$	$P(X)$ or $P(X, Y)$
Key Models	Logistic Reg., SVM, ResNet	VAE, GAN, Diffusion, GPT
Primary Risk	Overfitting / Decision error	Mode Collapse / Hallucination

Key Takeaway

Generative AI represents a fundamental evolution from pattern **recognition** to pattern **creation**, enabling machines to model the underlying structure of high-dimensional data spaces and sample novel realizations.

Fundamental Paradigm: Discriminative vs. Generative I

Discriminative Modeling

Focuses on learning the conditional distribution $P(Y \mid X)$ or a decision boundary $f(X) \to Y$.

- **Goal**: Predict label Y given feature set X.
- **Examples**: Logistic Regression, SVMs, ResNet classifiers.

Generative Modeling

Focuses on estimating the joint distribution $P(X, Y)$ or data distribution $P(X)$.

- **Goal**: Model the underlying probability distribution of high-dimensional data X to sample novel instances $\tilde{x} \sim P_\theta(X)$.
- **Key Challenge**: High-dimensional probability estimation ($X \in \mathbb{R}^D$ where D is large).

Core Components of Generative Workflows I

Generative models transform simple noise distributions into complex target data distributions through three key mechanisms:

1. **Latent Space Representation (Z)**:
   - Maps raw high-dimensional data $\mathbf{X}$ to a lower-dimensional manifold $\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
   - Captures semantic attributes (e.g., lighting, pose, tone).
2. **Density Estimation / Mapping Function (g_θ)**:
   - Neural network parameterizes the transformation: $g_\theta : \mathbf{Z} \to \mathbf{X}$.
   - Explicit density models estimate $p_\theta(x)$; implicit models directly generate samples.
3. **Sampling & Decoding**:
   - Draw seed $\mathbf{z} \sim p(\mathbf{z})$ and compute $\mathbf{x}_{\text{gen}} = g_\theta(\mathbf{z})$.

1. Variational Autoencoders (VAEs) I

Probabilistic Formulation

VAEs optimize the Evidence Lower Bound (ELBO) using an Encoder $q_\phi(\mathbf{z} \mid \mathbf{x})$ and Decoder $p_\theta(\mathbf{x} \mid \mathbf{z})$:

$$\mathcal{L}_{\text{ELBO}}(\theta, \phi; \mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel p(\mathbf{z}))$$

The Reparameterization Trick

Direct sampling $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_\phi, \boldsymbol{\sigma}_\phi^2)$ blocks backpropagation.

- **Solution**: Isolate stochasticity by sampling standard noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$:

$$\mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{x}) + \boldsymbol{\sigma}_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}$$

- Enables gradient flow through deterministic network operations.

2. Generative Adversarial Networks (GANs) I

Adversarial Game Theory

GANs operate via a zero-sum game between Generator G_θ and Discriminator D_ϕ:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))]$$

- **Discriminator (D_ϕ)**: Tries to distinguish real data $\mathbf{x}$ from fake data $G(\mathbf{z})$.
- **Generator (G_θ)**: Maps noise $\mathbf{z}$ to fake samples, aiming to fool D.
- **Training Dynamics**: Reaches Nash Equilibrium when $p_g = p_{\text{data}}$ and $D(\mathbf{x}) = 0.5$.
- **Key Challenges**: Mode collapse, vanishing gradients, training instability.

3. Diffusion Models: Iterative Denoising I

Forward and Reverse Processes

1. **Forward Process (q)**: Gradually adds Gaussian noise to data $\mathbf{x}_0$ over T steps:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

2. **Reverse Process (p_θ)**: Neural network parameterizes noise removal:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

Training Objective

Predict the added noise $\boldsymbol{\epsilon}$ at step t given noisy sample $\mathbf{x}_t$:

$$\mathcal{L}_{\text{simple}}(\theta) = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right]$$

Chain Rule of Probability

Decomposes the joint probability of a sequence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ into conditional probabilities:

$$P(\mathbf{x}) = \prod_{t=1}^{T} P(x_t \mid x_1, x_2, \ldots, x_{t-1}) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$$

- **Mechanism**: Predicts the next element (token) given all preceding context.
- **Architecture**: Transformer Decoders using Causal Self-Attention masks.
- **Inference Strategies**:
  - **Greedy Decoding**: Select $\arg\max P(x_t \mid x_{<t})$.
  - **Top-k / Nucleus (Top-p) Sampling**: Sample from high-probability subset.
  - **Temperature Scaling**: Adjust logits via $P(x_i) = \frac{\exp(z_i/T)}{\sum \exp(z_j/T)}$.

Comparative Analysis of Generative Paradigms I

Model Family	Likelihood	Sampling Speed	Sample Quality
VAE	Explicit (ELBO)	Fast (1 step)	Moderate / Blurry
GAN	Implicit	Fast (1 step)	High (Sharp)
Diffusion	Explicit	Slow (T steps)	State-of-the-Art
Autoregressive	Exact	Slow (Sequential)	High (Text/Audio)

Generative Trilemma

Ideal generative models seek three properties simultaneously:

1. High sample quality.
2. Fast sampling speed.
3. Mode coverage (diversity).

Existing architectures trade off between these three attributes.

Python: VAE Reparameterization & Sampling I

```python
import torch
import torch.nn as nn

class VAESampler(nn.Module):
    def __init__(self, latent_dim=20):
        super().__init__()
        self.latent_dim = latent_dim

    def reparameterize(self, mu, logvar):
        """
        Reparameterization Trick: z = mu + std * epsilon
        """
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)  # N(0, I)
        return mu + eps * std

    def decode_latent_noise(self, decoder, num_samples=16):
        # Sample directly from standard prior N(0, I)
        z = torch.randn(num_samples, self.latent_dim)
        with torch.no_grad():
            generated_samples = decoder(z)
        return generated_samples
```

Python: GAN Loss Functions I

```python
import torch
import torch.nn.functional as F

def compute_gan_losses(discriminator, generator, real_imgs, z):
    # 1. Train Discriminator: Maximize log(D(x)) + log(1 - D(G(z)))
    fake_imgs = generator(z)
    d_real_logits = discriminator(real_imgs)
    d_fake_logits = discriminator(fake_imgs.detach())

    loss_D_real = F.binary_cross_entropy_with_logits(
        d_real_logits, torch.ones_like(d_real_logits))
    loss_D_fake = F.binary_cross_entropy_with_logits(
        d_fake_logits, torch.zeros_like(d_fake_logits))
    loss_D = (loss_D_real + loss_D_fake) / 2.0

    # 2. Train Generator: Maximize log(D(G(z)))
    d_fake_logits_g = discriminator(fake_imgs)
    loss_G = F.binary_cross_entropy_with_logits(
        d_fake_logits_g, torch.ones_like(d_fake_logits_g))

    return loss_D, loss_G
```

Evaluating Generative AI Models I

Fréchet Inception Distance (FID)

Measures distance between feature distributions of real (r) and generated (g) images extracted from an Inception-v3 network:

$$\text{FID} = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|^2 + \text{Tr}\left(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{1/2}\right)$$

- Lower FID indicates higher visual quality and closer distribution match.

Inception Score (IS) & Precision/Recall

- **Inception Score**: Evaluates sharpness $P(y \mid \mathbf{x})$ and diversity $P(y)$.
- **Precision vs. Recall**: Precision measures quality (fidelity of samples); Recall measures coverage of real data modes.

Summary: Working of Generative AI I

- **Core Goal**: Model data distribution $P(\mathbf{X})$ to sample new, realistic data points.
- **Latent Space**: Low-dimensional structured representation driving generative controllers.
- **Diverse Mechanics**:
  - **VAE**: ELBO optimization via reparameterization trick.
  - **GAN**: Minimax adversarial game between G and D.
  - **Diffusion**: Iterative noise estimation and removal over discrete timestamps.
  - **Autoregressive**: Sequentially sampling next tokens conditioned on past tokens.
- **Evaluation**: Quantitative evaluation via metrics like FID, IS, and Precision/Recall balance.

Overview & Domain Taxonomy of Generative AI I

Generative AI synthesizes novel, high-dimensional data points $\tilde{\mathbf{x}} \sim P_{\text{model}}(\mathbf{X})$ across diverse modalities:

Taxonomy of Generative Modalities

- **Text & Language**: Autoregressive language modeling, dialog, summarization, and translation.
- **Vision & Graphics**: Image synthesis, video generation, 3D asset generation, and neural rendering.
- **Code & Software**: Code completion, automated bug fixing, unit test generation, and docstring synthesis.
- **Science & Healthcare**: Protein design, molecular generation, materials discovery, and EHR synthesis.

Operational Paradigms

- **Unconditional**: Sample directly from latent space $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \rightarrow \mathbf{x}$.
- **Conditional ($P(\mathbf{X} \mid \mathbf{C})$)**: Guided synthesis using prompt, class label, or layout constraint $\mathbf{C}$.

Text Generation & Decoding Strategies I

Autoregressive Parameterization

Text generation factors joint likelihood over tokens $\mathbf{w} = (w_1, w_2, \ldots, w_T)$ via the chain rule:

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t \mid w_1, w_2, \ldots, w_{t-1}; \boldsymbol{\theta})$$

Sampling Techniques for Text Synthesis

Given predicted logits $\mathbf{z}_t$, token probabilities are computed via Temperature Scaling:

$$P(w_t = i \mid w_{<t}) = \frac{\exp(z_{t,i}/T)}{\sum_j \exp(z_{t,j}/T)}$$

- **Temperature (T)**: $T \to 0$ approaches greedy decoding; higher T increases diversity.
- **Top-k Sampling**: Truncates candidate pool to the top k most probable tokens.
- **Top-p (Nucleus) Sampling**: Selects smallest set of tokens whose cumulative probability $\geq p$.

Retrieval-Augmented Generation (RAG) I

RAG addresses static knowledge cutoffs and hallucinations by coupling parametric memory (LLM) with non-parametric vector databases.

RAG Algorithmic Workflow

1. **Query Embedding**: Embed user query $\mathbf{q} \in \mathbb{R}^d$ via encoder $E_Q(\mathbf{q})$.
2. **Dense Retrieval**: Fetch top-k relevant document chunks $\mathcal{D}_k$ using cosine similarity:

$$\text{sim}(\mathbf{q}, \mathbf{d}_i) = \frac{E_Q(\mathbf{q})^\top E_D(\mathbf{d}_i)}{\|E_Q(\mathbf{q})\| \|E_D(\mathbf{d}_i)\|}$$

3. **Augmented Generation**: Condition LLM on retrieved context $\mathcal{D}_k$:

$$P(\mathbf{y} \mid \mathbf{q}) = \prod_{t=1}^{|Y|} P(y_t \mid y_{<t}, \mathbf{q}, \mathcal{D}_k)$$

Key Advantage

Enables domain-specific QA and real-time knowledge integration without expensive model fine-tuning.

Latent Diffusion Models (LDMs)

LDMs perform diffusion in a compressed latent space $\mathbf{z} = \mathcal{E}(\mathbf{x})$, drastically reducing computational cost:

- **Conditioning Mechanism**: Cross-attention integrates prompt embeddings $\tau_\theta(\mathbf{y})$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V$$

$$Q = W_Q^{(i)} \phi_i(z_t), \; K = W_K^{(i)} \tau_\theta(y)$$

Key Vision Applications

- **Text-to-Image / Video**: Generating photorealistic artwork and videos from natural language.
- **Inpainting & Outpainting**: Restoring corrupted regions or expanding image boundaries.
- **Spatial Control (ControlNet)**: Adding structural conditioning (e.g., edge maps, depth maps, human poses) to diffusion models.

Code-Specialized Language Models

Models trained on source code (e.g., StarCoder, CodeLlama) leverage structured syntax (ASTs) and high context lengths.

- **Task Scenarios**: Autocompletion, text-to-code translation, docstring generation, unit test creation.
- **Fill-in-the-Middle (FIM)**: Trained to predict missing code given prefix and suffix: $P(\text{middle} \mid \text{prefix}, \text{suffix})$.

Evaluation Benchmark Metric: pass@k

Measures the probability that at least one of k generated code samples passes all functional unit tests:

$$\text{pass@k} = \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

where n is total generated samples per problem ($n \geq k$) and c is the number of correct samples ($c \leq n$).

Python Implementation: Generative Pipelines I

Generating Text and Images using Hugging Face Ecosystem

```
import torch
from transformers import pipeline
from diffusers import StableDiffusionPipeline

# 1. Text Generation with an LLM pipeline
text_gen = pipeline("text-generation", model="gpt2")
prompt = "Unsupervised learning discovers hidden patterns by"
text_output = text_gen(prompt, max_length=50, num_return_sequences=1)
print("Generated Text:", text_output[0]['generated_text'])

# 2. Text-to-Image Generation with Latent Diffusion
model_id = "runwayml/stable-diffusion-v1-5"
pipe = StableDiffusionPipeline.from_pretrained(
    model_id, torch_dtype=torch.float16
)
pipe = pipe.to("cuda")

image_prompt = "A high-tech neural network visualization, futuristic style"
image = pipe(image_prompt).images[0]
image.save("generated_network.png")
```

Generative AI in Science & Healthcare I

De Novo Molecular & Protein Design

Generative architectures create novel physical structures satisfying specific biological constraints:

- **Molecular Graph Generation**: VAEs and Flow models generate target small molecules parameterized as SMILES strings or molecular graphs.
- **Protein Backbone Design**: Diffusion models (e.g., RFdiffusion) generate de novo 3D protein structures tailored for receptor binding.

Healthcare Data Privacy & Augmentation

- **Synthetic EHR Generation**: Generative adversarial networks generate privacy-preserving synthetic patient medical records.
- **Medical Image Augmentation**: Diffusion models generate synthetic MRI/CT scans for rare disease classification tasks.

Quantitative Evaluation of Generative Models I

Evaluating generative models requires assessing sample quality, diversity, and fidelity.

Image Evaluation Metrics

- **Fréchet Inception Distance (FID)**: Measures distance between real (r) and generated (g) feature distributions in Inception-v3 latent space:

$$\text{FID} = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|_2^2 + \text{Tr}\left(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{1/2}\right)$$

- **Inception Score (IS)**: Evaluates sample quality and diversity via conditional class distributions.
- **LPIPS**: Perceptual similarity distance based on deep feature representations.

Text Evaluation Metrics

- **Perplexity (PPL)**: Exponentiated average negative log-likelihood per token.
- **BERTScore & LLM-as-a-Judge**: Semantic similarity matching using contextual embeddings or strong evaluator LLMs (e.g., GPT-4).

Safety, Alignment & Open Technical Challenges I

Human Alignment Strategies

- **RLHF**: Aligning model responses using a Reward Model $R_\psi(\mathbf{x}, \mathbf{y})$ and PPO optimization:

$$\max_\theta \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim D}\left[R_\psi(\mathbf{x},\mathbf{y})\right] - \beta D_{\mathrm{KL}}\left(\pi_\theta(\mathbf{y} \mid \mathbf{x}) \parallel \pi_{\mathrm{ref}}(\mathbf{y} \mid \mathbf{x})\right)$$

- **Direct Preference Optimization (DPO)**: Eliminates explicit reward modeling by optimizing directly on preference pairs $(y_w \succ y_l)$.

Critical Challenges

- **Hallucination & Factuality**: Generating plausible but factually incorrect assertions.
- **Mode Collapse**: Loss of output diversity in GANs and diffusion processes.
- **Copyright & Provenance**: Watermarking generated media (e.g., SynthID) to track authenticity.