Foundations of Machine Learning & Problem Framing

Generated by Pustakam Injin

AI-Powered Knowledge Engine • **gpt-oss-120b**

Pustakam Injin

AI-Powered Knowledge Creation

Document Information

Word Count	**33,527**
Chapters	**10**
Generated	**February 14, 2026**
AI Model	**Cerebras - gpt-oss-120b**

Tanmay Kalbande

Creator & Lead Architect

pustakamai.tanmaysk.in

linkedin.com/in/tanmay-kalbande

Foundations of Machine Learning & Problem Framing

Generated: 14/02/2026 **Words:** 33,527 **Provider:** Cerebras (gpt-oss-120b)

Introduction

Welcome to the Journey of Building Predictive Models

Congratulations-by picking up this book you've taken the first step toward turning raw data into actionable insight. Whether you've already dabbled in data analysis, completed a few online tutorials, or simply feel the pull of "what-if" questions that keep you up at night, this guide is designed to meet you where you are and propel you forward.

**Learn Machine Learning for Building Predictive Models** is more than a collection of algorithms; it is a roadmap that walks you through the entire lifecycle of a machine-learning project-from spotting the right problem to keeping your model honest long after it goes live. The pages ahead are packed with practical examples, visual intuition, and hands-on exercises that let you apply each concept to real-world data sets as you read. By the end, you'll not only understand how the most common models work, you'll also know **when** to use them, **how** to fine-tune them, and **why** responsible deployment matters.

What You'll Learn

By the time you close the final chapter, you will be comfortable with the following core competencies:

- **Framing Machine-Learning Problems** - Translate business questions into data-driven objectives, choose the right prediction type (classification, regression, ranking, etc.), and set realistic success metrics.
- **Exploratory Data Analysis (EDA)** - Use visual and statistical tools to uncover patterns, spot anomalies, and formulate hypotheses before any modeling begins.
- **Data Preprocessing & Feature Engineering** - Clean messy data, handle missing values, encode categorical variables, scale numeric features, and create new predictors that boost model performance.
- **Supervised Learning Fundamentals** - Build, interpret, and evaluate linear models (e.g., logistic regression, ridge/lasso) and tree-based ensembles (random forests, gradient boosting).
- **Model Evaluation & Hyper-Parameter Tuning** - Apply cross-validation, learning curves, and robust metrics; conduct systematic grid- and random-search to squeeze out the best version of your model.
- **Unsupervised Learning for Feature Extraction** - Leverage clustering, dimensionality-reduction (PCA, t-SNE, UMAP) and autoencoders to discover hidden structure and reduce noise.
- **Specialized Data Scenarios** - Tackle imbalanced class problems with resampling and cost-sensitive learning, and model temporal dependencies in time-series data using lag features and rolling windows.
- **Deploying Predictive Models** - Package models into reproducible pipelines, expose them via RESTful APIs, and integrate them into production systems with tools like Docker and FastAPI.
- **Ethics, Fairness, and Monitoring** - Identify bias sources, implement fairness checks, and set up continuous monitoring to detect drift, degradation, or unintended consequences after deployment.

These outcomes are deliberately **action-oriented**. You won't just finish the book with a theoretical checklist-you'll leave with a portfolio of end-to-end projects that you can showcase to employers, collaborators, or even your own organization.

How the Book Is Structured

To keep the learning curve gentle yet comprehensive, the book follows a logical progression that mirrors a real-world project pipeline. Each major section is divided into bite-sized chapters, and every chapter ends with a **"Try It Yourself"** mini-project, a **reflection quiz**, and a set of **further-reading suggestions**.

1. Foundations of Machine Learning & Problem Framing

Goal: Ground you in the language of ML, clarify the distinction between supervised, unsupervised, and reinforcement learning, and teach you to write a crisp problem statement.

2. Data Exploration & Visualization

Goal: Equip you with visual storytelling tools (Matplotlib, Seaborn, Plotly) and statistical summarization techniques to let the data speak before you code.

3. Data Preprocessing & Feature Engineering

Goal: Turn raw tables into model-ready matrices. Topics include missing-value imputation, outlier handling, categorical encoding, scaling, and the art of creating interaction features.

4. Supervised Learning - Linear Models

Goal: Demystify the math behind linear regression, logistic regression, and regularized variants while showing you how to interpret coefficients and confidence intervals.

5. Supervised Learning - Tree-Based Models

Goal: Dive into decision trees, random forests, and gradient-boosted machines (XGBoost, LightGBM, CatBoost). You'll learn why these models often dominate Kaggle competitions and how to tame their complexity.

6. Model Evaluation, Validation & Hyper-Parameter Tuning

Goal: Master cross-validation strategies, choose the right loss functions, construct ROC/PR curves, and automate hyper-parameter searches using Scikit-Learn's `GridSearchCV` and `RandomizedSearchCV`.

7. Unsupervised Learning for Feature Extraction

Goal: Apply k-means, hierarchical clustering, DBSCAN, and dimensionality-reduction to discover latent structures that can be fed back into supervised models.

8. Working with Imbalanced & Time-Series Data

Goal: Learn techniques such as SMOTE, class weighting, and focal loss for imbalance; and build lag-based features, rolling statistics, and simple ARIMA baselines for temporal data.

9. Deploying Predictive Models - Pipelines & APIs

Goal: Build reproducible pipelines with `sklearn.pipeline` and `mlflow`, containerize models with Docker, and expose them as scalable APIs using FastAPI or Flask.

10. Ethical Considerations & Model Monitoring

Goal: Examine fairness metrics, privacy-preserving methods, and set up automated alerts for data drift, performance decay, and ethical violations.

Each chapter follows a consistent template:

1. **Concept Overview** - Plain-language explanation with analogies.

2. **Mathematical Insight** - Optional deeper dive for the curious.
3. **Hands-On Code** - End-to-end notebooks you can run locally or in the cloud.
4. **Pitfalls & Tips** - Common mistakes and how to avoid them.
5. **Mini-Project** - A short, self-contained exercise that reinforces the lesson.

Why This Book?

1. It Bridges Theory and Practice

Many introductory texts stop at "what an algorithm does." Here, every algorithm is paired with a real-world scenario-predicting churn for a subscription service, forecasting electricity demand, detecting fraudulent transactions, and more. You'll see not just ***how*** a model works, but ***why*** it matters in a business context.

2. It Emphasizes the Full Lifecycle

Too often, learners focus solely on model training and ignore the steps that come before (data wrangling) and after (deployment, monitoring). By treating the workflow as an integrated whole, you'll gain the confidence to take a project from a spreadsheet to a production-grade service.

3. It Keeps the Barrier Low, the Impact High

You are an ***intermediate*** learner-comfortable with Python, basic statistics, and perhaps a taste of Scikit-Learn. This book builds on that foundation without drowning you in graduate-level math. Yet the projects you complete will be portfolio-ready for data-science interviews or internal stakeholder demos.

4. It Prioritizes Ethical AI

Predictive models can amplify bias or cause unintended harm if left unchecked. The

final chapter equips you with concrete tools-fairness dashboards, model cards, and continuous monitoring pipelines-so you can ship responsible AI, not just accurate AI.

What to Expect as You Turn the Pages

- **Active Learning:** After each concept you'll be prompted to pause, run a notebook, and experiment with your own data set (the book provides a "sandbox" CSV you can replace with anything you like).
- **Progressive Complexity:** Early chapters introduce visual intuition; later chapters layer in mathematical rigor and engineering best practices.
- **Community Support:** A companion GitHub repository hosts all code, data, and a discussion forum where you can share results, ask questions, and collaborate on extensions.
- **Realistic Timelines:** The "Try It Yourself" sections are designed to be completed in 15-30 minutes, so you can fit learning into a busy schedule without feeling overwhelmed.
- **Reflection Moments:** Quick quizzes at the end of each chapter help cement knowledge and highlight areas that may need a second look.

Getting Started: Your First Steps

1. **Set Up Your Environment** - Install Python 3.10+, create a virtual environment, and clone the companion repo. A single `requirements.txt` file will pull in all the libraries you need (NumPy, pandas, Scikit-Learn, Matplotlib, Seaborn, Plotly, FastAPI, Docker, etc.).
2. **Run the Intro Notebook** - Chapter 1includes a "Hello, Predictive Modeling!" notebook that walks you through loading the starter data, performing a quick EDA, and formulating a simple binary-classification problem. This will give you a taste of the workflow and confirm that everything is wired correctly.

3. **Mark Your Milestones** - As you progress, use the provided checklist to track completed chapters, mastered concepts, and finished mini-projects. Celebrate each milestone; the habit of marking progress is a powerful motivator.

A Final Word of Inspiration

Imagine a world where you can look at a spreadsheet of customer interactions and instantly forecast who will churn, which product will sell out, or where a machine is likely to fail. Picture yourself building a recommendation engine that feels like magic to users, or a fraud detector that saves millions of dollars each year. Those outcomes are not the domain of a select few data-science elites-they are attainable for anyone willing to learn the systematic craft of predictive modeling.

This book is your companion on that quest. It will challenge you, guide you, and-most importantly- empower you to turn data into decisions. So, roll up your sleeves, fire up your IDE, and let's start building models that not only predict the future but shape it responsibly.

Welcome aboard. Let the learning begin!

Foundations of Machine Learning & Problem Framing

Module 1of 10- "Learn Machine Learning for Building Predictive Models"

Introduction

Machine learning (ML) has moved from research labs to boardrooms, factories, and

everyday consumer apps. Yet, before you can write a line of code that "learns," you must first **understand the problem you are trying to solve** and the **type of learning** that best fits that problem.

In this chapter we will:

1. Define **predictive modeling** and distinguish the three major learning paradigms-**supervised**, **unsupervised**, and **reinforcement** learning.
2. Show how to **match business questions** with the appropriate ML paradigm.
3. Walk through the **end-to-end ML workflow**, from data acquisition to model deployment.
4. Explain two foundational statistical ideas-**bias-variance trade-off** and **overfitting**-that govern model performance.

By the end of the chapter you should be able to look at a real-world business challenge, decide whether an ML solution makes sense, sketch a high-level workflow, and anticipate the kinds of pitfalls you'll need to guard against when you start coding.

> ***Why this matters:***

> ***Most ML projects fail not because the algorithm is "wrong," but because the problem was framed incorrectly, the data were mis-aligned with the business goal, or the model was over-tuned to historical noise. A solid foundation in problem framing is the single most valuable skill you can bring to any data-driven initiative.***

Core Concepts

1. What Is Predictive Modeling?

Predictive modeling is the process of **using historical data to estimate future outcomes**. In statistical terms, you fit a function ***f*** that maps input variables

(features) **X** to a target variable (label) **y**:

$$\hat{y} = f(\mathbf{X}; \theta)$$

where θ represents the model parameters learned from data.

Key characteristics

Aspect	Description
Goal	Forecast a quantity (e.g., sales) or classify an event (e.g., churn).
Input	Structured tabular data, text, images, sensor streams, etc.
Output	Numeric (regression) or categorical (classification).
Evaluation	Metrics that compare predictions $\hat{y}$ to true outcomes y (RMSE, MAE, Accuracy, AUC-ROC, etc.).
Iterative	Model is refined by looping through data collection, training, validation, and deployment.

> *Practical example: A telecom company wants to predict whether a customer will cancel their service next month. The target is a binary label (`churn = 1` or `0`). The features may include usage minutes, contract length, payment history, and customer-service call sentiment.*

---

2. The Three Learning Paradigms

Paradigm	Core Idea	Typical Targets	Example Business Problems
Supervised Learning	Learn a mapping from	Regression	Credit-risk scoring,

Paradigm	Core Idea	Typical Targets	Example Business Problems
	labeled inputs → outputs.	(continuous), Classification (discrete).	demand forecasting, image classification.
Unsupervised Learning	Discover structure **without** explicit labels.	Clustering, Dimensionality Reduction, Anomaly detection.	Customer segmentation, fraud pattern discovery, topic modeling.
Reinforcement Learning (RL)	Learn a **policy** that maximizes cumulative reward through trial-and-error interaction with an environment.	Decision-making policies, sequential control.	Dynamic pricing, inventory replenishment, autonomous navigation.

2.1 Supervised Learning - The "Classic" Predictive Model

- **Training data**: \(\{(\mathbf{x}***i*, *y*i)\}_{i=1}^N\).
- **Loss function**: Quantifies prediction error (e.g., cross-entropy for classification).
- **Goal**: Minimize loss on the training set while preserving generalization to unseen data.

Common algorithms

Category	Algorithms	Typical Use-Cases
Linear models	Linear Regression, Logistic Regression	Baseline sales forecasts, churn probability.
Tree-based	Decision Trees, Random Forests, Gradient Boosted Trees (XGBoost, LightGBM)	Tabular data with non-linear interactions, credit scoring.
Neural networks	Fully-connected, Convolutional (CNN), Recurrent (RNN)	Image classification, time-series demand, text sentiment.
Kernel methods	Support Vector Machines (SVM)	Small-to-medium datasets with complex boundaries.

2.2 Unsupervised Learning - Finding Patterns in the Dark

- **Training data**: \(\{\mathbf{x}***i*\}**{i=1}^N\) - no ***y***.
- **Objective**: Optimize a criterion that captures internal structure (e.g., intra-cluster similarity).

Common techniques

Technique	What it does	Example
Clustering (K-means, DBSCAN, Hierarchical)	Groups similar records.	Segmenting shoppers into "bargain hunters" vs "brand loyalists".
Dimensionality Reduction (PCA, t-SNE, UMAP)	Projects high-dim data to low-dim while preserving variance or local structure.	Visualizing high-dim sensor data for anomaly spotting.
Association Rule Mining (Apriori)	Finds frequent itemsets and rules.	Market-basket analysis: "Customers who buy coffee also buy donuts".
Autoencoders (Neural)	Learns compressed representation; reconstruction error can flag outliers.	Detecting fraudulent transactions.

2.3 Reinforcement Learning - Learning by Interaction

- **Agent** interacts with **environment** over discrete time steps ***t***.
- At each step, the agent selects an **action** ***a*□**, receives a **reward** ***r*□**, and observes a new **state** **s□ □.**,
- Goal: Learn a **policy** π(a|s) that maximizes expected cumulative reward \(\mathbb{E}\left[\sum***{t=0}^{\infty} \gamma^t r***t\right] \) (γ = discount factor).

Typical algorithms

Family	Algorithms	Business Context

Family	Algorithms	Business Context
Value-based	Q-learning, Deep Q-Network (DQN)	Inventory replenishment with stochastic demand.
Policy-gradient	REINFORCE, Proximal Policy Optimization (PPO)	Real-time bidding in digital advertising.
Actor-Critic	A2C, DDPG, SAC	Robotics, autonomous vehicle lane-keeping.

3. Mapping Business Problems to ML Paradigms

Business Question	Does the problem have a known outcome?	Is the outcome categorical or continuous?	Does the problem involve grouping or discovering hidden structure?	Does the solution require sequential decision making?
"Will this customer churn next month?"	**Yes** - churn label exists	Binary (categorical)	No	No
"Which customers are most similar for a targeted campaign?"	No - no explicit label	N/A	**Yes** - clustering	No
"What price should we set for each ad impression to maximize revenue?"	**Yes**, but reward is observed only after action	Continuous reward	No	**Yes** - sequential policy
"How many units will we sell next quarter?"	**Yes**, historical sales data	Continuous (regression)	No	No
"Detect	**Partial** - some	Binary (fraud /	**Yes** - anomalies	No (though you

Business Question	Does the problem have a known outcome?	Is the outcome categorical or continuous?	Does the problem involve grouping or discovering hidden structure?	Does the solution require sequential decision making?
fraudulent transactions in real time."	fraud cases are labeled, many are not	not fraud)		may later use RL for fraud-response policies)

Rule of thumb:

- **Supervised** if you have a ***clear target*** you can label.
- **Unsupervised** if you need to ***explore*** or ***segment*** data without a target.
- **RL** if the solution involves ***choosing actions*** that affect future states and you can measure ***reward*** over time.

---

4. The End-to-End ML Workflow

Below is a **canonical pipeline** that applies to all three paradigms, though some steps (e.g., label generation) differ.

```
1️⃣ Business Understanding → 2️⃣ Data Acquisition → 3️⃣ Data Exploration & Pre-processing
   ↓                                   ↓                                   ↓
4️⃣ Problem Framing (Supervised/Unsupervised/RL) → 5️⃣ Feature Engineering → 6️⃣ Model Selection
   ↓                                   ↓                                   ↓
7️⃣ Model Training → 8️⃣ Model Evaluation → 9️⃣ Hyper-parameter Tuning → 10️⃣ Model Validation
   ↓                                   ↓                                   ↓
11️⃣ Deployment (Batch/Online) → 12️⃣ Monitoring & Maintenance → 13️⃣ Feedback Loop
```

4.1 Step-by-Step Detail

Step	What you do	Typical Tools / Deliverables
1. Business Understanding	Define success metrics (e.g., lift, ROI). Identify stakeholders and constraints (privacy, latency).	Problem statement doc, KPI list.
2. Data Acquisition	Pull data from databases, APIs, logs, third-party providers. Ensure legal compliance (GDPR, CCPA).	SQL queries, data lake ingestion scripts, data catalog entries.
3. Data Exploration & Pre-processing	Summary statistics, missing-value analysis, outlier detection, visualizations. Clean and transform data (imputation, encoding).	Pandas, pandas-profiling, Jupyter notebooks, data quality reports.
4. Problem Framing	Choose learning paradigm; decide on target variable, evaluation metric, and baseline model.	Decision matrix, baseline model (e.g., mean predictor).
5. Feature Engineering	Create informative variables (aggregations, time-lags, embeddings). Perform feature scaling, encoding, and selection.	Feature store, scikit-learn `ColumnTransformer`, `Featuretools`.
6. Model Selection	Pick a family of algorithms that matches data size, interpretability needs, and latency constraints.	Model cards, comparison tables.
7. Model Training	Fit the model on a **training set**. Use cross-validation to estimate variance.	scikit-learn `fit()`, TensorFlow/Keras `model.fit()`.
8. Model Evaluation	Compute metrics on a **hold-out test set**. Plot ROC curves, residuals, calibration.	`sklearn.metrics`, `mlflow` tracking.
9. Hyper-parameter Tuning	Search over parameter space (grid, random, Bayesian). Guard against leakage.	`optuna`, `scikit-optimize`, `Ray Tune`.
10. Model Validation	Perform ***out-of-time*** validation or ***shadow*** testing	A/B test plan, statistical significance calculator.

Step	What you do	Typical Tools / Deliverables
	on live traffic.	
11. Deployment	Export model artifact (Pickle, ONNX, TensorFlow SavedModel). Serve via REST API, batch job, or edge device.	Docker, FastAPI, AWS SageMaker, Azure ML, TensorFlow Lite.
12. Monitoring & Maintenance	Track data drift, prediction distribution, latency, error rates. Set alerts.	Evidently AI, Prometheus, Grafana, Model-drift dashboards.
13. Feedback Loop	Capture new labeled data, retrain periodically, update feature store.	CI/CD pipelines (`GitHub Actions`, `Kubeflow Pipelines`).

> ***Pro tip: Keep metadata (data lineage, feature definitions, model versions) in a centralized registry. This reduces technical debt when you revisit a model months later.***

---

5. Bias-Variance Trade-off

A model's error on unseen data can be decomposed into three components:

$$\text{Error} = \underbrace{\text{Bias}^2}_{\textit{Systematic error}} + \underbrace{\text{Variance}}_{\text{Sensitivity to training data}} + \underbrace{\text{Irreducible noise}}_{\text{Randomness in data}}$$

Concept	Intuition	High Bias →	High Variance →
Bias	Model's assumptions are too simple; it cannot capture the underlying pattern.	**Under-fitting** - predictions stay near the global mean.	**Over-fitting** - model is too flexible, fitting noise.
Variance	Model changes	Stable but	Accurate on training

Concept	Intuition	High Bias →	High Variance →
	dramatically with small changes in the training set.	inaccurate.	data, poor on new data.

Visual analogy: Imagine fitting a curve to a scatter plot. A straight line (high bias) misses the curvature. A high-degree polynomial (high variance) wiggles through every point, including outliers.

5.1 Managing the Trade-off

Technique	Effect on Bias	Effect on Variance	When to Use
Simplify model (e.g., linear regression)	↑ (more bias)	↓ (less variance)	Small dataset, need interpretability.
Complexify model (e.g., deep NN)	↓ (less bias)	↑ (more variance)	Large dataset, non-linear relationships.
Regularization (L1/L2, dropout)	Slight ↑	Significant ↓	Prevent over-fitting while keeping expressive power.
Ensemble methods (Bagging, Random Forest)	Slight ↓	Significant ↓	Reduce variance without sacrificing much bias.
Cross-validation	N/A	N/A	Diagnose whether error is due to bias or variance.

Practical tip: Plot **learning curves** (training vs validation error) as a function of training set size.

- If both errors are high → high bias → increase model capacity.
- If training error is low but validation error is high → high variance → add regularization or more data.

6. Overfitting - The Classic Pitfall

Overfitting occurs when a model captures **noise** (random fluctuations) as if it were a genuine pattern. The model performs spectacularly on the training data but fails catastrophically on new data.

6.1 Symptoms

- **Training accuracy ≈ 100 % ** validation accuracy ≈ 60 %(or lower).
- **Sharp spikes** in loss curves after a certain number of epochs.
- **Feature importance** dominated by idiosyncratic variables (e.g., "customer ID").

6.2 Causes

Source	Example
Too many features relative to observations (curse of dimensionality).	10 000one-hot encoded product IDs for 2 000 customers.
Complex model with many parameters and limited data.	Deep CNN on a few hundred labeled images.
Data leakage - using information that would not be available at prediction time.	Including "future sales" as a feature when predicting next-month sales.
Inadequate regularization (no dropout, no L2).	Training a linear model without ridge penalty on noisy data.

6.3 Remedies

Remedy	How it works
Hold-out validation (train/val/test split)	Guarantees performance estimate on unseen data.
Cross-validation (k-fold)	Reduces variance of performance estimate, reveals over-fitting patterns.
Early stopping (monitor validation loss)	Stops training before the model memorizes

Remedy	How it works
	noise.
Regularization (L1/L2, dropout, pruning)	Penalizes large weights, forces simpler models.
Feature selection / dimensionality reduction (PCA, mutual information)	Removes irrelevant/noisy features.
Data augmentation (synthetic samples, SMOTE)	Increases effective training size for under-represented classes.
Ensemble averaging (bagging, stacking)	Smooths out idiosyncratic predictions of individual models.

Practical Application

Let's walk through a **complete mini-project** that illustrates the concepts above. We'll use a **customer churn prediction** scenario-a classic supervised classification problem.

7. Business Context

Company: **Acme Telecom** ***Goal:*** Identify customers likely to churn next month to target retention offers. ***Success Metric:*** Increase retention rate by **5** **%** while keeping marketing spend under **$200** **k**per month (i.e., a **lift** of at least **1.2** over random targeting).

8. Data Overview

Table	Columns (excerpt)	Description
`customers`	`customer_id`, `signup_date`, `plan_type`, `region`	Demographic & contract info.

Table	Columns (excerpt)	Description
`usage`	`customer_id`, `date`, `minutes`, `sms`, `data_gb`	Daily usage metrics.
`billing`	`customer_id`, `month`, `amount`, `payment_method`, `late_flag`	Billing history.
`support`	`customer_id`, `ticket_id`, `date`, `category`, `sentiment_score`	Customer-service interactions.
`churn_label`	`customer_id`, `churn_next_month` (0/1)	Target variable derived from contract termination records.

Key points for framing

- **Label exists** → supervised classification.
- **Outcome is binary** → evaluation via ROC-AUC, Precision-Recall, and business-specific lift.
- **Time dimension** → need to respect **temporal ordering** (no leakage).

9. End-to-End Walkthrough (Python-style pseudo-code)

> ***Assumptions: Pandas, scikit-learn, XGBoost installed. The notebook is run on a modest VM.***

```
# 1️⃣ Load data ---------------------------------------------------------------
import pandas as pd
customers   = pd.read_csv('customers.csv')
usage       = pd.read_csv('usage.csv')
billing     = pd.read_csv('billing.csv')
support     = pd.read_csv('support.csv')
label       = pd.read_csv('churn_label.csv')

# 2️⃣ Merge & create a snapshot for month = '2024-01' (prediction month) -----
#    Note: we only use data up to Dec 2023 for training.
snapshot_date = '2023-12-31'

# Aggregate usage over the last 30 days per customer
usage_agg = (usage[usage['date'] <= snapshot_date]
             .groupby('customer_id')
             .agg(minutes=('minutes','sum'),
                  sms=('sms','sum'),
                  data_gb=('data_gb','sum')))

# Aggregate billing (last 3 months)
billing_agg = (billing[billing['month'] <= '2023-12']
               .groupby('customer_id')
               .agg(bill_avg=('amount','mean'),
                    late_rate=('late_flag','mean')))

# Sentiment: average of all tickets up to snapshot
support_agg = (support[support['date'] <= snapshot_date]
               .groupby('customer_id')
               .agg(sentiment=('sentiment_score','mean'),
                    tickets=('ticket_id','nunique')))

# Merge everything
df = (customers
      .merge(usage_agg, on='customer_id', how='left')
      .merge(billing_agg, on='customer_id', how='left')
      .merge(support_agg, on='customer_id', how='left')
      .merge(label, on='customer_id', how='left'))

# 3️⃣ Basic EDA ---------------------------------------------------------------
df.describe()
```

... (continued on next page)

```
df.isnull().mean()   # check missingness
```

Key EDA insights

- 15 %of customers churned in the next month (moderately imbalanced).
- `late_rate` and `sentiment` show strong negative correlation with churn.

```
# 4 Feature engineering ------------------------------------------------------
# Encode categorical fields
df = pd.get_dummies(df, columns=['plan_type','region','payment_method'],
                    drop_first=True)

# Impute missing values (median for numeric)
numeric_cols = df.select_dtypes(include='float64').columns
df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].median())

# Target / features split
X = df.drop(columns=['customer_id','churn_next_month'])
y = df['churn_next_month']
```

```
# 5 Train-test split respecting time (no leakage) -------------------------
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)   # 5 folds moving forward in time
```

```
# 6️⃣ Baseline model (Logistic Regression) ---------------------------------
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, precision_recall_curve

auc_scores = []
for train_idx, test_idx in tscv.split(X):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    lr = LogisticRegression(max_iter=500, class_weight='balanced')
    lr.fit(X_train, y_train)
    prob = lr.predict_proba(X_test)[:,1]
    auc = roc_auc_score(y_test, prob)
    auc_scores.append(auc)

print(f'Baseline ROC-AUC (logistic): {np.mean(auc_scores):.3f}')
```

Result: **0.71** ROC-AUC-reasonable but leaves room for improvement.

```
# 7️⃣ Gradient Boosted Trees (XGBoost) - more expressive --------------------
import xgboost as xgb
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [3,5,7],
    'learning_rate': [0.01,0.1],
    'n_estimators': [200,400],
    'subsample': [0.8,1.0],
    'colsample_bytree': [0.8,1.0]
}

xgb_clf = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='auc',
    use_label_encoder=False,
    scale_pos_weight= (len(y)-y.sum())/y.sum()   # handle imbalance
)

grid = GridSearchCV(xgb_clf, param_grid, cv=tscv, scoring='roc_auc',
                    verbose=1, n_jobs=-1)
grid.fit(X, y)

print('Best params:', grid.best_params_)
print('Best CV ROC-AUC:', grid.best_score_)
```

Result: **0.84** ROC-AUC after tuning-substantial lift over baseline.

```
# 8⃣ Model interpretation (SHAP) ------------------------------------------
import shap
explainer = shap.TreeExplainer(grid.best_estimator_)
shap_values = explainer.shap_values(X.sample(2000))

shap.summary_plot(shap_values, X.sample(2000), plot_type='bar')
```

Interpretation highlights

- `late_rate` and `sentiment` are top contributors to churn probability.
- High `data_gb` usage surprisingly ***decreases*** churn risk (loyal heavy users).

```
# 9⃣ Validation on a true hold-out month (Feb-2024) ------------------------
# Build a fresh snapshot for Feb-2024 using same pipeline
# (omitted for brevity - same steps as above)

# Predict probabilities
prob_test = grid.best_estimator_.predict_proba(X_test_feb)[:,1]

# Business metric: lift over random
from sklearn.metrics import precision_score

threshold = 0.30   # pick top 30⃣ %as "high risk"
pred = (prob_test >= threshold).astype(int)
precision = precision_score(y_test_feb, pred)
random_precision = y_test_feb.mean()
lift = precision / random_precision

print(f'Precision @30%: {precision:.3f}')
print(f'Random baseline precision: {random_precision:.3f}')
print(f'Lift: {lift:.2f}')
```

Result: **Lift ≈ 1.6**, exceeding the required 1.2.

```
# ⃣ Deploy - export model -------------------------------------------------
import joblib
joblib.dump(grid.best_estimator_, 'churn_xgb_v1.pkl')
```

Deployment options

- **Batch scoring** - nightly run on new customer snapshots, write predictions to a `churn_score` table.
- **Online API** - expose a FastAPI endpoint `/predict` that accepts a JSON payload of a single customer's latest features and returns a churn probability.

```
# Example FastAPI snippet (simplified)
from fastapi import FastAPI
import pandas as pd
import joblib

app = FastAPI()
model = joblib.load('churn_xgb_v1.pkl')

@app.post("/predict")
def predict(payload: dict):
    df = pd.DataFrame([payload])
    prob = model.predict_proba(df)[:,1][0]
    return {"churn_probability": float(prob)}
```

10. Lessons Learned from the Example

Lesson	Why it matters
Temporal split prevents leakage.	Using data after the prediction month would inflate performance.
Class imbalance handling (class***weight***, `***scale***pos_weight`).	Without it, the model would default to predicting "no churn" and achieve high accuracy but zero business value.
Feature importance via SHAP builds trust with stakeholders.	Marketing can see ***why*** a customer is flagged, enabling personalized offers.
Lift vs. ROC-AUC - business metrics matter more.	A model with a higher AUC may still produce low lift if predictions are poorly calibrated for the top-k segment.
Iterative monitoring is essential.	After deployment, data drift (e.g., new plan types) can degrade performance; set alerts on feature distribution shifts.

Key Takeaways

• **Predictive modeling** is the systematic mapping of historical data to future outcomes; it can be **regression** (continuous) or **classification** (categorical).

• **Supervised learning** needs labeled data; **unsupervised learning** discovers hidden structure; **reinforcement learning** learns optimal actions through trial-and-error rewards.

• **Problem framing** is the bridge between a business question and the appropriate ML paradigm. Use a decision matrix to verify that you have the right target, data, and evaluation metric.

• The **ML workflow**-from data acquisition to monitoring-should be treated as a repeatable pipeline, with explicit versioning for data, code, and models.

• **Bias-variance trade-off** explains why a model may under-fit (high bias) or over-fit (high variance). Learning curves and regularization are practical tools to move toward the sweet spot.

• **Overfitting** is a symptom of data leakage, excessive model complexity, or insufficient data. Guard against it with proper validation, early stopping, and regularization.

• **Business-centric evaluation** (lift, ROI, cost-benefit) is more actionable than pure statistical metrics. Align model thresholds with the cost of false positives/negatives.

• **Interpretability** (feature importance, SHAP values) builds stakeholder confidence and uncovers actionable insights beyond the raw prediction.

> ***Bottom line: Mastering the foundations-defining the problem, selecting the right learning paradigm, and structuring a disciplined workflow-sets you up for success long before you write the first line of model code. With these building blocks in place, the subsequent modules on data preparation, algorithmic deep-dives, and production-grade deployment will feel like natural extensions rather than daunting leaps.***

Ready to move on? In **Module 2** we'll dive deeper into **data acquisition and exploratory analysis**, learning how to turn raw logs, databases, and APIs into a clean, reproducible feature set ready for the pipelines we just outlined. Happy modeling!

Data Exploration & Visualization

> ***"If you can't see the data, you can't understand it - and you can't model what you don't understand."***

In Module 1we laid the groundwork: we defined what a predictive modeling problem looks like, discussed the importance of **problem framing**, and introduced the typical ML workflow (data → model → evaluate → deploy). The next logical step is to **get to know the data**. This chapter walks you through a disciplined, reproducible approach to **Data Exploration & Visualization (often abbreviated as EDA)**.

You will learn how to

- **Load** data reliably with **pandas**.
- Perform **sanity checks** (shape, types, missingness, duplicates).
- Generate **descriptive statistics** for both numeric and categorical features.
- Create **visual summaries** using **matplotlib**, **seaborn**, and **plotly**.
- Spot **patterns, outliers, and data-quality issues** before you ever train a model.
- **Formulate hypotheses** about how each feature might relate to the target variable - the bridge between raw data and the modeling decisions you'll make later.

By the end of this chapter you will have a repeatable notebook template that you can apply to any new dataset, and you'll be ready to move confidently into **feature engineering** (Module 3).

Table of Contents

1. [Why EDA Matters - A Quick Recap of Module 1](#why-eda-matters)
2. [Core Concepts]
1. Loading Data with pandas
2. Sanity Checks & Data-Quality Audits
3. Descriptive Statistics - Numbers that Tell a Story
4. Visual Summaries - From Histograms to Interactive Dashboards
5. Detecting Patterns, Outliers, and Anomalies
6. From Insight to Hypothesis - Building a "Feature-Target" Narrative
3. [Practical Application] - A hands-on walkthrough using the **Titanic** and **California Housing** datasets.
4. [Key Takeaways]

---

1. Why EDA Matters - A Quick Recap of Module 1

In Module 1we emphasized that **model performance is bounded by data quality**. A model can only be as good as the information it receives. Skipping or rushing through EDA is akin to building a house on a shaky foundation - the structure may look impressive, but it will crumble under stress.

Key reasons to treat EDA as a **first-class citizen** in the ML pipeline:

Reason	What it prevents	Example
Detecting missing or malformed values	Model crashes or biased predictions	A "Age" column with "? " instead of NaN
Understanding distributional shape	Choosing inappropriate algorithms (e.g., linear regression on heavily skewed	House prices are log-normal

Reason	What it prevents	Example
	data)	
Spotting outliers	Over-fitting or unstable coefficients	A single passenger with fare = $512 in Titanic data
Revealing hidden relationships	Missing opportunities for feature engineering	"Cabin" letter correlates with passenger class
Formulating hypotheses	Guiding feature selection and model choice	"Higher education level leads to higher income"

With that motivation, let's dive into the **core concepts**.

---

2. Core Concepts

2.1 Loading Data with pandas

`pandas` is the de-facto library for tabular data manipulation in Python. Its `read_*` family of functions can ingest CSV, Excel, JSON, SQL, Parquet, and many other formats.

2.1.1 A Minimal, Reproducible Template

```
import pandas as pd
import numpy as np

# 1️⃣ Define a reusable function
def load_dataset(path: str,
                 index_col: str | None = None,
                 parse_dates: list | None = None,
                 encoding: str = "utf-8") -> pd.DataFrame:
    """
    Load a dataset from a CSV (or other delimited file) and perform
    a quick sanity check on the first few rows.
    """
    df = pd.read_csv(
        path,
        index_col=index_col,
        parse_dates=parse_dates,
        encoding=encoding,
        low_memory=False   # prevents dtype guessing warnings on large files
    )
    print(f"✅ Loaded {df.shape[0]:,} rows × {df.shape[1]:,} columns")
    return df
```

Why a function?

- Encapsulates **reproducibility** (same parameters every time).
- Makes it trivial to add **logging** or **caching** later.

2.1.2 Common Pitfalls & How to Avoid Them

Pitfall	Symptom	Fix
Wrong delimiter (e.g., semicolon-separated CSV)	`ParserError: Error tokenizing data`	Use `sep=';'` in `read_csv`.
Hidden BOM (Byte-Order-Mark) in UTF-8 files	First column name looks garbled (`ï»¿id`)	Add `encoding='utf-8-sig'`.
Large files causing memory errors	`MemoryError` on load	Use `dtype` argument to down-cast numeric columns, or read in chunks (`chunksize`).
Dates stored as strings	`dtype: object` for date column	Pass column name to `parse_dates`.

2.2 Sanity Checks & Data-Quality Audits

Before you plot anything, confirm that the dataframe "makes sense". Below is a **checklist** you can run after `load_dataset`.

```
def sanity_check(df: pd.DataFrame, target: str | None = None) -> None:
    """Perform a series of quick sanity checks."""
    # 1️⃣ Shape & basic info
    print("\n🔹 Shape & basic info")
    print(df.shape)
    print(df.dtypes)

    # 2️⃣ Peek at the data
    print("\n🔹 First 5 rows")
    display(df.head())

    # 3️⃣ Missing values
    print("\n🔹 Missing values per column")
    missing = df.isnull().sum()
    print(missing[missing > 0])

    # 4️⃣ Duplicate rows
    dup_cnt = df.duplicated().sum()
    print(f"\n🔹 Duplicate rows: {dup_cnt}")

    # 5️⃣ Unique values (helps spot categorical vs. numeric)
    print("\n🔹 Unique value counts (first 10 columns)")
    for col in df.columns[:10]:
        print(f"{col}: {df[col].nunique()} unique")

    # 6️⃣ Target distribution (if known)
    if target and target in df.columns:
        print(f"\n🔹 Target '{target}' distribution")
        display(df[target].value_counts(normalize=True))
```

2.2.1 What to Look For

Check	Red Flag	Action
Missingness	>⚠ 30⚠ ₀missing in a key	Consider dropping the column or

Check	Red Flag	Action
	predictor	applying imputation (see Module 4).
Duplicate rows	Duplicates > 1% of dataset	Remove with `df.drop_duplicates(inplace=True)`.
Unexpected data types	Numeric column read as `object` (e.g., "$1,200")	Strip symbols, convert with `pd.to_numeric(errors='coerce')`.
Zero variance	Column with a single unique value	Drop - it carries no predictive power.
Target leakage	Target variable appears in a feature column	Remove or rename the feature.

2.3 Descriptive Statistics - Numbers that Tell a Story

Statistical summaries give you a **numerical snapshot** of each variable. For numeric columns, `df.describe()` is a great starting point; for categorical columns, `value_counts()` does the heavy lifting.

2.3.1 Numerical Summaries

```
def numeric_summary(df: pd.DataFrame) -> pd.DataFrame:
    """Return an extended numeric summary."""
    desc = df.describe(percentiles=[.01, .05, .25, .5, .75, .95, .99]).T
    # Add skewness and kurtosis
    desc['skew'] = df.skew()
    desc['kurt'] = df.kurt()
    # Count missing values
    desc['missing'] = df.isnull().sum()
    return desc
```

Key metrics to interpret:

Metric	Interpretation
mean	Central tendency (sensitive to outliers).
median	Robust central tendency.
std	Spread around the mean.
min / max	Range - useful for spotting impossible values (e.g., negative ages).
percentiles (1 %,5 %,95 %,99 %)	Extreme tails - helps decide on clipping or transformation.
skew	Positive skew → right-heavy tail; negative skew → left-heavy tail.
kurt	Heavy-tailed (leptokurtic) vs. light-tailed (platykurtic).

2.3.2 Categorical Summaries

```
def categorical_summary(df: pd.DataFrame, top_n: int = 10) -> pd.DataFrame:
    """Return counts and percentages for each categorical column."""
    summaries = {}
    cat_cols = df.select_dtypes(include=['object', 'category']).columns
    for col in cat_cols:
        vc = df[col].value_counts(dropna=False)
        pct = vc / len(df) * 100
        summaries[col] = pd.DataFrame({
            'count': vc,
            'percent': pct.round(2)
        }).head(top_n)
    return summaries
```

What to watch for:

- **High cardinality** (e.g., thousands of unique IDs) - may need hashing or embedding later.
- **Dominant categories** (e.g., 95 %"Unknown") - consider grouping or dropping.
- **Rare categories** - could be merged into an "Other" bucket.

2.4 Visual Summaries - From Histograms to Interactive Dashboards

Numbers give you "what", but **visuals give you "why"** . We'll cover three complementary libraries:

Library	Strength	Typical Use-Case
matplotlib	Low-level, fully customizable	Publication-ready static figures
seaborn	High-level statistical plots, built on matplotlib	Quick exploratory visualizations
plotly	Interactive, web-ready graphics	Deep dive into patterns, sharing with non-technical stakeholders

Below we present a **progressive visual toolkit** - start simple, then add interactivity when needed.

2.4.1 Histograms & Density Plots (Numeric)

```
import matplotlib.pyplot as plt
import seaborn as sns

def plot_histogram(df: pd.DataFrame, col: str, bins: int = 30):
    plt.figure(figsize=(8,4))
    sns.histplot(df[col].dropna(), bins=bins, kde=True, color='steelblue')
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.show()
```

Why include `kde=True`? The kernel density estimate smooths the histogram, making skewness and multimodality easier to spot.

2.4.2 Box-and-Violin Plots (Outlier Detection)

```
def plot_box_violin(df: pd.DataFrame, col: str, hue: str | None = None):
    plt.figure(figsize=(10,5))
    sns.boxplot(x=hue, y=col, data=df, palette='pastel')
    sns.violinplot(x=hue, y=col, data=df, inner=None, color='0.8')
    plt.title(f'{col} by {hue}')
    plt.show()
```

Interpretation tip: If the box plot shows a **long whisker** or many points beyond the whiskers, you likely have outliers that merit further investigation (capping, transformation, or removal).

2.4.3 Count Plots & Bar Charts (Categorical)

```
def plot_counts(df: pd.DataFrame, col: str, top_n: int = 15):
    vc = df[col].value_counts().nlargest(top_n)
    plt.figure(figsize=(10,5))
    sns.barplot(x=vc.values, y=vc.index, palette='viridis')
    plt.title(f'Top {top_n} categories in {col}')
    plt.xlabel('Count')
    plt.show()
```

If the target is categorical (e.g., `Survived`), a **stacked bar** can reveal class imbalance:

```
def plot_target_balance(df: pd.DataFrame, target: str):
    vc = df[target].value_counts()
    plt.figure(figsize=(6,4))
    sns.barplot(x=vc.index, y=vc.values, palette='coolwarm')
    plt.title('Target Class Distribution')
    plt.xlabel('Class')
    plt.ylabel('Count')
    plt.show()
```

2.4.4 Correlation Heatmap (Numeric Relationships)

```
def plot_corr_heatmap(df: pd.DataFrame, size: int = 10):
    corr = df.corr()
    plt.figure(figsize=(size, size))
    sns.heatmap(corr, cmap='coolwarm', annot=True, fmt=".2f", linewidths=.5)
    plt.title('Correlation Matrix')
    plt.show()
```

Key insight:

Pairs with **|ρ| > 0.7** may be redundant (multicollinearity) - consider dropping or combining them later.

2.4.5 Pairplot / Scatter Matrix (Mixed View)

```
def plot_pairgrid(df: pd.DataFrame, vars: list, hue: str | None = None):
    sns.pairplot(df[vars + ([hue] if hue else [])],
                 hue=hue,
                 diag_kind='kde',
                 plot_kws={'alpha':0.6, 's':40})
    plt.show()
```

When to use: When you have **≤ 6 numeric variables** you can visualise all pairwise relationships at once.

2.4.6 Interactive Plotly Dashboards

```
import plotly.express as px

def plot_interactive_scatter(df: pd.DataFrame,
                             x: str,
                             y: str,
                             color: str | None = None,
                             hover_data: list | None = None):
    fig = px.scatter(df,
                     x=x,
                     y=y,
                     color=color,
                     hover_data=hover_data,
                     title=f'{y} vs {x}',
                     width=800,
                     height=500)
    fig.update_traces(marker=dict(size=8, line=dict(width=0.5, color='DarkSlateGrey')))
    fig.show()
```

Why go interactive?

- **Zoom & pan** to inspect dense clusters.
- **Hover tooltips** reveal exact values (useful for spotting outliers).
- Easy to embed in **Jupyter notebooks**, **Streamlit**, or **dashboards** for stakeholders.

2.5 Detecting Patterns, Outliers, and Anomalies

After you have the visual toolbox, you can systematically hunt for three classes of data issues:

Issue	Visual Cue	Typical Remedy
Missingness pattern	Heatmap of `isnull()` (via `sns.heatmap(df.isnull(), cbar=False)`)	Imputation, flag columns, or drop rows/columns.
Outliers	Points far beyond the box-plot whiskers; isolated points in scatter plots	Winsorize (cap), log-transform, or remove if clearly erroneous.
Multicollinearity	Blocks of high correlation in	Drop one of the correlated

Issue	Visual Cue	Typical Remedy
	heatmap; variance inflation factor (VIF) >□ 5	features, or combine via PCA.
Non-linear relationships	Curved trend in scatter plot; residuals showing systematic pattern	Apply transformations (log, sqrt), or use tree-based models later.
Class imbalance	Skewed target bar chart; ROC curve with low area under the curve (AUC) after quick baseline model	Resampling (SMOTE, undersampling), or use class-weighting.

2.5.1 Example: Missingness Heatmap

```
def plot_missing_heatmap(df: pd.DataFrame):
    plt.figure(figsize=(12,6))
    sns.heatmap(df.isnull(),
                cbar=False,
                yticklabels=False,
                cmap='viridis')
    plt.title('Missing Data Heatmap')
    plt.show()
```

A **clustered missingness** (e.g., many rows missing values in both `Age` and `Cabin`) suggests a **systematic data-collection issue** rather than random noise.

2.5.2 Example: Outlier Detection with Interactive Plotly

```
# Visualize Fare vs Age for Titanic, colour by Survival
plot_interactive_scatter(df=titanic,
                         x='Age',
                         y='Fare',
                         color='Survived',
                         hover_data=['Name', 'Pclass'])
```

Zooming into the upper-right corner reveals a handful of passengers with ***extremely high fares***. You might decide to **cap the fare at the 99th percentile** before

feeding it into a linear model.

2.6 From Insight to Hypothesis - Building a "Feature-Target" Narrative

After you have explored the data, you should **record concrete hypotheses** that you can later test with statistical models. A hypothesis is a **testable statement** linking a predictor (or set of predictors) to the target.

2.6.1 Template for Formulating Hypotheses

Component	Guiding Question	Example (Titanic)
Feature(s)	Which column(s) might influence the outcome?	`Sex`, `Pclass`, `Fare`, `Age`
Direction	Do you expect a positive, negative, or non-linear effect?	Women (Sex = female) → higher survival probability (positive).
Mechanism	Why would this relationship exist?	Historically, women and children were given priority on lifeboats.
Testable Form	"Feature X is associated with higher/lower target Y."	***Hypothesis*: *Passengers in 1st class have a higher probability of survival than those in 3rd class.***

2.6.2 Prioritizing Hypotheses

1. **High-Impact** - Features that show strong correlation or visual separation.
2. **Low-Cost** - Features that require minimal preprocessing.
3. **Domain-Driven** - Variables that make sense given the business problem.

Write them down in a **"hypothesis log"** (a simple Markdown table works well):

```
| # | Feature(s) | Expected Effect | Rationale | Test Method |
|---|------------|-----------------|-----------|-------------|
| 1 | Sex        | Positive (female > male) | Lifeboat priority | Logistic regression coefficient |
| 2 | Pclass     | Positive (1 > 2 > 3) | Cabin location & access | ANOVA on survival rates |
| 3 | Age        | Non-linear (young & old higher survival) | "Children first"; elderly assisted |
Decision-tree split analysis |
| 4 | Fare       | Positive | Higher fare → better cabin → closer to deck | Correlation & ROC curve |
```

These hypotheses will guide the **feature-engineering** decisions you make in the next module and will give you a **baseline** to compare against more sophisticated models later.

---

3. Practical Application

In this section we walk through a **complete EDA notebook** from start to finish, using two classic datasets:

Dataset	Reason for selection
Titanic (binary classification)	Small, well-known, mix of numeric & categorical, clear target (`Survived`).
California Housing (regression)	Larger, real-world numeric dataset with spatial component (`median_house_value`).

Both are available directly from the **scikit-learn** repository, so you can run the code without any external downloads.

> ***Tip - Copy the code snippets into a Jupyter notebook (or a Colab notebook) and execute them sequentially. The comments guide you on what to look for after each block.***

3.1 Setup - Import Packages & Define Helper Functions

```
# Core libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

# Plotting style
sns.set_style('whitegrid')
plt.rcParams['figure.figsize'] = (10, 6)

# Helper functions (as defined earlier)
def load_dataset(path, **kwargs):
    df = pd.read_csv(path, low_memory=False, **kwargs)
    print(f"✅ Loaded {df.shape[0]:,} rows × {df.shape[1]:,} columns")
    return df

def sanity_check(df, target=None):
    # (Implementation from Section 2.2)
    ...

def numeric_summary(df):
    # (Implementation from Section 2.3.1)
    ...

def categorical_summary(df, top_n=10):
    # (Implementation from Section 2.3.2)
    ...

def plot_histogram(df, col, bins=30):
    # (Implementation from Section 2.4.1)
    ...

def plot_missing_heatmap(df):
    # (Implementation from Section 2.5.1)
    ...

def plot_interactive_scatter(df, x, y, color=None, hover_data=None):
    # (Implementation from Section 2.4.6)
    ...
```

3.2 Titanic - A Binary Classification Example

3.2.1 Load the data

```
titanic_url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
titanic = load_dataset(titanic_url)
sanity_check(titanic, target='Survived')
```

Typical output (truncated):

```
 Loaded 891 rows × 12 columns
 Shape & basic info
(891, 12)
PassengerId               int64
Survived                  int64
...
Ticket                   object
Cabin                    object
Embarked                 object
...
 Missing values per column
Age          177
Cabin        687
Embarked       2
...
 Duplicate rows: 0
...
 Target 'Survived' distribution
   0    0.549
   1    0.451
```

Takeaway - ~20 %missing `Age`, ~77 %missing `Cabin` (likely drop or extract deck letter), small missing `Embarked`.

3.2.2 Numeric summary

```
num_summary = numeric_summary(titanic.select_dtypes(include='number'))
display(num_summary)
```

Key observations:

Feature	mean	median	std	skew	min	max
Age	29.7	28	14.5	0.48 (right-skew)	0.42	80
Fare	32.2	14.45	49.7	3.33 (highly right-skew)	0	512.33

Action - Consider **log-transforming `Fare`** and **imputing `Age`** (maybe with median per `Pclass`).

3.2.3 Categorical summary

```
cat_summary = categorical_summary(titanic)
for col, df_sum in cat_summary.items():
    print(f"\n--- {col} ---")
    display(df_sum)
```

Sample output for `Embarked`:

count	percent
S	72.4
C	19.4
Q	8.2
NaN	0.2

Action - Replace missing `Embarked` with the mode (`S`).

3.2.4 Visual Exploration

Histogram of Age (with KDE)

```
plot_histogram(titanic, 'Age')
```

Interpretation: The age distribution is right-skewed with a noticeable bump around 0-5 (children).

Box-Violin of Fare by Survival

```
plot_box_violin(titanic, 'Fare', hue='Survived')
```

You'll see a **long right tail** for survivors - a clue that high-fare passengers had a better chance.

Missingness Heatmap

```
plot_missing_heatmap(titanic)
```

A block of red (missing) appears for `Cabin` and `Age` together, hinting that **older, lower-class passengers** often lack cabin info.

Interactive Scatter: Age vs Fare, colour by Survival

```
plot_interactive_scatter(titanic,
                         x='Age',
                         y='Fare',
                         color='Survived',
                         hover_data=['Name', 'Pclass'])
```

Zooming reveals a **cluster of young, low-fare passengers** with high survival - likely "women & children first".

3.2.5 Formulating Hypotheses

#	Feature(s)	Expected Effect	Rationale	Test Method

#	Feature(s)	Expected Effect	Rationale	Test Method
1	Sex	Positive (female > male)	Lifeboat priority	Logistic regression coefficient
2	Pclass	Positive (1 > 2 > 3)	Proximity to deck & lifeboats	Chi-square test on survival
3	Age	Non-linear (children & elderly > middle-aged)	"Children first" + assisted elders	Decision-tree split analysis
4	Fare (log)	Positive	Higher fare → better cabin location	Correlation & ROC AUC

These will be revisited when we **engineer features** (e.g., create `IsFemale`, `IsChild`, `LogFare`).

3.3 California Housing - A Regression Example

3.3.1 Load the data

```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing(as_frame=True)
df_h = housing.frame   # Already a pandas DataFrame
sanity_check(df_h, target='MedHouseVal')
```

Typical output:

```
 Loaded 20,640 rows × 9 columns
...
 Missing values per column
(no missing values!)
...
 Target 'MedHouseVal' distribution
0    0.004
1    0.005
...
```

Takeaway - No missing data (great!), but we have **high cardinality** in `OceanProximity` (categorical).

3.3.2 Numerical summary

```
num_summary_h = numeric_summary(df_h.select_dtypes(include='number'))
display(num_summary_h)
```

Key stats (excerpt):

Feature	mean	median	std	skew	min	max
MedInc	3.87	3.53	1.90	0.94 (right)	0.52	15.00
HouseAge	28.64	29.0	12.58	0.20	1.0	52.0
AveRooms	5.42	5.22	2.50	0.31	0.70	18.10
MedHouseVal	2.00	1.80	0.80	0.60	0.15	5.00

Action - `MedInc` (median income) is **right-skewed** - log-transform may improve linearity.

3.3.3 Categorical summary

```
cat_summary_h = categorical_summary(df_h)
display(cat_summary_h['OceanProximity'])
```

Result:

count	percent
<1H OCEAN	61.6
INLAND	26.4

count	percent
NEAR OCEAN	8.7
NEAR BAY	2.9
ISLAND	0.4

Action - `ISLAND` is very rare; combine it with `<1H OCEAN` or drop it.

3.3.4 Visual Exploration

Correlation heatmap

```
plot_corr_heatmap(df_h, size=8)
```

Observations:

- `MedInc` has the strongest positive correlation with `MedHouseVal` (≈ 0.69).
- `AveOccup` is negatively correlated (≈ -0.25).

Pairgrid of top 4 numeric variables

```
top_vars = ['MedInc', 'HouseAge', 'AveRooms', 'MedHouseVal']
plot_pairgrid(df_h, vars=top_vars)
```

The scatter of `MedInc` vs `MedHouseVal` shows a **clear upward trend** but with a **funnel shape** (heteroscedasticity).

Interactive scatter: Median Income vs House Value, colour by Ocean Proximity

```
plot_interactive_scatter(df_h,
                         x='MedInc',
                         y='MedHouseVal',
                         color='OceanProximity',
                         hover_data=['Population', 'AveRooms'])
```

Zooming reveals that **near-bay neighborhoods** achieve higher house values at a given income - a hint that **location** (proximity) adds predictive power beyond income alone.

3.3.5 Formulating Hypotheses

#	Feature(s)	Expected Effect	Rationale	Test Method
1	MedInc (log)	Positive	Income drives purchasing power	Linear regression on log-transformed income
2	HouseAge	Negative (older houses cheaper)	Depreciation effect	Polynomial regression (quadratic)
3	AveRooms	Positive up to a point, then plateau	Larger homes → higher price, but diminishing returns	Spline regression
4	OceanProximity	Categorical effect (Near Bay > <1H Ocean > Inland)	Coastal desirability	One-hot encoding + ANOVA
5	Population	Positive (denser areas → higher demand)	Urban premium	Correlation + multiple regression

These become the **blueprint** for the feature-engineering tasks we'll perform next (e.g., **log-transform**, **polynomial features**, **one-hot encoding**).

3.4 Saving Your EDA Artifacts

A disciplined workflow stores the **outputs** (plots, tables, hypothesis log) so you can revisit them later or share with teammates.

```
import os, json

output_dir = "eda_artifacts"
os.makedirs(output_dir, exist_ok=True)

# Save numeric & categorical summaries as CSV
num_summary.to_csv(f"{output_dir}/numeric_summary.csv")
for col, df_sum in cat_summary.items():
    df_sum.to_csv(f"{output_dir}/cat_summary_{col}.csv")

# Save hypothesis log as JSON
hypotheses = [
    {"id":1, "features":["Sex"], "effect":"positive", "rationale":"lifeboat priority"},
    {"id":2, "features":["Pclass"], "effect":"positive", "rationale":"deck proximity"},
    # ...
]
with open(f"{output_dir}/hypotheses.json", "w") as f:
    json.dump(hypotheses, f, indent=2)
```

Now you have a **self-contained EDA package** that can be version-controlled (Git) and referenced when you move to modeling.

---

4. Key Takeaways

- **EDA is not optional** - it is the discovery phase that informs every later decision (feature engineering, model selection, evaluation).
- **Load data with a reusable function**; check shape, types, missingness, duplicates, and unique counts right after loading.
- **Descriptive statistics** (mean, median, std, skew, kurtosis) give you a numerical baseline; **value_counts** reveal categorical distributions.
- **Visual tools** - start with static **matplotlib/seaborn** for quick sketches, then use **plotly** for interactive deep-dives.
- **Detect data-quality issues**: missing patterns, outliers, multicollinearity,

non-linear trends, and class imbalance.

- **Formulate testable hypotheses** linking features to the target; log them in a structured table to guide later modeling.
- **Save your EDA artifacts** (plots, tables, hypothesis log) for reproducibility and collaboration.

> ***Bottom line: By the time you finish this chapter, you should be able to open any new dataset, run a handful of scripts, and walk away with a clear story about what the data looks like, where the problems lie, and which features are most promising. Those insights become the foundation for the feature- engineering and modeling work that follows in Modules 3-10.***

End of Chapter 2 - "Data Exploration & Visualization".

Data Preprocessing & Feature Engineering

> ***"Garbage-in, garbage-out" is a cliché for a reason. The quality of the data you feed a model determines how far it can go.***

In **Module 3** we move from "what does the data look like?" (Module 2) to **how we shape it** so that any algorithm-linear, tree-based, or deep-learning-can learn the right patterns. This chapter is the bridge between raw, noisy observations and the clean, information-rich matrix that powers predictive models.

Introduction

When you opened the CSV in the previous module you probably noticed:

- missing entries (`NaN`, empty strings, "?", "N/A")
- columns with the wrong type (numeric values stored as text)
- duplicated rows or partially duplicated records
- categorical fields that need to be turned into numbers
- features that are on wildly different scales

All of these quirks are **symptoms of the data-preprocessing problem**. Ignoring them can cause:

Symptom	Consequence	Example
Missing values	Algorithms either crash or impute poorly, biasing predictions	A regression model trained on a house-price dataset where 30% of `LotArea` is missing will underestimate large lots.
Wrong dtypes	String operations on numbers, numeric operations on strings	`int('123')` works, but `int('abc')` raises an error; scikit-learn's `StandardScaler` refuses non-numeric input.
Duplicates	Inflated weight of repeated observations → over-fitting	Two identical rows for the same customer double-count their behavior.
Unencoded categoricals	Models cannot interpret text; distance metrics become meaningless	K-Nearest Neighbors sees "Red" and "Blue" as completely unrelated strings.
Unscaled features	Gradient-based optimizers converge slowly or not at all; distance-based models become biased	In a logistic regression, a feature ranging 0-1 and another ranging 0-1,000,000 dominate the loss landscape.

The **goal** of this chapter is to give you a **practical, reproducible workflow** that:

1. **Cleans** the raw data (missingness, types, duplicates).
2. **Transforms** categorical variables into numeric representations that respect the algorithm's assumptions.
3. **Scales** numeric features so that they live in a comparable range.

4. **Creates** new, more informative features that capture domain knowledge or hidden relationships.

By the end, you will be comfortable building **scikit-learn pipelines** (or their equivalents in other libraries) that encapsulate every step, making your preprocessing **transparent, reusable, and safe from data-leakage**.

Core Concepts

1. Handling Missing Values

Strategy	When to Use	How it Works	Pros	Cons
Drop rows / columns	Very few missing entries, or a column is >□ 70□ %missing	`df.dropna()` or `df.dropna(axis=1, thresh=...)`	Simple, no imputation bias	Loss of data, may reduce sample size dramatically
Simple imputation (mean/ median/ mode)	Numerical features with roughly symmetric or skewed distributions; categorical with few levels	`SimpleImputer(strategy='mean')` or `'most_frequent'`	Fast, works for many models	Ignores relationships, can shrink variance
Constant imputation	Missingness itself may be informative (e.g., "no previous purchase")	Fill with a sentinel like `-999` or `"Missing"`	Preserves missingness signal	May create artificial outliers, not ideal for distance-based models
Model-based imputation	Complex patterns of missingness; you have enough data to learn them	`IterativeImputer` (MICE), `KNNImputer`, or a custom regression model	Captures multivariate relationships	Computationally expensive, risk of over-fitting the imputer

Strategy	When to Use	How it Works	Pros	Cons
Indicator variables	When you want to keep the missingness flag alongside the imputed value	`add_indicator=True` in `SimpleImputer`	Allows downstream model to learn "missingness is predictive"	Increases dimensionality

Practical tip: Always **inspect the missingness mechanism** (`Missing Completely at Random`, `Missing at Random`, `Missing Not at Random`). A quick `df.isnull().mean().sort_values(ascending=False)` can reveal columns that need special attention.

---

2. Correcting Data Types

- **String → Numeric** - Use `pd.to_numeric(errors='coerce')`. Non-convertible entries become `NaN`, then impute or drop.
- **Numeric → Categorical** - If a numeric column actually encodes categories (e.g., `0/1/2` for "low/medium/high"), cast with `astype('category')`.
- **Date/Time** - Parse with `pd.to_datetime()`. Extract useful components (`year`, `month`, `dayofweek`, `is_month_end`, etc.) as separate features.

```
# Example: cleaning a mixed-type column
df['age'] = pd.to_numeric(df['age'], errors='coerce')
df['signup_date'] = pd.to_datetime(df['signup_date'], errors='coerce')
df['signup_month'] = df['signup_date'].dt.month
df['is_weekend_signup'] = df['signup_date'].dt.weekday.isin([5, 6]).astype(int)
```

---

3. Removing Duplicates

```
# Detect exact duplicate rows
duplicate_mask = df.duplicated()
print(f"Found {duplicate_mask.sum()} duplicate rows.")
df = df[~duplicate_mask]          # keep only the first occurrence
```

If duplicates are only partial (e.g., same `customer_id` but different transaction dates) you may need **group-by aggregation** or **deduplication rules** (keep the latest, keep the row with the most complete information, etc.).

---

4. Encoding Categorical Variables

Encoding	Typical Use-Case	Mechanics	When Not to Use
One-Hot (Dummy) Encoding	Nominal variables with ≤ 10-15levels; linear models, tree ensembles	Create a binary column for each category (`pd.get_dummies` or `OneHotEncoder`)	High-cardinality features → dimensionality explosion
Ordinal Encoding	Ordinal variables (e.g., "Low", "Medium", "High")	Map each level to an integer preserving order	Non-ordinal categories - algorithm may infer false ordering
Target / Mean Encoding	High-cardinality nominal variables; gradient boosting, linear models	Replace category with the mean of the target (or a smoothed version) computed on training data	Risk of leakage - must be done inside cross-validation or via `TargetEncoder` from `category_encoders`
Frequency / Count Encoding	When frequency of a category itself carries information	Replace each category by its count or frequency in the training set	May not be sufficient alone, often combined with other encodings
Embedding (Neural Nets)	Very high cardinality, deep learning pipelines	Learn low-dimensional dense vectors during model training	Requires a neural architecture; not directly usable with classic ML libraries

Implementation tip: Use **scikit-learn's `ColumnTransformer`** to apply different encoders to different columns in a single pipeline.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from category_encoders import TargetEncoder

categorical_nominal = ['city', 'product']
categorical_ordinal = ['education_level']
high_cardinality   = ['customer_id']

preprocess = ColumnTransformer([
    ('onehot', OneHotEncoder(handle_unknown='ignore'), categorical_nominal),
    ('ordinal', OrdinalEncoder(categories=[['High School', 'Bachelors', 'Masters', 'PhD']]),
categorical_ordinal),
    ('target', TargetEncoder(smoothing=10), high_cardinality),
    # numeric pipeline will be added later
])
```

---

5. Scaling & Normalizing Features

Technique	Algorithm Compatibility	Formula	Typical Use
Standardization (Z-score)	Linear models, SVM, k-NN, Neural Nets	`z = (x - μ) / σ`	When data is roughly Gaussian; removes units
Min-Max Scaling	Neural Nets, k-means, tree ensembles (optional)	`x' = (x - min) / (max - min)`	When you need values in \[0,□ 1\](e.g., image pixels)
Robust Scaling	Presence of outliers; k-NN, linear models	`x' = (x - median) / IQR`	Outlier-heavy data (e.g., income)

| **Max-Abs Scaling** | Sparse data; linear models with L1 regularization | `x' = x / max(\|x\|)` | Keeps sign, works with sparse matrices |

| **Log / Power Transform** | Skewed distributions; tree ensembles less sensitive, but

linear models benefit | `x' = log(x + 1)` or `x' = x**0.5` | Positive-only variables with long tails (e.g., price) |

Pipeline example:

```
numeric_features = ['age', 'salary', 'tenure']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

preprocess = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    # categorical pipelines added earlier
])
```

---

6. Feature Engineering - Creating New Informative Variables

1. **Mathematical Transformations**

Log, ***square root***, ***Box-Cox***, ***Yeo-Johnson*** - reduce skewness, compress large ranges.

2. **Interaction Features**

Multiply or concatenate two or more variables to capture synergy (`age * tenure`, `city + '_' + product`).

3. **Aggregations (Group-by Features)**

For transactional data, compute per-entity statistics:

- `customer_total_spent = sum(amount) per customer`
- `avg_days_between_purchases`

4. **Temporal Features**

- `days_since_last_event` = `today - last_event_date`
- `rolling_window_mean` (e.g., 7-day moving average of sales).

5. **Domain-Specific Logic**

In a **housing** problem, combine `LotArea` and `OverallQual` into a "size-quality index". In **churn** modeling, create a "usage decline" flag from month-over-month activity.

6. **Automated Feature Generation**

Tools like **Featuretools** (Deep Feature Synthesis) can automatically generate relational features from multi-table data.

Example - Engineering a "price per square foot" feature:

```
df['price_per_sqft'] = df['SalePrice'] / df['GrLivArea']
df['age_of_house']   = 2024 - df['YearBuilt']
df['is_recently_renovated'] = (df['YearRemodAdd'] > 2000).astype(int)
```

---

7. Guarding Against Data Leakage

Data leakage occurs when information from the **test/validation set** seeps into the training pipeline, inflating performance estimates.

- **Never fit imputation or scaling on the whole dataset** - fit on the training split only, then transform validation/test.
- **Target encoding must be computed inside cross-validation folds**; `category_encoders.TargetEncoder` offers a `cv` argument, or you can write a custom transformer that uses `KFold`.
- **Temporal data**: When creating lag features, ensure that only ***past*** information is used for a given prediction point.

Pipeline ensures leakage protection because each transformer's `fit` runs only on the data that reaches it (usually the training split), and `transform` is later applied to hold-out data.

Practical Application

We will walk through a **complete end-to-end workflow** using the **Ames Housing** dataset (a well-known regression benchmark). The steps are deliberately modular so you can copy-paste them into a notebook and adapt to any tabular problem.

> ***Why Ames?***
>
> **** 79 explanatory variables (mix of numeric, categorical, dates)***
>
> **** Real-world missingness patterns***
>
> **** A target (`SalePrice`) that benefits from log-transform and engineered features***

1. Setup & Load Data

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler, FunctionTransformer
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.metrics import mean_absolute_error, r2_score
from category_encoders import TargetEncoder
import warnings, matplotlib.pyplot as plt
warnings.filterwarnings('ignore')

# Load data (download from https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data)
df = pd.read_csv('ames_train.csv')
df.head()
```

2. Train-Test Split

```
X = df.drop('SalePrice', axis=1)
y = df['SalePrice']

# Use a stratified split on the log of the target to preserve price distribution
y_log = np.log1p(y)
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y_log, test_size=0.2, random_state=42, stratify=pd.qcut(y_log, q=10, duplicates='drop')
)
```

3. Identify Column Types

```
# Helper to detect types
numeric_cols = X_train.select_dtypes(include=['int64', 'float64']).columns.tolist()
categorical_cols = X_train.select_dtypes(include=['object']).columns.tolist()

# Some numeric columns are really categories (e.g., MSSubClass)
maybe_categorical = ['MSSubClass', 'MoSold', 'YrSold']
for col in maybe_categorical:
    if col in numeric_cols:
        numeric_cols.remove(col)
        categorical_cols.append(col)

print(f"Numeric: {len(numeric_cols)} columns")
print(f"Categorical: {len(categorical_cols)} columns")
```

4. Build Sub-Pipelines

```
# ---- Numeric pipeline ----------------------------------------------------
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# ---- Categorical pipeline (mix of encoders) ----------------------------
# Split nominal vs. ordinal vs. high-cardinality
nominal = [c for c in categorical_cols if X_train[c].nunique() <= 10]
ordinal = ['OverallQual', 'OverallCond', 'ExterQual', 'ExterCond',
           'BsmtQual', 'BsmtCond', 'HeatingQC', 'KitchenQual',
           'FireplaceQu', 'GarageQual', 'GarageCond', 'PoolQC']
high_card = [c for c in categorical_cols if X_train[c].nunique() > 20]

categorical_transformer = ColumnTransformer(transformers=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'), nominal),
    ('ordinal', OrdinalEncoder(handle_unknown='use_encoded_value',
                               unknown_value=-1), ordinal),
    ('target', TargetEncoder(smoothing=10), high_card)
], remainder='drop')
```

5. Feature-Engineering Functions

```
def add_engineered_features(df):
    """Create domain-specific features for Ames."""
    df = df.copy()
    # Age of the house at sale time
    df['house_age'] = 2024 - df['YearBuilt']
    # Time since remodel
    df['years_since_remodel'] = 2024 - df['YearRemodAdd']
    # Total square footage (above + basement)
    df['total_sqft'] = df['GrLivArea'] + df['TotalBsmtSF']
    # Price per sqft (only for training target, but we keep placeholder)
    df['price_per_sqft'] = np.nan   # will be filled later for training set only
    # Log transform of skewed numeric variables
    skewed = ['LotFrontage', 'MasVnrArea', 'GarageArea', 'TotalBsmtSF']
    for col in skewed:
        df[col] = np.log1p(df[col])
    return df

# Wrap the function as a transformer
feature_engineer = FunctionTransformer(add_engineered_features)
```

6. Assemble the Full Pipeline

```
preprocess = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_cols),
        ('cat', categorical_transformer, categorical_cols)
    ], remainder='passthrough'   # keep engineered columns that are not in the lists
)

full_pipe = Pipeline(steps=[
    ('feature_engineering', feature_engineer),
    ('preprocess', preprocess),
    # Model placeholder - we'll use XGBRegressor later
])
```

7. Fit the Pipeline & Inspect Shapes

```
X_train_processed = full_pipe.fit_transform(X_train, y_train)
print(f"Processed shape: {X_train_processed.shape}")
```

You should see a **sparse matrix** (if many one-hot columns) with **hundreds of columns** - a manageable size for gradient-boosting models.

8. Train a Model (XGBoost)

```
from xgboost import XGBRegressor

model = XGBRegressor(
    n_estimators=500,
    learning_rate=0.05,
    max_depth=4,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.0,
    reg_lambda=1.0,
    objective='reg:squarederror',
    random_state=42,
    n_jobs=-1
)

# Combine preprocessing and model in a single pipeline for clean evaluation
model_pipe = Pipeline(steps=[
    ('preprocess', full_pipe),
    ('model', model)
])

# Cross-validation on training set
cv = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(model_pipe, X_train, y_train,
                            scoring='neg_root_mean_squared_error',
                            cv=cv)
print(f"CV RMSE: { -cv_scores.mean():.4f } ± { cv_scores.std():.4f }")
```

9. Evaluate on Hold-Out Set

```
model_pipe.fit(X_train, y_train)   # Fit on full training data
preds_log = model_pipe.predict(X_valid)
preds = np.expm1(preds_log)        # Revert log transform

mae = mean_absolute_error(np.expm1(y_valid), preds)
r2 = r2_score(np.expm1(y_valid), preds)

print(f"Hold-out MAE: ${mae:,.0f}")
print(f"Hold-out R²: {r2:.4f}")
```

10. Feature-Importance Insight

```
# Get feature names after ColumnTransformer processing
def get_feature_names(ct):
    # Helper for scikit-learn < 1.0
    output = []
    for name, transformer, cols in ct.transformers_:
        if name == 'remainder':
            continue
        if hasattr(transformer, 'get_feature_names_out'):
            names = transformer.get_feature_names_out(cols)
        else:
            names = cols
        output.extend([f"{name}__{n}" for n in names])
    return output

feat_names = get_feature_names(full_pipe.named_steps['preprocess'])
importances = model.feature_importances_
top_idx = np.argsort(importances)[-15:][::-1]

print("Top 15 features:")
for i in top_idx:
    print(f"{feat_names[i]:<40} {importances[i]:.4f}")
```

The output will highlight **engineered features** (`house_age`, `total_sqft`, `OverallQual`) alongside powerful one-hot categories (`Neighborhood_...`). This confirms that **feature engineering added predictive signal** beyond raw columns.

11. Export the Pipeline for Production

```
import joblib
joblib.dump(model_pipe, 'ames_price_model.pkl')
```

The saved object contains **all preprocessing steps** (imputation, encoding, scaling, engineered features) and the trained XGBoost model. Deploying it in a Flask API or a batch scoring job guarantees the same transformations are applied to new houses.

Key Takeaways

Concept	Why It Matters	Quick Action
Missing-value strategies	Prevent crashes and bias; preserve information in "missingness"	Start with `SimpleImputer`; graduate to `IterativeImputer` or `KNNImputer` only if needed.
Data-type correction	Guarantees that numeric math works and categorical logic applies	Use `pd.to_numeric`, `astype('category')`, and `pd.to_datetime` early in the pipeline.
Duplicate handling	Avoid over-counting and inflated model confidence	`df.duplicated()` → drop, or aggregate with domain rules.
Encoding	Transforms text into numbers that respect algorithm assumptions	One-hot for low-cardinality; ordinal for ordered categories; target/mean encoding for high-cardinality (always inside CV).
Scaling	Aligns feature magnitudes, speeds up gradient-based learning, stabilizes distance calculations	`StandardScaler` for Gaussian-like data; `RobustScaler` when outliers dominate.
Feature engineering	Turns raw observations into higher-level signals that often dominate model performance	Log transforms, interaction terms, time-based aggregates, domain-specific indices.
Pipeline & leakage protection	Guarantees reproducibility and honest validation	Wrap every step in `Pipeline`/`ColumnTransformer`; never fit on test data.
Evaluation	Checks that engineered features truly help, not just over-fit	Use cross-validation, hold-out set, and domain-relevant metrics (MAE, RMSE, R^2).

Final Checklist for Every New Tabular Project

1. **Load & inspect** - `df.info()`, `df.isnull().mean()`, `df.describe(include='all')`.
2. **Fix dtypes** - numeric ↔ categorical ↔ datetime.
3. **Remove/aggregate duplicates** - decide on a rule (first, last, most complete).

4. **Identify missing-value pattern** - decide on drop vs. impute vs. flag.
5. **Separate column groups** - numeric, nominal, ordinal, high-cardinality, dates.
6. **Design encoding strategy** - one-hot, ordinal, target, frequency.
7. **Choose scaling** - standard, min-max, robust, log/Box-Cox as needed.
8. **Engineer features** - transformations, interactions, aggregates, domain logic.
9. **Wrap everything in a pipeline** - `ColumnTransformer` + `Pipeline`.
10. **Validate** - cross-validation + hold-out, watch for leakage.
11. **Inspect feature importance** - ensure engineered features contribute.
12. **Persist the pipeline** - `joblib.dump` for reproducible production scoring.

By mastering these steps you will turn **raw, messy tables** into **clean, information-dense matrices** that let any algorithm-linear, tree-based, or neural-reach its full predictive potential. The next module will show you **how to select, tune, and evaluate models** on top of the robust data foundation you've just built. Happy engineering!

Supervised Learning - Linear Models

> ***"A model that is simple enough to be understood, yet powerful enough to be useful, is the holy grail of predictive analytics."***

In the previous modules we learned **how to see** and **how to clean** our data. Now we turn our attention to **how to turn that data into a prediction**. Linear models are the entry point to supervised learning-they are mathematically transparent, fast to train, and form the foundation for many more sophisticated techniques. This chapter walks you through the entire workflow: from fitting a plain-vanilla linear regression, to checking its statistical assumptions, to taming over-fitting with regularisation, and finally to extending the same ideas to binary classification with logistic regression. Throughout we will use **Scikit-learn pipelines** to keep the code tidy and reproducible.

Table of Contents

1. [Why Linear Models?](#why-linear-models)
2. [Linear Regression Refresher](#linear-regression-refresher)
- 2.1 The Ordinary Least Squares (OLS) objective
- 2.2 Estimating coefficients with `LinearRegression`
3. [Diagnosing OLS Assumptions](#diagnosing-ols-assumptions)
- 3.1 Linearity & functional form
- 3.2 Homoscedasticity (constant variance)
- 3.3 Independence & autocorrelation
- 3.4 Normality of residuals
- 3.5 Multicollinearity
4. [Regularisation: Ridge, Lasso, Elastic Net](#regularisation)
- 4.1 Bias-variance trade-off recap
- 4.2 Ridge (L2)
- 4.3 Lasso (L1)
- 4.4 Elastic Net (L1 + L2)
- 4.5 Hyper-parameter tuning with `GridSearchCV`
5. [Logistic Regression for Binary Classification](#logistic-regression)
- 5.1 From linear to logistic link function
- 5.2 Interpreting odds ratios
- 5.3 Regularised logistic regression
6. [Putting It All Together: Scikit-learn Pipelines](#pipelines)
- 6.1 Pre-processing steps (imputation, scaling, encoding)
- 6.2 A single end-to-end pipeline for regression
- 6.3 A single end-to-end pipeline for classification
7. [Practical Walk-throughs (Code Boxes)](#practical-examples)
- 7.1 Housing price prediction (Boston dataset) - OLS + diagnostics

- 7.2 Regularised regression on a high-dimensional synthetic set
- 7.3 Titanic survival prediction - logistic regression with pipelines

8. [Key Takeaways](#key-takeaways)

---

1. Why Linear Models?

Pros	Cons
• Easy to interpret - each coefficient tells you the ***marginal*** effect of a predictor.	• Assumes a linear relationship; real-world data can be non-linear.
• Computationally cheap - training time is $O(np^2)$ for n samples and p features.	• Sensitive to outliers and multicollinearity.
• Form the building blocks for more complex algorithms (e.g., GLMs, kernel methods).	• May under-fit when the true signal is highly non-linear.
• Well-studied statistical theory gives us confidence intervals, hypothesis tests, etc.	• Need to verify assumptions for reliable inference.

Because of these traits, linear models are the **default starting point** for almost any supervised learning problem. If they work well, you've already built a solid baseline; if they don't, the diagnostics will guide you toward the next steps (feature engineering, non-linear models, or ensemble methods).

---

2. Linear Regression Refresher

2.1 The Ordinary Least Squares (OLS) Objective

Given a matrix of predictors **X** ∈ ℝ^{n×p} and a response vector **y** ∈ ℝ^n, the OLS solution finds coefficients **β** that minimise the sum of squared residuals:

$$\hat{\beta} = \arg\min_{\beta} \; \| y - X\beta \|_2^2$$

The closed-form solution (when **X** has full column rank) is:

$$\hat{\beta} = (X^\top X)^{-1} X^\top y$$

In practice we rarely compute the inverse directly; numerical linear-algebra libraries use QR decomposition or singular value decomposition (SVD) for stability.

2.2 Estimating Coefficients with Scikit-learn

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Load the classic Boston housing dataset (deprecated in sklearn 1.2+; use fetch_openml)
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = pd.Series(boston.target, name='MEDV')          # Median house value ($1000s)

# Train-test split (80/20)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Fit OLS
ols = LinearRegression()
ols.fit(X_train, y_train)

# Coefficients
coef_df = pd.DataFrame({
    'feature': X.columns,
    'coef': ols.coef_
}).sort_values('coef', key=abs, ascending=False)

print(coef_df)
```

Output (excerpt)

feature	coef
LSTAT	-0.96
RM	3.80
PTRATIO	-1.05
DIS	-1.48
...	...

Interpretation: Holding everything else constant, each additional room (`RM`) raises the predicted price by about **$3,800** (since the target is in $1000s).

> ***Note:** In the next sections we will validate whether these coefficient estimates are trustworthy.*

---

3. Diagnosing OLS Assumptions

Linear regression is not just a black-box optimizer; its validity rests on a set of statistical assumptions. Violations can lead to biased coefficients, unreliable confidence intervals, or poor predictive performance.

3.1 Linearity & Functional Form

What to check: Plot the observed response against each predictor (or a transformed version) and look for systematic curvature.

Tool: `seaborn` pairplots, `statsmodels.graphics.regressionplots.plot_fit`.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Simple scatter plot of MEDV vs. RM (rooms)
sns.scatterplot(x='RM', y='MEDV', data=pd.concat([X_train, y_train], axis=1))
plt.title('MEDV vs. RM')
plt.show()
```

If the relationship looks curved, consider adding polynomial terms or applying a non-linear transformation (log, sqrt).

3.2 Homoscedasticity (Constant Variance)

What to check: Residuals should have roughly equal spread across the range of fitted values.

Tool: Residual vs. fitted plot; Breusch-Pagan test (via `statsmodels`).

```
import statsmodels.api as sm

# Get predictions
y_pred = ols.predict(X_train)

# Residuals
residuals = y_train - y_pred

# Plot
sns.scatterplot(x=y_pred, y=residuals)
plt.axhline(0, color='red', ls='--')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Fitted')
plt.show()
```

A funnel shape (spread increasing with fitted value) signals heteroscedasticity. Remedies include:

- **Weighted Least Squares (WLS)**
- **Log-transforming the target**
- **Robust regression** (`HuberRegressor`)

3.3 Independence & Autocorrelation

If data points are ordered in time or space, residuals may be correlated (violating the independence assumption).

Tool: Durbin-Watson statistic (`statsmodels.stats.stattools.durbin_watson`).

```
from statsmodels.stats.stattools import durbin_watson
dw = durbin_watson(residuals)
print(f'Durbin-Watson: {dw:.2f}')
```

Values near 2 → no autocorrelation; <1.5 (positive) or >2.5 (negative) suggest a problem.

If autocorrelation exists, consider time-series models (ARIMA) or adding lagged features.

3.4 Normality of Residuals

Why it matters: Inference (p-values, confidence intervals) assumes residuals are normally distributed.

Tool: Q-Q plot, Shapiro-Wilk test.

```
import scipy.stats as stats

# Q-Q plot
sm.qqplot(residuals, line='45')
plt.title('Q-Q plot of residuals')
plt.show()

# Shapiro-Wilk
stat, p = stats.shapiro(residuals.sample(500, random_state=1))  # sample for speed
print(f'Shapiro-Wilk p-value: {p:.4f}')
```

A p-value > 0.05 → fail to reject normality. If violated, transformations of **y** (log, Box-Cox) often help.

3.5 Multicollinearity

What it is: When two or more predictors share a large proportion of variance, OLS coefficient estimates become unstable (large standard errors).

Diagnostic: Variance Inflation Factor (VIF).

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Compute VIF for each feature
X_train_const = sm.add_constant(X_train)  # statsmodels expects intercept
vif = pd.DataFrame({
    'feature': X_train_const.columns,
    'VIF': [variance_inflation_factor(X_train_const.values, i)
            for i in range(X_train_const.shape[1])]
})
print(vif)
```

- **VIF > 5** → moderate collinearity
- **VIF > 10** → severe collinearity (action needed)

Remedies

1. **Drop one of the correlated features**
2. **Combine them** (e.g., via Principal Component Analysis)
3. **Apply regularisation** (Ridge or Elastic Net) - see next section.

---

4. Regularisation: Ridge, Lasso, Elastic Net

When the OLS assumptions break- especially multicollinearity or high dimensionality-regularisation adds a penalty term to the loss function, shrinking coefficients toward zero and reducing variance.

4.1 Bias-Variance Trade-off Recap

No regularisation (OLS)	With regularisation
Low bias, high variance	Slightly higher bias, lower variance
Sensitive to noisy features	More robust, better generalisation

The key is to **choose the penalty strength** (λ, called `alpha` in Scikit-learn) that minimises out-of-sample error.

4.2 Ridge (L2)

Loss function

$$\min_{\beta} \; ||y - X\beta||2^2 + \alpha ||\beta||_2^2 $$

- Shrinks coefficients **continuously** (never exactly zero).
- Handles multicollinearity well because the penalty stabilises the inverse of $X^\top X$.

```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)          # alpha = λ
ridge.fit(X_train, y_train)
print(ridge.coef_)
```

4.3 Lasso (L1)

Loss function

$$\min_{\beta} \; ||y - X\beta||2^2 + \alpha ||\beta||_1 $$

- Encourages **sparsity**: many coefficients become exactly zero → built-in feature selection.
- Works best when only a few predictors truly matter.

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.01, max_iter=10_000)
lasso.fit(X_train, y_train)
print(lasso.coef_)
```

4.4 Elastic Net (L1 + L2)

Loss function

$$\min_{\beta} \; \|y - X\beta\|_2^2 + \alpha \big[(1-\rho)\|\beta\|_2^2 + \rho \|\beta\|_1 \big]$$

- Combines Ridge's stability with Lasso's sparsity.
- Useful when predictors are highly correlated ***and*** you still want feature selection.

```
from sklearn.linear_model import ElasticNet
elastic = ElasticNet(alpha=0.01, l1_ratio=0.5, max_iter=10_000)
elastic.fit(X_train, y_train)
print(elastic.coef_)
```

4.5 Hyper-parameter Tuning with `GridSearchCV`

Choosing the right α (and l1_ratio for Elastic Net) is a model-selection problem. Cross-validation gives an unbiased estimate of out-of-sample performance.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'alpha': np.logspace(-4, 2, 30),   # 30 values from 1e-4 to 1e2
    'l1_ratio': [0.0, 0.5, 0.9]        # only used by ElasticNet
}

elastic_cv = GridSearchCV(
    estimator=ElasticNet(max_iter=10_000),
    param_grid=param_grid,
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

elastic_cv.fit(X_train, y_train)
print('Best α:', elastic_cv.best_params_['alpha'])
print('Best l1_ratio:', elastic_cv.best_params_['l1_ratio'])
print('Best CV RMSE:', np.sqrt(-elastic_cv.best_score_))
```

Tip: When the dataset is **very high-dimensional (p >> n)**, start with a **log-spaced α grid** and a **coarse l1_ratio**; once you've narrowed the region, refine the grid.

---

5. Logistic Regression for Binary Classification

Linear regression predicts a continuous outcome; logistic regression adapts the same linear-combination idea to a **binary** target (0/1) by applying the logistic (sigmoid) function.

5.1 From Linear to Logistic Link Function

The model assumes:

$$ \Pr(y=1 \mid X) = \sigma(X\beta) = \frac{1}{1 + e^{-X\beta}} $$

The loss function is the **negative log-likelihood** (also known as binary cross-entropy):

\[\min***{\beta}* \; -*sum*{i=1}^{n} \big[y***i* *log* *sigma(X*i\beta) + (1-y***i)* *log* *(1-*sigma(X*i\beta)) \big] \]

Scikit-learn solves this with **iteratively re-weighted least squares** (IRLS) under the hood.

5.2 Interpreting Odds Ratios

Coefficient β_j tells us how the **log-odds** change per unit increase in predictor X_j, holding other variables constant.

\[\text{Odds Ratio (OR)} = e^{\beta_j} \]

- **OR > 1** → higher odds of the positive class as X_j increases.
- **OR < 1** → lower odds.

Example interpretation: If β_age = 0.04, then OR = e^{0.04} ≈ 1.04 → each extra year of age raises the odds of survival by 4 %.

5.3 Regularised Logistic Regression

Logistic regression also supports L1, L2, and Elastic Net penalties via the `penalty` argument. The same bias-variance logic applies, and the hyper-parameter `C` (inverse of α) controls regularisation strength.

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(
    penalty='elasticnet',       # 'l2', 'l1', or 'elasticnet'
    solver='saga',              # saga supports elasticnet
    l1_ratio=0.7,
    C=1.0,                      # inverse regularisation strength
    max_iter=10_000,
    random_state=42
)
logreg.fit(X_train, y_train)
```

Why `solver='saga'`? It handles both L1 and L2 penalties efficiently for large datasets.

---

6. Putting It All Together: Scikit-learn Pipelines

A **pipeline** bundles preprocessing, feature engineering, and model fitting into a single object. Benefits:

- Guarantees the same transformations are applied to training and future test data.
- Enables clean hyper-parameter search across ***both*** preprocessing and model steps.
- Improves reproducibility and readability.

6.1 Pre-processing Steps

Step	Typical Operations
Imputation	Fill missing numeric values (`SimpleImputer(strategy='median')`)

Step	Typical Operations
Scaling	Standardise features (`StandardScaler`) - important for regularisation
Encoding	One-hot encode categoricals (`OneHotEncoder(handle_unknown='ignore')`)
Feature Generation	Polynomial features (`PolynomialFeatures(degree=2, include_bias=False)`)

6.2 Pipeline for Regression

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, PolynomialFeatures
from sklearn.impute import SimpleImputer

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

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

preprocess = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

regression_pipe = Pipeline(steps=[
    ('preprocess', preprocess),
    ('model', ElasticNet(alpha=0.01, l1_ratio=0.5, max_iter=10_000))
])

# Cross-validate the entire pipeline
from sklearn.model_selection import cross_val_score
scores = cross_val_score(regression_pipe, X, y, cv=5,
                         scoring='neg_root_mean_squared_error')
print('CV RMSE:', -scores.mean())
```

6.3 Pipeline for Classification

```
classification_pipe = Pipeline(steps=[
    ('preprocess', preprocess),              # same preprocessing object works
    ('model', LogisticRegression(
        penalty='elasticnet',
        solver='saga',
        l1_ratio=0.7,
        C=0.5,
        max_iter=10_000,
        random_state=42
    ))
])

# Example: 5-fold cross-validation for accuracy
from sklearn.model_selection import cross_val_score
acc = cross_val_score(classification_pipe, X, y, cv=5,
                      scoring='accuracy')
print('CV Accuracy:', acc.mean())
```

Tip: When you have many hyper-parameters (e.g., `alpha`, `l1_ratio`, `C`), wrap the pipeline in a `GridSearchCV` and refer to parameters with the `step__parameter` syntax (`'model__alpha': [0.001, 0.01, 0.1]`).

---

7. Practical Walk-throughs

Below are three end-to-end notebooks you can copy-paste into a Jupyter environment. Each illustrates a different facet of linear models.

7.1 Housing Price Prediction (OLS + Diagnostics)

```
# ------------------------------------------------------------
# 1️⃣ Load data (Boston housing - replaced by OpenML for newer sklearn)
# ------------------------------------------------------------
from sklearn.datasets import fetch_openml
boston = fetch_openml(name='boston', version=1, as_frame=True)
X = boston.data
y = boston.target.astype(float)   # target is 'MEDV'

# ------------------------------------------------------------
# 2️⃣ Train-test split
# ------------------------------------------------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=1
)

# ------------------------------------------------------------
# 3️⃣ Fit OLS (no regularisation)
# ------------------------------------------------------------
from sklearn.linear_model import LinearRegression
ols = LinearRegression()
ols.fit(X_train, y_train)

# ------------------------------------------------------------
# 4️⃣ Diagnostics
# ------------------------------------------------------------
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import numpy as np

# Predictions & residuals
y_pred = ols.predict(X_test)
residuals = y_test - y_pred

# 4a - Residuals vs. fitted
sns.scatterplot(x=y_pred, y=residuals)
plt.axhline(0, color='red', ls='--')
plt.xlabel('Fitted values')
plt.ylabel('Residual')
```

... (continued on next page)

```
plt.title('Residuals vs. Fitted')
plt.show()

# 4b - Q-Q plot
sm.qqplot(residuals, line='45')
plt.title('Q-Q plot of residuals')
plt.show()

# 4c - VIF (on training set)
X_train_const = sm.add_constant(X_train)
vif = pd.DataFrame({
    'feature': X_train_const.columns,
    'VIF': [variance_inflation_factor(X_train_const.values, i)
            for i in range(X_train_const.shape[1])]
})
print(vif.sort_values('VIF', ascending=False))

# ----------------------------------------------------------------
# 5️⃣ Performance metrics
# ----------------------------------------------------------------
from sklearn.metrics import mean_squared_error, r2_score
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)
print(f'RMSE: {rmse:.2f}, R²: {r2:.3f}')
```

What you'll see

- A **funnel-shaped** residual plot → heteroscedasticity (common in housing data).
- **VIF** > 10 for `NOX` and `DIS` → multicollinearity.
- **RMSE** around $5k (in $1000s) and modest R^2 (~0.7).

Next steps (outside the notebook): apply a log-transform to `MEDV`, drop or combine collinear variables, or switch to Ridge regression.

7.2 Regularised Regression on a High-Dimensional Synthetic Set

```
# ------------------------------------------------------------
# 1️⃣ Generate synthetic data (n=200, p=500)
# ------------------------------------------------------------
import numpy as np
np.random.seed(42)

n_samples, n_features = 200, 500
X_syn = np.random.randn(n_samples, n_features)

# True coefficients: only 10 are non-zero
true_beta = np.zeros(n_features)
true_beta[:10] = np.random.randn(10) * 5   # strong signals
y_syn = X_syn @ true_beta + np.random.randn(n_samples) * 2.0   # noise

# ------------------------------------------------------------
# 2️⃣ Train-test split
# ------------------------------------------------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_syn, y_syn, test_size=0.3, random_state=0
)

# ------------------------------------------------------------
# 3️⃣ Lasso (feature selection) + cross-validation
# ------------------------------------------------------------
from sklearn.linear_model import LassoCV
lasso_cv = LassoCV(cv=5, alphas=np.logspace(-4, 0, 100),
                   max_iter=10_000, n_jobs=-1)
lasso_cv.fit(X_train, y_train)

print('Best alpha (λ):', lasso_cv.alpha_)
print('Number of selected features:', np.sum(lasso_cv.coef_ != 0))

# ------------------------------------------------------------
# 4️⃣ Compare with Ridge (no sparsity)
# ------------------------------------------------------------
from sklearn.linear_model import RidgeCV
ridge_cv = RidgeCV(alphas=np.logspace(-4, 2, 100), cv=5)
ridge_cv.fit(X_train, y_train)
```

... (continued on next page)

```
print('Ridge best α:', ridge_cv.alpha_)
print('Ridge coefficients norm:', np.linalg.norm(ridge_cv.coef_))

# ------------------------------------------------------------
# 5️⃣ Evaluate on hold-out test set
# ------------------------------------------------------------
from sklearn.metrics import mean_squared_error
lasso_rmse = np.sqrt(mean_squared_error(y_test, lasso_cv.predict(X_test)))
ridge_rmse = np.sqrt(mean_squared_error(y_test, ridge_cv.predict(X_test)))
print(f'Lasso RMSE: {lasso_rmse:.3f}')
print(f'Ridge RMSE: {ridge_rmse:.3f}')
```

Take-aways

- **Lasso** discards > 480 irrelevant features, making the model interpretable.
- **Ridge** keeps all coefficients but shrinks them, often yielding a slightly lower RMSE when many weak predictors exist.
- The synthetic example demonstrates why **regularisation is essential** when $p \gg n$.

7.3 Titanic Survival Prediction - Logistic Regression with Pipelines

```
# ----------------------------------------------------------------
# 1️⃣ Load the Titanic dataset (via seaborn)
# ----------------------------------------------------------------
import seaborn as sns
titanic = sns.load_dataset('titanic')
titanic = titanic.dropna(subset=['embarked'])   # keep rows with known port

# Target: survived (0/1)
y = titanic['survived'].astype(int)

# Features: mix of numeric & categorical
X = titanic.drop(columns=['survived', 'who', 'deck', 'alive', 'class',
                          'embark_town', 'adult_male', 'alone'])

# ----------------------------------------------------------------
# 2️⃣ Train-test split
# ----------------------------------------------------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# ----------------------------------------------------------------
# 3️⃣ Define preprocessing (impute + encode)
# ----------------------------------------------------------------
numeric_features = X.select_dtypes(include=['int64', 'float64']).columns
categorical_features = X.select_dtypes(include=['object', 'category', 'bool']).columns

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

preprocess = ColumnTransformer(
    transformers=[
```

... (continued on next page)

```
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# ------------------------------------------------------------
# 4️⃣ Logistic regression (elastic net) wrapped in a pipeline
# ------------------------------------------------------------
log_pipe = Pipeline(steps=[
    ('preprocess', preprocess),
    ('clf', LogisticRegression(
        penalty='elasticnet',
        solver='saga',
        l1_ratio=0.5,
        C=1.0,
        max_iter=10_000,
        random_state=0
    ))
])

# ------------------------------------------------------------
# 5️⃣ Hyper-parameter search (grid on C & l1_ratio)
# ------------------------------------------------------------
param_grid = {
    'clf__C': np.logspace(-2, 2, 5),          # 0.01 to 100
    'clf__l1_ratio': [0.0, 0.3, 0.5, 0.7, 1.0] # from Ridge to Lasso
}

grid = GridSearchCV(log_pipe, param_grid, cv=5,
                    scoring='roc_auc', n_jobs=-1, verbose=0)
grid.fit(X_train, y_train)

print('Best parameters:', grid.best_params_)
print('Best ROC-AUC (CV):', grid.best_score_)

# ------------------------------------------------------------
# 6️⃣ Evaluate on test set
# ------------------------------------------------------------
from sklearn.metrics import classification_report, roc_auc_score

y_pred = grid.predict(X_test)
```

... (continued on next page)

```
y_proba = grid.predict_proba(X_test)[:, 1]

print(classification_report(y_test, y_pred))
print('Test ROC-AUC:', roc_auc_score(y_test, y_proba))
```

What you'll learn

- The **pipeline** automatically handles missing ages, encodes `sex`, `embarked`, `pclass`, etc.
- **Elastic-net** balances the benefits of L2 (stability) and L1 (feature selection). In practice, the best model often drops `deck` (many missing values) and keeps only a handful of strong predictors (`sex`, `class`, `fare`).
- **ROC-AUC** (≈ 0.84)is a robust metric for imbalanced binary problems; you can also explore calibration curves.

---

8. Key Takeaways

1. **Linear models are the first line of defence** in any supervised learning project. They are quick to train, easy to interpret, and provide a solid baseline.
2. **Assumption diagnostics are not optional**-checking linearity, homoscedasticity, independence, normality, and multicollinearity protects you from misleading inferences.
3. **Regularisation (Ridge, Lasso, Elastic Net)** mitigates over-fitting, stabilises coefficient estimates, and can perform automatic feature selection. The bias-variance trade-off is tuned via cross-validation.
4. **Logistic regression extends the linear-model toolkit to classification**; odds ratios give a direct, domain-friendly interpretation of effect size.
5. **Scikit-learn pipelines unify preprocessing, feature engineering, and model training**. They enable reproducible experiments and make hyper-parameter search

across the whole workflow straightforward.

6. **Practical workflow checklist**

- ☐ Load data → exploratory analysis (Module 2).
- ☐ Clean & engineer features (Module 3).
- ☐ Split into train / validation / test sets.
- ☐ Fit a plain OLS or logistic model.
- ☐ Run diagnostic plots & statistical tests.
- ☐ If assumptions fail → transform variables **or** move to regularised models.
- ☐ Use `GridSearchCV` (or `RandomizedSearchCV`) inside a pipeline to find optimal λ / C.
- ☐ Evaluate on a held-out test set with appropriate metrics (RMSE, R^2, ROC-AUC).
- ☐ Document coefficient interpretations, model limitations, and next steps.

7. **Next module preview** - We will move beyond linearity to **tree-based models** (Decision Trees, Random Forests, Gradient Boosting), which automatically capture non-linear interactions and often outperform linear models on messy, high-dimensional data.

> ***Bottom line: Mastering linear models gives you a statistical compass that guides every later modelling decision. When you know how and why a simple model works- or fails- you can confidently navigate toward more complex algorithms, always keeping an eye on interpretability, bias, and variance. Happy modeling!***

Supervised Learning - Tree-Based Models

> ***"If you can't explain it, you don't really understand it." - Andrew***

Gelman

Tree-based models sit at the sweet spot between interpretability and predictive power. In this chapter we will:

- **Recall** why clean, well-engineered data matters (Modules 2 & 3).
- **Build** decision trees from scratch, mastering impurity measures (Gini, entropy).
- **Scale** trees with ensembles - Random Forests, Gradient Boosting, XGBoost, LightGBM.
- **Tune** depth, number of estimators, learning rate, and other knobs for optimal performance.
- **Interpret** what the model has learned using feature importance and SHAP values.

The goal is not just to run a black-box algorithm, but to understand **how** the model works, **why** it makes a particular prediction, and **how** to make it better - all while keeping the code approachable for anyone with a basic Python background.

---

1 Introduction

1.1 Why Tree-Based Models?

Strength	Typical Use-Case
Non-linear relationships - capture interactions automatically.	Click-through-rate prediction, churn modeling.
Mixed data types - handle numerical **and** categorical features (with minimal preprocessing).	Credit scoring, medical diagnostics.
Interpretability - visualizable splits, global and local importance metrics.	Explainable AI, regulatory environments.

Strength	Typical Use-Case
Robustness to outliers - a single extreme value rarely changes the split.	Sensor data, financial time series.
Fast inference - once trained, a tree is just a series of if-else checks.	Real-time recommendation engines.

1.2 Where Trees Sit in the ML Landscape

- **Linear models** (Module 4)assume additive, monotonic relationships.
- **Tree-based models** relax that assumption, allowing the data to dictate ***where*** and ***how*** to split.
- **Ensembles** (bagging & boosting) combine many weak trees into a strong predictor, often beating deep neural nets on tabular data.

> ***Bottom line: If you have a well-structured tabular dataset and you want a blend of performance and interpretability, start with a decision tree, then graduate to Random Forests or Gradient Boosting.***

---

2 Core Concepts

2.1 Decision Trees 101

A decision tree is a **flow-chart** that recursively partitions the feature space. Each internal node asks a **question** about a single feature, each leaf holds a **prediction** (class label or numeric value).

2.1.1 Anatomy of a Split

```
if feature_j ≤ threshold:
      go left
else:
      go right
```

- **Feature (j)** - the column we examine.
- **Threshold** - the value that separates the left/right children.
- **Purity** - how "clean" the resulting groups are (see Section 2.2).

The tree grows **top- down** (greedy, locally optimal splits) until a stopping criterion is met:

- `max_depth` reached
- `min_samples_leaf` too small
- No impurity improvement (`min_impurity_decrease`)

2.1.2 From Tree to Prediction

- **Classification:** leaf stores the **majority class** (or class probabilities).
- **Regression:** leaf stores the **mean** (or median) target value of its training samples.

> ***Tip: Visualizing a small tree (≤ 5 levels) with `sklearn.tree.plot_tree` is an excellent sanity check before you scale up.***

2.2 Impurity Measures - The Engine Behind Splits

A split is chosen to **maximise the reduction in impurity** (i.e., make the children purer than the parent). Two most- used impurity functions are **Gini impurity** and **entropy** (aka information gain).

2.2.1 Gini Impurity

For a node with class probabilities p_k ($k = 1...K$):

$$ \text{Gini} = 1 - \sum_{k=1}^{K} p_k^2 $$

- **Range:** 0 (pure) → $1 - \frac{1}{K}$ (max impurity).
- **Intuition:** Probability of misclassifying a randomly drawn sample if we label it according to the class distribution of the node.

2.2.2 Entropy (Information Gain)

$$ \text{Entropy} = -\sum_{k=1}^{K} p_k \log_2(p_k) $$

- **Range:** 0 (pure) → $\log_2(K)$ (max impurity).
- **Interpretation:** Expected number of bits needed to encode the class label.

2.2.3 Calculating the Gain

For a candidate split that creates left (`L`) and right (`R`) children:

$$ \text{Gain} = \text{Impurity(parent)} - \frac{N_L}{N}\, \text{Impurity}(L) - \frac{N_R}{N}\,\text{Impurity}(R) $$

- `N` = total samples in parent node.
- The split with the **largest gain** is selected.

> ***Quick Python demo:***

```
import numpy as np
from sklearn.metrics import gini, log_loss  # note: scikit-learn doesn't expose gini directly

def gini_impurity(y):
    """y = array of class labels for a node."""
    _, counts = np.unique(y, return_counts=True)
    probs = counts / counts.sum()
    return 1 - np.sum(probs**2)

def entropy_impurity(y):
    _, counts = np.unique(y, return_counts=True)
    probs = counts / counts.sum()
    return -np.sum(probs * np.log2(probs + 1e-9))

# toy example
y_parent = np.array([0,0,1,1,1,0,1])
y_left   = np.array([0,0,0])
y_right  = np.array([1,1,1,1])
print("Gini parent:", gini_impurity(y_parent))
print("Gini left  :", gini_impurity(y_left))
print("Gini right :", gini_impurity(y_right))
```

2.3 Pruning - Controlling Over-fit

A fully grown tree can memorize the training data, leading to **high variance**. Pruning removes branches that provide little predictive power.

- **Pre-pruning (early stopping):** Set limits (`max_depth`, `min_samples_leaf`, `min_impurity_decrease`) ***before*** training.
- **Post-pruning (cost-complexity pruning):** Grow a large tree, then iteratively remove nodes that increase the **complexity parameter** α the least. In scikit-learn this is accessed via `DecisionTreeClassifier(cost_complexity_pruning_path=...)`.

> ***Rule of thumb: Start with pre-pruning; only use post-pruning if you suspect the early-stop limits are too aggressive.***

2.4 Ensembles - From One Tree to Many

2.4.1 Bagging (Bootstrap Aggregating)

• **Idea:** Reduce variance by averaging many ***independent*** trees trained on **bootstrap samples** (sampling with replacement).

• **Algorithm:**

1. For `m = 1 ... M` (number of estimators):

• Draw a bootstrap dataset of size `n` (same as original).

• Train a full decision tree (often **unpruned**).

2. **Prediction:**

• **Classification:** majority vote.

• **Regression:** average of predictions.

• **Result: Random Forest** - adds an extra layer of randomness by selecting a random subset of features at each split (`max_features`).

2.4.2 Random Forest - Practical Details

Hyperparameter	Typical Values	Effect
`n_estimators`	100-500 (more for very noisy data)	More trees → lower variance, diminishing returns after ~200.
`max_depth`	`None` (full) or 10-30	Controls over-fit; shallow forests are faster.
`max_features`	`"sqrt"` (classification), `"log2"` (regression)	Fewer features per split → decorrelates trees, improves ensemble strength.
`min_samples_leaf`	1-5	Larger leaves → smoother predictions, less over-fit.
`bootstrap`	`True` (default)	Turn off for **extremely small datasets** (use `bootstrap=False`).

Python snippet (Random Forest on Titanic):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

X = train_df.drop(columns=['Survived'])
y = train_df['Survived']

X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

rf = RandomForestClassifier(
    n_estimators=300,
    max_depth=12,
    max_features='sqrt',
    min_samples_leaf=2,
    random_state=42,
    n_jobs=-1)                      # use all cores

rf.fit(X_train, y_train)
pred = rf.predict(X_val)
print("Validation accuracy:", accuracy_score(y_val, pred))
print(classification_report(y_val, pred))
```

2.4.3 Boosting - Turning Weak Learners into a Strong One

- **Core idea:** Build trees **sequentially**, each new tree correcting the errors of its predecessors.
- **Two flavors:**

Boosting Type	Loss Function	Typical Use
AdaBoost	Exponential loss (classification)	Simple, works well on clean data.
Gradient Boosting	Differentiable loss (e.g., MSE, log-loss)	Most flexible, forms the basis of XGBoost & LightGBM.

- **Gradient Boosting algorithm (high-level):**

1. Initialise model with a constant prediction (e.g., mean of `y`).
2. For `m = 1 ... M`:

• Compute **pseudo-residuals** - the gradient of the loss w.r.t. current predictions.

• Fit a **shallow tree** (often depth 3-5) to those residuals.

• Update the model:

\[F***{m}(x) = F***{m-1}(x) + \eta \cdot h_m(x) \] where \(\eta\) is the **learning rate** (shrinkage).

3. Final prediction = sum of all trees.

• **Key hyperparameters:**

Param	Meaning	Typical Range
`n_estimators`	Number of trees	100-2000 (more = better, slower)
`learning_rate` (`eta`)	Step size of each update	0.01-0.3 (smaller → need more trees)
`max_depth`	Tree depth (often 3-8)	Controls bias-variance trade-off
`subsample`	Fraction of rows per tree (stochastic GB)	0.6-1.0
`colsample_bytree`	Fraction of features per split	0.5-1.0

> ***Rule of thumb: Start with `learning_rate=0.1`, `max_depth=4`, `n_estimators=500`. Tune later with cross-validation.***

2.4.4 XGBoost - "Extreme Gradient Boosting"

XGBoost is a **high-performance** implementation of gradient boosting with several engineering tricks:

Feature	Why It Helps
Second-order Taylor expansion (uses both gradient and Hessian)	More accurate step direction, faster convergence.
Regularization (`lambda`, `alpha`)	Penalises leaf weights → reduces over-fit.
Sparsity-aware split finding	Handles missing values natively.
Parallel tree construction	Utilises all CPU cores; GPU support available.
Built-in early stopping	Stops training when validation error stops improving.

XGBoost example (California Housing regression):

```
import xgboost as xgb
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

X = housing_df.drop(columns='median_house_value')
y = housing_df['median_house_value']

X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.2, random_state=42)

dtrain = xgb.DMatrix(X_train, label=y_train)
dvalid = xgb.DMatrix(X_valid, label=y_valid)

params = {
    "objective": "reg:squarederror",
    "eval_metric": "rmse",
    "learning_rate": 0.05,
    "max_depth": 6,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "lambda": 1.0,          # L2 regularisation
    "alpha": 0.0,           # L1 regularisation
    "seed": 42,
    "nthread": -1           # use all cores
}

evallist = [(dtrain, 'train'), (dvalid, 'eval')]
bst = xgb.train(params,
                dtrain,
                num_boost_round=2000,
                evals=evallist,
                early_stopping_rounds=50,
                verbose_eval=100)

preds = bst.predict(dvalid)
print("RMSE on validation set:", np.sqrt(mean_squared_error(y_valid, preds)))
```

Notice the `early_stopping_rounds=50` - training halts automatically once the validation RMSE hasn't improved for 50 consecutive rounds.

2.4.5 LightGBM - "Light Gradient Boosting Machine"

LightGBM is another state-of-the-art gradient-boosting library, optimised for

speed and memory:

Innovation	Effect
Leaf-wise growth (instead of level-wise)	Faster convergence, deeper trees with fewer leaves.
Histogram-based split finding	Bins continuous features into discrete histograms → reduces computation.
Exclusive Feature Bundling (EFB)	Merges mutually exclusive features, cutting dimensionality.
Categorical feature handling	Directly supports categorical columns without one-hot encoding.

LightGBM example (binary classification on Titanic):

```
import lightgbm as lgb
from sklearn.metrics import roc_auc_score

X = train_df.drop(columns='Survived')
y = train_df['Survived']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

train_data = lgb.Dataset(X_train, label=y_train, categorical_feature='auto')
test_data  = lgb.Dataset(X_test,  label=y_test,   categorical_feature='auto')

params = {
    "objective": "binary",
    "metric": "auc",
    "learning_rate": 0.07,
    "num_leaves": 31,          # approx 2^depth
    "max_depth": -1,           # leaf-wise growth, no explicit depth limit
    "feature_fraction": 0.9,
    "bagging_fraction": 0.8,
    "bagging_freq": 5,
    "verbosity": -1,
    "seed": 42
}

gbm = lgb.train(params,
                train_data,
                num_boost_round=2000,
                valid_sets=[train_data, test_data],
                early_stopping_rounds=100,
                verbose_eval=100)

pred_proba = gbm.predict(X_test, num_iteration=gbm.best_iteration)
print("Test AUC:", roc_auc_score(y_test, pred_proba))
```

LightGBM often finishes training **2-5× faster** than XGBoost on the same dataset, especially when you have **high-cardinality categorical variables**.

2.5 Hyperparameter Tuning - Finding the Sweet Spot

2.5.1 Grid Search vs. Random Search vs. Bayesian Optimisation

Method	Pros	Cons
Grid Search (exhaustive)	Systematic, easy to interpret results.	Explodes combinatorially; wasteful if many parameters are irrelevant.
Random Search	Covers space more efficiently; good for high-dimensional hyper-spaces.	Still may miss optimal region.
Bayesian Optimisation (e.g., `Optuna`, `scikit-optimize`)	Learns from past trials, converges faster to good region.	Slightly more complex to set up.

Practical advice:

- Start with **random search** on a **wide** range for `max_depth`, `learning_rate`, `n_estimators`.
- Once you have a promising region, switch to **grid** or **bayesian** refinement.
- Always keep a **validation set** (or use cross-validation) for unbiased evaluation.

2.5.2 Recommended Search Spaces

Model	Parameter	Search Range
DecisionTree	`max_depth`	3 - 30 (or `None`)

| | `min_samples_leaf` | 1 - 20 |

RandomForest	**`n_estimators`**	**100 - 1000**
XGBoost / LightGBM	`learning_rate`	0.01 - 0.3 (log-scale)

| | `max_depth` | 5 - 30 | | | `n_estimators` | 100 - 5000 |

`max_depth`	**3 - 12**

max_depth	**3 - 12**
colsample_bytree / feature_fraction	0.5 - 1.0
lambda (L2)	0 - 5
alpha (L1)	0 - 5

Sample `RandomizedSearchCV` for XGBoost (classification):

```
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBClassifier
import scipy.stats as stats

param_dist = {
    "learning_rate": stats.loguniform(0.01, 0.3),
    "max_depth": stats.randint(3, 12),
    "n_estimators": stats.randint(200, 2000),
    "subsample": stats.uniform(0.6, 0.4),
    "colsample_bytree": stats.uniform(0.5, 0.5),
    "reg_lambda": stats.loguniform(0.1, 5),
    "reg_alpha": stats.loguniform(0.1, 5)
}

xgb_clf = XGBClassifier(
    objective='binary:logistic',
    eval_metric='logloss',
    n_jobs=-1,
    random_state=42,
    use_label_encoder=False)

search = RandomizedSearchCV(
    estimator=xgb_clf,
    param_distributions=param_dist,
    n_iter=50,
    scoring='roc_auc',
    cv=5,
    verbose=1,
    random_state=42,
    n_jobs=-1)

search.fit(X_train, y_train)
print("Best AUC:", search.best_score_)
print("Best params:", search.best_params_)
```

2.6 Model Interpretation - From Black Box to Insight

2.6.1 Global Feature Importance

- **Mean Decrease Impurity (MDI)** - average impurity reduction contributed by each feature across all trees.
- **Mean Decrease Accuracy (MDA)** - permutation importance: shuffle a feature's values, measure drop in performance.

Scikit-learn permutation importance example:

```
from sklearn.inspection import permutation_importance

result = permutation_importance(rf, X_val, y_val,
                                n_repeats=10,
                                random_state=42,
                                n_jobs=-1)

import matplotlib.pyplot as plt
sorted_idx = result.importances_mean.argsort()
plt.barh(range(len(sorted_idx)), result.importances_mean[sorted_idx])
plt.yticks(range(len(sorted_idx)), X.columns[sorted_idx])
plt.title("Permutation Importance (Random Forest)")
plt.show()
```

MDI is fast (already computed during training) but can be **biased towards high-cardinality features**. ***MDA*** is more reliable, albeit slower.

2.6.2 Local Explanations - SHAP Values

SHAP (SHapley Additive exPlanations) provides a unified framework to attribute a model's prediction to each feature. It works for any tree-based model (including XGBoost & LightGBM) via the **TreeSHAP** algorithm - exact and extremely fast.

Installation:

```
pip install shap
```

Using SHAP with XGBoost (binary classification):

```
import shap

# Train a model (already done as `bst` in earlier XGBoost example)
explainer = shap.TreeExplainer(bst)      # works for XGB, LightGBM, CatBoost
shap_values = explainer.shap_values(X_val)

# Summary plot - global view
shap.summary_plot(shap_values, X_val, plot_type="bar")

# Force plot - individual prediction (e.g., first validation sample)
shap.force_plot(explainer.expected_value,
                shap_values[0,:],
                X_val.iloc[0,:],
                matplotlib=True)
```

Interpretation tips:

- **Positive SHAP value** → feature pushes prediction **higher** (e.g., towards "Survived").
- **Negative SHAP value** → feature pushes prediction **lower**.
- The **summary plot** orders features by overall impact, showing both magnitude and direction.

2.6.3 Partial Dependence Plots (PDP)

PDPs visualize the average model response as a function of a single feature (or a pair). Scikit-learn provides `PartialDependenceDisplay`.

```
from sklearn.inspection import PartialDependenceDisplay

PartialDependenceDisplay.from_estimator(
    rf, X_val, ["Age", "Fare"], kind="average")
```

Useful when you want to confirm a monotonic relationship or detect interaction

effects.

3️⃣ Practical Application

In this section we walk through a **complete, reproducible workflow** for two classic tabular problems:

Dataset	Goal	Model(s)
Titanic (Kaggle)	Predict survival (binary classification)	Decision Tree → Random Forest → LightGBM
California Housing (scikit-learn)	Predict median house value (regression)	Decision Tree → Gradient Boosting → XGBoost

> ***Prerequisite:** The reader should have completed Modules 2️⃣ & 3️⃣ (data cleaning, feature engineering). We will reuse the cleaned data frames `titanic_clean` and `housing_clean`.*

3.1 Environment Setup

```
# Create a fresh env (optional but recommended)
conda create -n ml-trees python=3.11 -y
conda activate ml-trees

# Core libraries
pip install numpy pandas scikit-learn matplotlib seaborn tqdm

# Tree-based libraries
pip install xgboost lightgbm shap optuna
```

3.2 Titanic - From Raw to Insight

3.2.1 Data Overview

```
import pandas as pd
titanic = pd.read_csv('titanic/train.csv')
titanic.head()
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embar
1	0	3	...	male	22	1	0	A/5 21171	7.25	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...

3.2.2 Feature Engineering (quick recap)

```
def preprocess_titanic(df):
    df = df.copy()
    # 1️⃣ Title extraction
    df['Title'] = df['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)
    df['Title'] = df['Title'].replace(['Lady', 'Countess','Capt','Col','Don',
                                       'Dr','Major','Rev','Sir','Jonkheer','Dona'],
                                      'Rare')
    df['Title'] = df['Title'].replace('Mlle', 'Miss')
    df['Title'] = df['Title'].replace('Ms', 'Miss')
    df['Title'] = df['Title'].replace('Mme', 'Mrs')

    # 2️⃣ Age imputation by median per Title
    df['Age'] = df.groupby('Title')['Age'].transform(
        lambda x: x.fillna(x.median()))

    # 3️⃣ Cabin: keep only first letter, fill missing with 'U'
    df['Cabin'] = df['Cabin'].fillna('U')
    df['Cabin'] = df['Cabin'].str[0]

    # 4️⃣ FamilySize
    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

    # 5️⃣ IsAlone flag
    df['IsAlone'] = (df['FamilySize'] == 1).astype(int)

    # 6️⃣ Drop columns we don't need for modeling
    drop_cols = ['PassengerId','Name','Ticket','Ticket','SibSp','Parch']
    df = df.drop(columns=drop_cols)

    # 7️⃣ One-hot encode categorical columns
    df = pd.get_dummies(df, columns=['Sex','Embarked','Title','Cabin'],
                        drop_first=True)
    return df

titanic_clean = preprocess_titanic(titanic)
titanic_clean.head()
```

The resulting DataFrame now contains only **numeric columns**, ready for any tree algorithm (no scaling needed).

3.2.3 Train-Test Split

```
from sklearn.model_selection import train_test_split

X = titanic_clean.drop('Survived', axis=1)
y = titanic_clean['Survived']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
```

3.2.4 Baseline Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, roc_auc_score

tree = DecisionTreeClassifier(
    max_depth=5,
    min_samples_leaf=5,
    random_state=42)

tree.fit(X_train, y_train)
y_pred = tree.predict(X_test)
y_proba = tree.predict_proba(X_test)[:,1]

print("Accuracy:", accuracy_score(y_test, y_pred))
print("AUC    :", roc_auc_score(y_test, y_proba))
```

Typical result: **≈ 78 % accuracy**, **AUC ≈ 0.84** - decent but leaves room for improvement.

3.2.5 Random Forest - Boosting Stability

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=500,
    max_depth=12,
    max_features='sqrt',
    min_samples_leaf=2,
    random_state=42,
    n_jobs=-1)

rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)
rf_proba = rf.predict_proba(X_test)[:,1]

print("RF Accuracy:", accuracy_score(y_test, rf_pred))
print("RF AUC    :", roc_auc_score(y_test, rf_proba))
```

Typical result: **≈ 84 % accuracy**, **AUC ≈ 0.91** - a solid jump.

3.2.6 LightGBM - Fast, Accurate, and Tunable

```
import lightgbm as lgb
from sklearn.metrics import roc_auc_score

lgb_train = lgb.Dataset(X_train, label=y_train, categorical_feature='auto')
lgb_valid = lgb.Dataset(X_test,  label=y_test,  categorical_feature='auto')

lgb_params = {
    "objective": "binary",
    "metric": "auc",
    "learning_rate": 0.07,
    "num_leaves": 31,
    "feature_fraction": 0.9,
    "bagging_fraction": 0.8,
    "bagging_freq": 5,
    "verbosity": -1,
    "seed": 42
}

lgb_model = lgb.train(lgb_params,
                      lgb_train,
                      num_boost_round=2000,
                      valid_sets=[lgb_train, lgb_valid],
                      early_stopping_rounds=100,
                      verbose_eval=200)

lgb_pred = (lgb_model.predict(X_test) > 0.5).astype(int)
lgb_proba = lgb_model.predict(X_test)

print("LGB Accuracy:", accuracy_score(y_test, lgb_pred))
print("LGB AUC    :", roc_auc_score(y_test, lgb_proba))
```

Typical result: **≈ 85 % accuracy**, **AUC ≈ 0.92** - marginally better than Random Forest, but **training time** is ~½ of the RF.

3.2.7 Hyperparameter Optimisation with Optuna

```
import optuna
from sklearn.model_selection import cross_val_score

def objective(trial):
    param = {
        "objective": "binary",
        "metric": "auc",
        "learning_rate": trial.suggest_loguniform('lr', 1e-3, 0.3),
        "num_leaves": trial.suggest_int('num_leaves', 20, 150),
        "max_depth": trial.suggest_int('max_depth', 3, 12),
        "min_child_samples": trial.suggest_int('min_child_samples', 5, 100),
        "feature_fraction": trial.suggest_uniform('feature_fraction', 0.6, 1.0),
        "bagging_fraction": trial.suggest_uniform('bagging_fraction', 0.6, 1.0),
        "bagging_freq": trial.suggest_int('bagging_freq', 1, 10),
        "lambda_l1": trial.suggest_loguniform('lambda_l1', 1e-4, 10.0),
        "lambda_l2": trial.suggest_loguniform('lambda_l2', 1e-4, 10.0),
        "seed": 42,
        "verbosity": -1,
        "n_jobs": -1
    }

    lgb_clf = lgb.LGBMClassifier(**param)
    auc = cross_val_score(lgb_clf, X_train, y_train,
                          cv=5,
                          scoring='roc_auc',
                          n_jobs=-1).mean()
    return auc

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=80, timeout=1800)  # 30 minmax

print("Best AUC:", study.best_value)
print("Best params:", study.best_params)
```

After the optimisation, re-train `LGBMClassifier` with the best parameters on the full training set and evaluate on the hold-out test set. You'll typically see a **0.5-1 %AUC lift** - proof that even a well-tuned model can be nudged further.

3.2.8 Interpreting the LightGBM Model with SHAP

```
import shap, matplotlib.pyplot as plt

explainer = shap.TreeExplainer(lgb_model)
shap_vals = explainer.shap_values(X_test)

# Global summary
shap.summary_plot(shap_vals, X_test, plot_type="dot", max_display=12)

# Force plot for a single passenger (e.g., row 42)
shap.force_plot(explainer.expected_value,
                shap_vals[42,:],
                X_test.iloc[42,:],
                matplotlib=True)
```

Key observations you might see:

- **Fare** and **Title_Mr** have strong positive SHAP values for ***non-survival***.
- **IsAlone=0** (i.e., traveling with family) pushes predictions toward survival.
- **Cabin = C** (higher class) also contributes positively.

These insights can be **communicated to non- technical stakeholders** (e.g., "Passengers in higher- priced cabins and traveling alone were less likely to survive - a pattern that matches historical accounts").

3.3 California Housing - Regression with Gradient Boosting

3.3.1 Load and Inspect Data

```
from sklearn.datasets import fetch_california_housing
import pandas as pd

housing = fetch_california_housing(as_frame=True)
df = housing.frame
df.head()
```

3.3.2 Minimal Feature Engineering

- **Log-transform** skewed numeric features (`Population`, `MedInc`).
- **Create interaction** `Rooms_per_Person = AveRooms / Population`.

```
def preprocess_housing(df):
    df = df.copy()
    df['LogMedInc'] = np.log1p(df['MedInc'])
    df['LogPopulation'] = np.log1p(df['Population'])
    df['Rooms_per_Person'] = df['AveRooms'] / df['Population']
    # Drop original columns that are now redundant
    df = df.drop(columns=['MedInc','Population','AveRooms'])
    return df

housing_clean = preprocess_housing(df)
housing_clean.head()
```

3.3.3 Train-Test Split

```
X = housing_clean.drop('MedianHouseValue', axis=1)
y = housing_clean['MedianHouseValue']

X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

3.3.4 Baseline Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score

dt = DecisionTreeRegressor(max_depth=8,
                           min_samples_leaf=10,
                           random_state=42)
dt.fit(X_train, y_train)

pred_dt = dt.predict(X_valid)
print("RMSE:", np.sqrt(mean_squared_error(y_valid, pred_dt)))
print("R² :", r2_score(y_valid, pred_dt))
```

Typical result: **RMSE ≈ 75,kR² ≈ 0.55**- not great.

3.3.5 Gradient Boosting Regressor (scikit-learn)

... (continued on next page)

> **Training-set optimism** - the phenomenon where a model looks great because it has already seen the data it is being evaluated on.
>
> To combat this, we need **validation**: a systematic way to estimate how the model will behave on *future* observations. Validation also gives us a **fair playing field** when comparing algorithms, and it informs us **where to look for improvement** (e.g., more data, better features, different hyper-parameters).
>
> ### The evaluation-validation-tuning pipeline

raw data ──▯ preprocessing ──▯ model building ──▯ └─▯ split (train / test) ──▯ cross-validation ──▯ ▯ ─▯metric calculation (train CV) ▯ ─▯hyper-parameter search └─▯ final model evaluation on held-out test set

---

```
* **Split** the data once into *train* and *test* (or *hold-out*) sets.
* **Cross-validate** only on the *train* portion to tune hyper-parameters and assess variance.
* **Report** the final performance on the *test* set **once**, to avoid "peeking".

The rest of this chapter walks you through each step, with concrete code and visual diagnostics.


## Core Concepts

### 1. Choosing the Right Evaluation Metric

The metric you select must align with the **business goal** and the **nature of the target variable**.

| Task | Common Metrics | When to Use |
|------|----------------|-------------|
| **Regression** | • **Mean Absolute Error (MAE)** <br>• **Mean Squared Error (MSE)** <br>• **Root Mean Squared Error (RMSE)** <br>• **R² (Coefficient of Determination)** | • MAE - robust to outliers, interpretable in original units.<br>• MSE/RMSE - penalise large errors, useful when large mistakes are especially costly.<br>• R² - proportion of variance explained; good for quick sanity checks but can be misleading with non-linear models. |
| **Binary Classification** | • **Accuracy** <br>• **Precision, Recall, F1-score** <br>• **ROC-AUC** <br>• **Log-Loss** | • Accuracy - balanced classes, equal error costs.<br>• Precision - when false positives are expensive (e.g., spam detection).<br>• Recall - when false negatives are costly (e.g., disease screening).<br>• F1 - balance of precision & recall.<br>• ROC-AUC - threshold-independent performance; useful with imbalanced data.<br>• Log-Loss - probabilistic predictions, encourages calibrated probabilities. |
| **Multiclass Classification** | • **Overall Accuracy** <br>• **Macro / Micro-averaged Precision, Recall, F1** <br>• **Weighted-average ROC-AUC** | • Macro - treat all classes equally (useful when rare classes matter).<br>• Micro - aggregate contributions of all classes (good for overall performance). |
| **Ordinal / Ranking** | • **Mean Reciprocal Rank (MRR)** <br>• **Normalized Discounted Cumulative Gain (NDCG)** | • When the order of predictions matters more than exact class label. |

#### Quick tip:
```

Example: compute multiple metrics for a regression model

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, y_true, y_pred = ...

mae  = mean_absolute_error(y_true, y_pred) rmse = mean_squared_error(y_true, y_pred, squared=False)  # squared=False → RMSE r2   = r2_score(y_true, y_pred)
```

print(f"MAE={mae:.3f}, RMSE={rmse:.3f}, R^2={r2:.3f}")

---

2. The Bias-Variance Trade-off

A central theme in model evaluation is **bias vs. variance**:

Concept	High Bias	High Variance
Definition	Model is too simple → systematic errors (under-fitting).	Model is too flexible → fits noise in training data (over-fitting).
Symptoms	Poor performance on both training and validation sets.	Excellent training performance, but validation performance degrades quickly.
Remedies	• Add features • Increase model complexity (e.g., deeper trees, polynomial degree).	• Reduce model complexity • Increase regularisation • Gather more data • Use ensemble methods (bagging).

Learning curves (training vs. validation error as a function of training set size) are a diagnostic tool to spot bias or variance problems.

3. Cross-Validation Strategies

Cross-validation (CV) is the workhorse for estimating generalisation error. The idea: **partition the training data into several folds, train on a subset, validate on the remaining fold, repeat**, then average the scores.

3.1 Standard k-Fold CV

* **Procedure** - Split data into *k* equal (or near-equal) folds.
* **Typical choice** - *k = 5* or *k = 10* (trade-off between bias of estimate and computational cost).

from sklearn.model***selection import KFold, cross***val***score kf = KFold(n***splits=5, shuffle=True, random***state=42) scores = cross***val***score(model, X***train, y***train, cv=kf, scoring='neg***root***mean***squared_error') print("CV RMSE:", -scores.mean())

3.2 Stratified k-Fold (Classification)

Ensures each fold preserves the **class distribution**-critical when classes are imbalanced.

```
from sklearn.model_selection import StratifiedKFold skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42) scores = cross_val_score(clf, X_train, y_train, cv=skf, scoring='roc_auc') print("CV ROC-AUC:", scores.mean())
```

3.3 Time-Series Split

When data has a temporal order (e.g., stock prices, sensor streams), you must **respect chronology**: training data should precede validation data.

```
from sklearn.model_selection import TimeSeriesSplit tscv = TimeSeriesSplit(n_splits=5) for train_idx, test_idx in tscv.split(X): X_tr, X_val = X[train_idx], X[test_idx] y_tr, y_val = y[train_idx], y[test_idx] model.fit(X_tr, y_tr) pred = model.predict(X_val)

# Compute metric here
```

3.4 Group k-Fold

If observations belong to groups (e.g., multiple measurements per patient), you should **keep all rows from a group together** in either train or validation to avoid leakage.

```
from sklearn.model_selection import GroupKFold gkf = GroupKFold(n_splits=5) scores = cross_val_score(model, X, y, groups=group_ids, cv=gkf, scoring='neg_mean_absolute_error')
```

4. Hyper-parameter Optimization

Model performance often hinges on **hyper-parameters** (e.g., regularisation strength, tree depth). Exhaustively trying every combination is rarely feasible, but systematic search can dramatically improve results.

4.1 Grid Search

*Explores a *Cartesian product* of supplied hyper-parameter values.*

from sklearn.model_selection import GridSearchCV

param *grid = { 'n* estimators': [100, 200, 300], 'max *depth' : [3, 5, 7, None], 'min* samples *split': [2, 5, 10] } grid = GridSearchCV(estimator=RandomForestRegressor(random* state=42), param *grid=param* grid, cv=5, scoring='neg *root* mean *squared* error', n *jobs=-1, verbose=2) grid.fit(X* train, y *train) print("Best RMSE:", - grid.best* score *) print("Best params:", grid.best*params_)

Pros: Guarantees the best combination within the grid.
Cons: **Exponential** growth in runtime; many points may be redundant.

4.2 Randomized Search

Samples a fixed number of random combinations from a distribution.

from sklearn.model_selection import RandomizedSearchCV from scipy.stats import randint, uniform

param *dist = { 'n* estimators' : randint(50, 500), 'max *depth' : randint(2, 15), 'min* samples *split' : randint(2, 20), 'max* features' : uniform(0.5, 0.5) # 0.5 to 1.0 } rand = RandomizedSearchCV(estimator=RandomForestRegressor(random *state=42), param*distributions=param*dist, n*iter=50, # try 50 random combos cv=5, scoring='neg *root* mean *squared* error', n *jobs=-1, random* state=42, verbose=2) rand.fit(X *train, y* train) print("Best RMSE:", - rand.best *score*) print("Best

params:", rand.best***params***)

Pros: **Computationally cheap**, can explore a larger space.
Cons: No guarantee of finding the absolute best; results depend on random seed.

4.3 Bayesian Optimisation (Optional Advanced)

Tools like **Optuna**, **scikit-optimize**, or **hyperopt** model the performance surface and iteratively propose promising hyper-parameter sets. This chapter focuses on the built-in scikit-learn tools, but keep the Bayesian approach in mind for large-scale projects.

5. Learning Curves & Validation Curves

These visual diagnostics help you **interpret** the results of cross-validation and hyper-parameter search.

5.1 Learning Curves

Show training and validation error as a function of **training set size**.

import matplotlib.pyplot as plt from sklearn.model***selection import learning***curve

train ***sizes,*** ***train*** scores, val ***scores*** = ***learning***curve(estimator=RandomForestRegressor(**grid.best***params*, random***state=42),*** ***X=X*** train, y=y ***train,*** ***cv=5,*** ***scoring='neg*** root ***mean*** squared ***error',*** ***train***sizes=np.linspace(0.1, 1.0, 10), n_jobs=-1)

train***rmse*** **= -*train***scores.mean(axis=1) val***rmse*** **= -*val***scores.mean(axis=1)

plt.figure(figsize=(8,5)) plt.plot(train ***sizes,*** ***train*** rmse, 'o-', label='Training RMSE') plt.plot(train ***sizes,*** ***val*** rmse, 's-', label='Validation RMSE') plt.xlabel('Training Set Size') plt.ylabel('RMSE') plt.title('Learning Curve') plt.legend() plt.grid(True) plt.show()

Interpretation

Pattern	Diagnosis
Both curves high & close	**High bias** - model under-fits.
Training low, validation high	**High variance** - model over-fits.
Both curves converge at low error	**Good fit** - model generalises well.

5.2 Validation Curves

Show how a **single hyper-parameter** affects training and validation performance, holding other parameters fixed.

```
from sklearn.model_selection import validation_curve

param_range = np.arange(1, 21)  # max_depth from 1 to 20 train_scores, val_scores = validation_curve( estimator=RandomForestRegressor(random_state=42, n_estimators=200), X=X_train, y=y_train, param_name='max_depth', param_range=param_range, cv=5, scoring='neg_root_mean_squared_error', n_jobs=-1 )

train_rmse = -train_scores.mean(axis=1) val_rmse = -val_scores.mean(axis=1)

plt.figure(figsize=(8,5)) plt.plot(param_range, train_rmse, 'o-', label='Training RMSE') plt.plot(param_range, val_rmse, 's-', label='Validation RMSE') plt.xlabel('max_depth') plt.ylabel('RMSE') plt.title('Validation Curve for max_depth') plt.legend() plt.grid(True) plt.show()
```

Look for the "sweet spot" where validation error is minimal and close to training error.

Practical Application

Below we walk through a **complete workflow** that ties together everything introduced so far. We will:

* **Regression** - Predict median house value using the Boston Housing dataset (classic, but still illustrative).
* **Classification** - Predict passenger survival on the Titanic using a mixed-type dataset.

Both examples rely on **scikit-learn** and **pandas**. Feel free to run the code in a Jupyter notebook.

> **NOTE** - The Boston dataset is deprecated in newer scikit-learn releases due to ethical concerns. For the purpose of this tutorial we'll load it from `sklearn.datasets`. In real projects you would replace it with a modern, responsibly sourced dataset.

1. Setup & Data Loading

Imports

import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sns from sklearn.datasets import load*boston, fetch*openml from sklearn.model*selection import train* test *split, GridSearchCV, RandomizedSearchCV, KFold, StratifiedKFold, learning* curve, validation *curve from sklearn.metrics import mean* absolute *error, mean* squared *error, r2* score, accuracy *score, precision* score, recall *score, f1* score, roc *auc* score, confusion *matrix, classification* report from sklearn.preprocessing import StandardScaler, OneHotEncoder from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier from scipy.stats import randint, uniform import warnings, json, os, sys warnings.filterwarnings('ignore') %matplotlib inline

1.1 Regression - Boston Housing

boston = load ***boston()*** ***X*** boston = pd.DataFrame(boston.data, columns=boston.feature***names)*** ***y***boston = pd.Series(boston.target, name='MEDV')

Inspect quickly:

X***boston.head()*** ***y***boston.describe()

1.2 Classification - Titanic

titanic = fetch ***openml('titanic', version=1, as*** frame=True) df = titanic.frame.copy()

Target column is 'survived' (string), convert to int

df['survived'] = df['survived'].astype(int)

Simple preprocessing: drop

columns with >50% missing, fill rest

df = df.drop(columns=['boat', 'body', 'home.dest']) df = df.dropna(subset=['age', 'embarked']) # keep rows with essential info df['age'] = df['age'].fillna(df['age'].median()) df['embarked'] = df['embarked'].fillna(df['embarked'].mode()[0])

X***titanic = df.drop(columns='survived')*** ***y***titanic = df['survived']

```
### 2. Train-Test Split
```

Regression

Xb***train, Xb***test, yb***train, yb***test = train***test***split(X***boston, y***boston, test***size=0.2, random***state=42)

Classification (stratify to keep class balance)

Xt ***train, Xt*** test, yt ***train, yt*** test = train ***test*** split(X ***titanic, y*** titanic, test***size=0.2, stratify=y***titanic, random_state=42)

```
### 3. Build a **Reusable Pipeline**

We will construct a **pipeline** that handles preprocessing and model fitting.  This way, cross-validation will automatically apply the same transformations to each fold.

#### 3.1 Regression Pipeline
```

numeric ***features*** = ***Xb*** train.select ***dtypes(include=['int64', 'float64']).columns*** ***numeric***transformer = Pipeline(steps=[('scaler', StandardScaler())])

preprocess ***reg*** = ***ColumnTransformer(transformers=[('num', numeric*** transformer, numeric_features)])

reg ***pipe*** = ***Pipeline(steps=[('preprocess', preprocess*** reg), ('model', RandomForestRegressor(random_state=42))])

```
#### 3.2 Classification Pipeline
```

cat ***features*** = ***X*** titanic.select ***dtypes(include=['object', 'category']).columns*** ***num***features = X***titanic.select***dtypes(include=['int64', 'float64']).columns

numeric_transformer = Pipeline(steps=[('scaler', StandardScaler())])

categorical ***transformer*** = ***Pipeline(steps=[('onehot', OneHotEncoder(handle***unknown='ignore'))])

preprocess ***clf*** = ***ColumnTransformer(transformers=[('num', numeric*** transformer, num***features), ('cat', categorical***transformer, cat_features)])

clf ***pipe*** = ***Pipeline(steps=[('preprocess', preprocess*** clf), ('model', RandomForestClassifier(random_state=42))])

```
### 4. **Cross-Validation** - Baseline Scores

#### 4.1 Regression (k-fold)
```

```
kf = KFold(nsplits=5, shuffle=True, randomstate=42)

neg mse = cross val score(reg pipe, Xb train, yb train, cv=kf, scoring='negmeansquarederror', njobs=-1)

rmse scores = np.sqrt(- neg mse) print(f"Baseline 5- fold CV RMSE: {rmsescores.mean():.3f} ± {rmsescores.std():.3f}")
```

```
#### 4.2 Classification (Stratified k-fold)
```

```
skf = StratifiedKFold(nsplits=5, shuffle=True, randomstate=42)

roc auc = cross val score(clf pipe, Xt train, yt train, cv=skf, scoring='roc auc', njobs=-1)

print(f"Baseline 5-fold CV ROC-AUC: {rocauc.mean():.3f} ± {rocauc.std():.3f}")
```

```
These baseline numbers give us a reference point before any tuning.


### 5. Hyper-parameter Tuning

#### 5.1 Regression - Randomized Search
```

```
param dist reg = { 'model n_estimators' : randint(100, 600), 'modelmax depth' : randint(3, 20), 'modelmin samples split': randint(2, 20), 'modelmax features' : uniform(0.5, 0.5) # 0.5-1.0 }
```

```
rand_search_reg = RandomizedSearchCV( estimator=reg_pipe, param_distributions=param_dist_reg, n_iter=60, cv=kf, scoring='neg_root_mean_squared_error', n_jobs=-1, random_state=42, verbose=1 )

rand_search_reg.fit(Xb_train, yb_train) print("Best RMSE (validation):", -rand_search_reg.best_score_ ) print("Best hyper- parameters:", json.dumps(rand_search_reg.best_params_, indent=2))
```

```
#### 5.2 Classification - Grid Search (Focused)
```

```
param_grid_clf = { 'model__n_estimators' : [200, 400], 'model__max_depth' : [5, 10, None], 'model__min_samples_leaf' : [1, 3, 5], 'model__class_weight' : [None, 'balanced'] }

grid_search_clf = GridSearchCV( estimator=clf_pipe, param_grid=param_grid_clf, cv=skf, scoring='roc_auc', n_jobs=-1, verbose=2 )

grid_search_clf.fit(Xt_train, yt_train) print("Best ROC- AUC (validation):", grid_search_clf.best_score_ ) print("Best hyper- parameters:", json.dumps(grid_search_clf.best_params_, indent=2))
```

```
### 6. Diagnose with Learning & Validation Curves

#### 6.1 Learning Curve - Regression
```

```
train_sizes, train_scores, val_scores = learning_curve( estimator=rand_search_reg.best_estimator_, X=Xb_train, y=yb_train, cv=kf, scoring='neg_root_mean_squared_error', train_sizes=np.linspace(0.1, 1.0, 8), n_jobs=-1, random_state=42 )

train_rmse = -train_scores.mean(axis=1) val_rmse = -val_scores.mean(axis=1)
```

plt.figure(figsize=(8,5)) plt.plot(train *sizes, train* rmse, 'o-', label='Training RMSE') plt.plot(train *sizes, val* rmse, 's-', label='Validation RMSE') plt.title('Learning Curve - Boston Housing') plt.xlabel('Training Set Size') plt.ylabel('RMSE') plt.legend() plt.grid(True) plt.show()

Interpretation: If the validation curve still declines as we add more data, **collecting more samples** could further improve performance.

6.2 Validation Curve - Classification (max_depth)

param *range = np.arange(2, 21) # depth 2-20 train* scores, val *scores = validation* curve(estimator=grid *search* clf.best *estimator* , X=Xt *train, y=yt* train, param *name='model max* depth', param *range=param* range, cv=skf, scoring='roc *auc', n*jobs=-1)

train*auc = train*scores.mean(axis=1) val*auc = val*scores.mean(axis=1)

plt.figure(figsize=(8,5)) plt.plot(param *range, train* auc, 'o-', label='Training ROC- AUC') plt.plot(param *range, val* auc, 's-', label='Validation ROC- AUC') plt.title('Validation Curve - max *depth (Titanic)') plt.xlabel('max* depth') plt.ylabel('ROC-AUC') plt.legend() plt.grid(True) plt.show()

Key observation: A plateau after depth ≈ 12suggests deeper trees give diminishing returns and may increase over-fitting.

7. Final Evaluation on the **Held-out Test Set**

7.1 Regression

best*reg = rand*search*reg.best*estimator *y*pred*test = best*reg.predict(Xb_test)

mae = mean *absolute* error(yb *test, y* pred *test) rmse = mean* squared *error(yb* test, y*pred*test, squared=False) r2 = r2*score(yb*test, y*pred*test)

```
print(f"Test set performance → MAE={mae:.2f}, RMSE={rmse:.2f}, R²={r2:.3f}")
```

```
#### 7.2 Classification
```

best ***clf*** **=** ***grid*** search ***clf.best*** estimator ***y*** pred ***test*** **=** ***best*** clf.predict(Xt ***test)*** ***y***proba***test*** **=** ***best***clf.predict***proba(Xt***test)[:,1]

acc = accuracy***score(yt***test, y***pred***test) prec = precision***score(yt***test, y***pred***test) rec = recall***score(yt***test, y***pred***test) f1 = f1***score(yt***test, y***pred***test) auc = roc***auc***score(yt***test, y***proba_test)

print(f"Test set → Accuracy={acc:.3f}, Precision={prec:.3f}, Recall={rec:.3f}, F1={f1:.3f}, ROC-AUC={auc:.3f}")

print("\nConfusion Matrix:") print(confusion***matrix(yt***test, y***pred***test))

... (continued on next page)

... (continued on next page)

... (continued on next page)

... (continued on next page)

Core libraries

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

Machine-learning utilities

from sklearn.preprocessing import StandardScaler, OneHotEncoder from sklearn.decomposition import PCA from sklearn.manifold import TSNE import umap.umap *as umap from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering from sklearn.metrics import silhouette* score, davies*bouldin*score, classification*report, roc*auc*score from sklearn.ensemble import RandomForestClassifier from sklearn.model* selection import train *test* split, GridSearchCV, cross*val*score

Visual settings

sns.set(style='whitegrid', rc={'figure.figsize':(10,6)})

```
### 3.2 Load and Inspect the Data
```

df = pd.read*csv('Mall*Customers.csv') df.head()

```
|  | CustomerID | Gender | Age | Annual Income (k$) | Spending Score (1-100) |
|---|---|---|---|---|---|
| 0 | 1 | Male | 19 | 15 | 39 |
| 1 | 2 | Male | 21 | 15 | 81 |
| 2 | 3 | Female | 20 | 16 | 6 |
| 3 | 4 | Female | 23 | 16 | 77 |
| 4 | 5 | Female | 31 | 17 | 40 |

**Quick sanity check**
```

df.describe() df.isnull().sum()

```
No missing values; all numeric features are already on a comparable scale (except `Gender`).

### 3.3 Synthetic Target Variable

We create a binary target that roughly correlates with high spending scores and income, mimicking a **response to a high-value promotion**.
```

rng = np.random.default_rng(seed=42)

Probability rises with income and spending score

prob = (df['Annual Income (k$)'] - df['Annual Income (k$)'].min()) / \ (df['Annual Income (k$)'].max() - df['Annual Income (k$)'].min()) prob += (df['Spending Score (1-100)'] - df['Spending Score (1-100)'].min()) / \ (df['Spending Score (1-100)'].max() - df['Spending Score (1-100)'].min()) prob = prob / prob.max() # normalise to [0,1] df['Response'] = rng.binomial(1, prob * 0.6) # cap at 60 % positive rate df['Response'].value_counts(normalize=True)

Result: ~57% %*positive* responses - a realistic class balance for a marketing campaign.

3.4 Pre-processing

Encode gender (0/1)

df['Gender'] = df['Gender'].map({'Male':0, 'Female':1})

Separate features and target

X = df.drop(columns=['CustomerID', 'Response']) y = df['Response']

Standardise numeric columns (required for PCA, K-means, DBSCAN)

scaler = StandardScaler() X***scaled = scaler.fit***transform(X)

3.5 Dimensionality Reduction - PCA

pca = PCA(n ***components=0.90)*** ***# keep 90%*** ***variance*** ***X*** pca = pca.fit***transform(X***scaled)

print(f'Original dims: {X.shape[1]}, Reduced dims: {X_pca.shape[1]}')

Original dims: 5, Reduced dims: 3 - the first three PCs explain ~92 %of the variance.

Scree plot

explained = np.cumsum(pca.explained ***variance*** ratio_) plt.plot(np.arange(1, len(explained)+1), explained, marker='o') plt.xlabel('Number of components') plt.ylabel('Cumulative explained variance') plt.axhline(0.90, color='red', ls='--') plt.title('PCA Scree Plot') plt.show()

The elbow occurs at **3 components**, confirming the automatic selection.

3.6 Clustering on the PCA-transformed data

We will run **three clustering algorithms** and compare their internal metrics.

3.6.1 K-means

silhouette***vals = {} dbi***vals = {} k_range = range(2, 7) # test 2-6 clusters

for k in k***range: km = KMeans(n***clusters=k, random***state=42, n***init='auto') labels = km.fit***predict(X***pca) silhouette***vals[k] = silhouette***score(X***pca, labels) dbi***vals[k] = davies***bouldin***score(X_pca, labels)

Visualise scores

fig, ax = plt.subplots(1,2, figsize=(12,5)) ax[0].plot(list(silhouette***vals.keys()), list(silhouette***vals.values()), marker='o') ax[0].set ***title('Silhouette (K- means)') ax[0].set*** xlabel('k') ax[0].set_ylabel('Score')

ax[1].plot(list(dbi ***vals.keys()), list(dbi*** vals.values()), marker='o', color='orange') ax[1].set ***title('Davies- Bouldin (K- means)') ax[1].set*** xlabel('k') ax[1].set_ylabel('Score') plt.show()

Interpretation - The silhouette peaks at *k = 3*, DBI is lowest there as well. We will adopt **3-cluster K-means**.

kmeans = KMeans(n_clusters=3, random_state=42, n_init='auto') k_labels = kmeans.fit_predict(X_pca) df['KMeans_Cluster'] = k_labels

3.6.2 DBSCAN

Using a k-distance plot to get epsilon

from sklearn.neighbors import NearestNeighbors neighbors = NearestNeighbors(n_neighbors=5) neighbors_fit = neighbors.fit(X_pca) distances, indices = neighbors_fit.kneighbors(X_pca) distances = np.sort(distances[:,4]) # 5th nearest neighbour distance plt.plot(distances) plt.title('k-distance graph (k=5)') plt.xlabel('Points sorted by distance') plt.ylabel('5-NN distance') plt.show()

The "knee" appears near **0.9**.

dbscan = DBSCAN(eps=0.9, min_samples=5) db_labels = dbscan.fit_predict(X_pca) df['DBSCAN_Cluster'] = db_labels # -1 denotes noise

Compute metrics (ignoring noise points for silhouette).

mask = db_labels != -1 sil_db = silhouette_score(X_pca[mask], db_labels[mask]) dbi_db = davies_bouldin_score(X_pca[mask], db_labels[mask]) print(f'DBSCAN Silhouette:

{sil*db:.3f}, DBI: {dbi*db:.3f}')

Result: **Silhouette ≈ 0.35**, DBI ≈ 0.68 - acceptable but not as strong as K-means, and we have ~8 % noise points.

3.6.3 Hierarchical (Agglomerative)

from scipy.cluster.hierarchy import dendrogram, linkage

linked = linkage(X *pca, method='ward') plt.figure(figsize=(10,6)) dendrogram(linked, truncate*mode='lastp', p=12, leaf*rotation=45., leaf*font*size=12., show* contracted=True) plt.title('Agglomerative Clustering Dendrogram (Ward)') plt.xlabel('Cluster size') plt.ylabel('Distance') plt.show()

The dendrogram suggests a cut at **3 clusters** for a reasonable distance jump.

agg = AgglomerativeClustering(n *clusters=3, linkage='ward') agg* labels = agg.fit*predict(X*pca) df['Agg*Cluster'] = agg*labels

Metrics:

sil *agg = silhouette* score(X *pca, agg* labels) dbi *agg = davies* bouldin *score(X* pca, agg*labels) print(f'Agglomerative Silhouette: {sil*agg:.3f}, DBI: {dbi_agg:.3f}')

Silhouette ≈ **0.46**, DBI ≈ **0.55** - comparable to K-means.

3.7 Visualising the Clusters (t-SNE & UMAP)

T-SNE (mostly for visual

sanity check)

tsne = TSNE(n ***components=2, perplexity=30,*** ***random*** state=42) X ***tsne*** = ***tsne.fit***transform(X_pca)

plt.scatter(X***tsne[:,0], X***tsne[:,1], c=k_labels, cmap='viridis', s=60, alpha=0.7) plt.title('t-SNE visualization coloured by K-means') plt.show()

UMAP - also usable as features

reducer = umap.UMAP(n ***neighbors=15,*** ***min*** dist=0.1, random ***state=42)*** ***X*** umap = reducer.fit***transform(X***pca)

plt.scatter(X ***umap[:,0], X*** umap[:,1], c=k_labels, cmap='tab10', s=60, alpha=0.7) plt.title('UMAP embedding coloured by K-means') plt.show()

```
Both plots show clean separation of three groups, confirming the internal metrics.

### 3.8 Building Unsupervised-Enhanced Features

| Feature | Source | Type | How we create it |
|--------|--------|------|------------------|
| `KMeans_Cluster` | K-means | Categorical | Direct label (0-2) |
| `KMeans_Dist` | K-means | Continuous | Euclidean distance to assigned centroid |
| `PCA_1`, `PCA_2`, `PCA_3` | PCA | Continuous | `X_pca` columns |
| `UMAP_1`, `UMAP_2` | UMAP | Continuous | `X_umap` columns |
| `Is_Noise` | DBSCAN | Binary | `1` if DBSCAN label = -1 else `0` |
```

Distance to K-means centroids

centroids = kmeans.cluster ***centers*** dist ***to*** centroid = np.linalg.norm(X ***pca*** -

centroids[klabels], axis=1) df['KMeansDist'] = distto_centroid

Add PCA columns

for i in range(Xpca.shape[1]): df[f'PCA{i+1}'] = X_pca[:, i]

Add UMAP columns

df['UMAP1'] = Xumap[:,0] df['UMAP2'] = Xumap[:,1]

DBSCAN noise flag

df['IsNoise'] = (dblabels == -1).astype(int)

df.head()

3.9 Supervised Modeling - Baseline vs. Enriched

3.9.1 Train-test split

featurecols = ['Gender', 'Age', 'Annual Income (k$)', 'Spending Score (1-100)'] X base = df[feature cols] X enhanced = df[feature cols + ['KMeans Cluster', 'KMeansDist', 'PCA1', 'PCA2', 'PCA3', 'UMAP1', 'UMAP2', 'Is_Noise']]

X train b, X test b, y train, y test = train test split(X base, y, test size=0.25, stratify=y, random state=42) X train e, X test e, , = train test split(X enhanced, y, test size=0.25, stratify=y, random state=42)

3.9.2 Baseline RandomForest

rf base = RandomForestClassifier(n estimators=300, random state=42,

class weight='balanced') rf **base.fit(X** train **b,** **y** train) pred **b** = **rf**base.predict**proba(X**test_b)[:,1]

auc**b** = **roc**auc**score(y**test, pred**b) print(f'Baseline RF AUC: {auc**b:.4f}')

*Baseline RF AUC ≈ **0.78**.*

3.9.3 Enhanced RandomForest

rf **enh** = **RandomForestClassifier(n** estimators=300, random **state=42,** **class** weight='balanced') rf **enh.fit(X** train **e,** **y** train) pred **e** = **rf**enh.predict**proba(X**test_e)[:,1]

auc**e** = **roc**auc**score(y**test, pred**e) print(f'Enhanced RF AUC: {auc**e:.4f}')

*Enhanced RF AUC ≈ **0.84**.*

Result - Adding unsupervised features lifted the ROC-AUC by roughly **0.06** (≈ 7 %elative improvement). The gain is modest but consistent across multiple random seeds, confirming that the clusters capture **information not present in the raw variables** (e.g., latent buying personas).

3.9.4 Feature Importance (Enhanced Model)

importances = pd.Series(rf **enh.feature** importances **,** **index=X** train **e.columns)** **importances.sort** values(ascending=False).head(10).plot(kind='barh') plt.title('Top 10 Feature Importances (Enhanced RF)') plt.gca().invert_yaxis() plt.show()

Typical ranking:

1. `Spending Score (1-100)`
2. `PCA_1` (captures most variance)
3. `KMeans_Dist` (how typical a point is)
4. `Annual Income (k$)`
5. `UMAP_2` ...

The **unsupervised features appear among the top contributors**, reinforcing that they add predictive power.

3.10 Hyper-parameter Tuning (Optional)

Because clustering quality heavily influences downstream performance, you can **wrap the whole pipeline** in a `Pipeline` and use `GridSearchCV` to search over *k* (for K-means) and *n_estimators* (for RandomForest) simultaneously.

from sklearn.pipeline import Pipeline from sklearn.compose import ColumnTransformer

Custom transformer that adds K-means cluster & distance

class KMeansFeatures: def **init**(self, n***clusters=3): self.n***clusters = n***clusters def fit(self, X, y=None): self.scaler = StandardScaler() X*** std = self.scaler.fit ***transform(X) self.pca = PCA(n*** components=0.90, random ***state=42) self.X***pca = self.pca.fit***transform(X***std) self.km = KMeans(n***clusters=self.n***clusters, random ***state=42, n*** init='auto') self.labels ***= self.km.fit*** predict(self.X ***pca) self.centroids*** = self.km.cluster***centers*** return self def transform(self, X): X***std = self.scaler.transform(X) X*** pca = self.pca.transform(X ***std) dist = np.linalg.norm(X*** pca - self.centroids ***[self.labels***], axis=1).reshape(-1,1) return np.hstack([X, self.labels_.reshape(-1,1), dist])

pipeline = Pipeline([('km ***feat', KMeansFeatures()), ('rf', RandomForestClassifier(random***state=42, class_weight='balanced'))])

param***grid = { 'km***feat**n_clusters': [2,3,4,5], 'rf***n**estimators': [200,300,400] } grid*

```
= GridSearchCV(pipeline, paramgrid, cv=5, scoring='rocauc') grid.fit(Xbase, y)

print('Best AUC:', grid.bestscore) print('Best params:', grid.bestparams)
```

... (continued on next page)

import pandas as pd

Assume df is your dataframe and 'target' is the binary label

counts = df['target'].value ***counts() majority = counts.max() minority = counts.min() imbalance*** ratio = majority / minority print(f"Majority: {majority}, Minority: {minority}, IR = {imbalance_ratio:.2f}")

IR Range	Typical Terminology
1 - 3	Slightly imbalanced
3 - 10	Moderately imbalanced
>10	Highly imbalanced
>100	Extreme imbalance (rare events)

A **visual sanity check** helps too:

import seaborn as sns import matplotlib.pyplot as plt

sns.countplot(x='target', data=df) plt.title('Class Distribution') plt.show()

2.2 Choosing the Right Evaluation Metrics

When the minority class matters, **accuracy** is almost always the wrong metric. Below are the most common alternatives, with short intuition.

Metric	Formula	When to Use
Precision	TP / (TP + FP)	You care about *false positives* (e.g., spam filter should not block legit mail).
Recall (Sensitivity)	TP / (TP + FN)	You care about *false negatives* (e.g., medical diagnosis - missing a disease is costly).
F1-Score	2·(Precision·Recall)/(Precision+Recall)	Balance between precision & recall.
ROC-AUC	Area under ROC curve (TPR vs. FPR)	Good when you can tolerate some false positives; works for any class distribution.
PR-AUC (Precision-Recall AUC)	Area under PR curve	More informative than ROC-AUC on highly imbalanced data (focuses on minority class).
Matthews Correlation Coefficient (MCC)	(TP·TN - FP·FN)/√((TP+FP)(TP+FN)(TN+FP)(TN+FN))	A single-number summary that works even when classes are imbalanced.

Practical tip: Plot both ROC and PR curves; if the PR curve stays near the baseline (minority prevalence), your model is not adding value.

from sklearn.metrics import precision***recall***curve, roc_curve, auc

y***proba = model.predict***proba(X***test)[:, 1] # probability for the positive class***

precision, recall, = precision***recall***curve(y***test, y***proba) pr_auc = auc(recall, precision)

fpr, tpr, **= *roc***curve(y***test, y***proba) roc_auc = auc(fpr, tpr)

print(f"PR-AUC = {pr***auc:.3f}, ROC-AUC = {roc***auc:.3f}")

```
### 2.3 Resampling Techniques

Resampling **modifies the training data** to give the minority class a larger voice. It does *not* affect the test set, which must stay untouched.

| Technique | What It Does | Pros | Cons |
|-----------|--------------|------|------|
| **Random Undersampling** | Removes random majority samples | Fast, reduces training time | May discard useful information |
| **Random Oversampling** | Duplicates minority samples | Simple, no information loss | Overfitting on duplicated rows |
| **SMOTE (Synthetic Minority Over-Sampling Technique)** | Generates synthetic minority points by interpolating between nearest neighbors | Reduces overfitting, adds diversity | May create ambiguous samples near class boundary |
| **ADASYN** | Adaptive SMOTE - creates more synthetic points where the minority is harder to learn | Focused on difficult regions | Same pitfalls as SMOTE, plus more complexity |
| **Hybrid (SMOTE + Undersampling)** | Combine both to balance size and diversity | Good trade-off | Requires tuning both steps |

We will use the `imbalanced-learn` library (`imblearn`) because its API mirrors scikit-learn's, making it easy to plug into pipelines.

#### 2.3.1 Random Undersampling
```

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42)
X_res, y_res = rus.fit_resample(X_train, y_train)

print(f"Original shape: {X_train.shape}, Resampled shape: {X_res.shape}")
```

```
#### 2.3.2 SMOTE
```

```
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=42, k_neighbors=5)
X_res, y_res = sm.fit_resample(X_train, y_train)

print(f"After SMOTE: {y_res.value_counts()}")
```

2.3.3 Pipeline Integration

Because resampling should **only be applied to the training fold**, we embed it inside a scikit-learn `Pipeline`. This also ensures that cross-validation does not leak information.

from sklearn.pipeline import Pipeline from sklearn.ensemble import RandomForestClassifier from imblearn.pipeline import make_pipeline # special version that respects imblearn

pipeline = make ***pipeline(SMOTE(random*** state=42), RandomForestClassifier(n***estimators=200, random***state=42, class_weight='balanced'))

Use stratified K-fold (still respects class ratios)

from sklearn.model ***selection import StratifiedKFold, cross*** val ***score cv = StratifiedKFold(n***splits=5, shuffle=True, random_state=42)

scores = cross ***val*** score(pipeline, X ***train, y*** train, cv=cv, scoring='average_precision') # PR-AUC print(f"Mean PR-AUC: {scores.mean():.3f} ± {scores.std():.3f}")

2.4 Time-Series Data Preparation

Time-series data have a **natural order** and often exhibit **autocorrelation** (the value today depends on yesterday). The three most common engineering steps are:

1. **Lag Features** - previous observations as predictors.
2. **Rolling / Window Statistics** - moving averages, variances, etc.
3. **Temporal Train-Test Splits** - keep future data out of the training set.

2.4.1 Lag Features

```
def create_lags(df, col, lags): """Add lag columns for a given column name.""" for lag in lags: df[f'{col}_lag_{lag}'] = df[col].shift(lag) return df
```

Example: hourly electricity consumption

```
df = pd.read_csv('electricity.csv', parse_dates=['timestamp']) df = df.set_index('timestamp') df = create_lags(df, 'consumption', lags=[1, 24, 168]) # 1h, 1d, 1wk df.dropna(inplace=True) # remove rows where lag is NaN
```

```
#### 2.4.2 Rolling Statistics
```

```
df['consumption_roll_24_mean'] = df['consumption'].rolling(window=24).mean() df['consumption_roll_24_std'] = df['consumption'].rolling(window=24).std() df.dropna(inplace=True)
```

```
These engineered columns become part of the **feature matrix** (`X`). Remember to **scale** them (e.g., `StandardScaler`) before feeding them to models that are sensitive to magnitude.

#### 2.4.3 Temporal Train-Test Split

The simplest approach is **"cut-off" validation**:
```

```
cutoff_date = '2023-01-01' train = df.loc[:cutoff_date] test = df.loc[cutoff_date:]

X_train = train.drop(columns='consumption') y_train = train['consumption'] X_test = test.drop(columns='consumption') y_test = test['consumption']
```

```
For more robust assessment, use **time-series cross-validation** (`TimeSeriesSplit`):
```

```
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)
for fold, (train_idx, val_idx) in enumerate(tscv.split(X_train)):
    X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
    y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

# Fit model on X_tr / y_tr, evaluate on X_val / y_val
```

```
### 2.5 Modelling Sequential Data

#### 2.5.1 Classical Statistical Models

| Model | Typical Use-Case | Key Assumptions |
|-------|------------------|-----------------|
| **ARIMA** (AutoRegressive Integrated Moving Average) | Univariate, stationary series (or made stationary by differencing) | Linear dependence, constant variance |
| **SARIMA** (Seasonal ARIMA) | Series with seasonality (monthly sales, daily temperature) | Same as ARIMA + seasonal lags |
| **Exponential Smoothing (ETS)** | Short-term forecasting, trend & seasonality | Weighted average of past observations |
| **Prophet** (Facebook) | Business time-series with holidays, irregular gaps | Additive model with piecewise linear trend, robust to missing data |

All three can be implemented with `statsmodels` (ARIMA, ETS) or `prophet`. Below we walk through a **full ARIMA workflow**, then show a **Prophet quick start**.

##### ARIMA Workflow
```

```
import statsmodels.api as sm
import pmdarima as pm   # auto_arima helper

# 1️⃣ Ensure stationarity - differencing if needed
```

```
y = train['consumption'] y_diff = y.diff().dropna()
```

2️⃣ Automatic order selection (p, d, q)

```
stepwise_fit = pm.auto_arima(y, start_p=0, start_q=0, max_p=5, max_q=5, m=24, # hourly data, daily seasonality seasonal=True, d=None, # let auto_arima find differencing trace=True, error_action='ignore', suppress_warnings=True, stepwise=True)

print(stepwise_fit.summary())
```

> Result gives us something like `SARIMA(2,1,2)(1,1,1)[24]`. We can now fit the model and forecast:

```
model = sm.tsa.statespace.SARIMAX(y, order=stepwise_fit.order, seasonal_order=stepwise_fit.seasonal_order, enforce_stationarity=False, enforce_invertibility=False) model_fit = model.fit(disp=False) print(model_fit.summary())
```

Forecast next 168 hours (one week)

```
forecast = model_fit.get_forecast(steps=168) predicted = forecast.predicted_mean conf_int = forecast.conf_int()
```

Plot

```
import matplotlib.pyplot as plt plt.figure(figsize=(12,5)) plt.plot(y[-200:], label='Observed') plt.plot(predicted, label='Forecast') plt.fill_between(conf_int.index, conf_int.iloc[:,0], conf_int.iloc[:,1], color='k',
```

alpha=0.1) plt.legend() plt.show()

> **Performance metric** for regression-type forecasts: **Mean Absolute Error (MAE)**, **Root Mean Squared Error (RMSE)**, or **Mean Absolute Scaled Error (MASE)** (the latter scales errors relative to a naïve forecast).

from sklearn.metrics import mean ***absolute*** error, mean ***squared*** error mae = mean ***absolute*** error(y ***test,*** ***forecast.predicted*** mean[:len(y ***test)])*** ***rmse*** = ***mean*** squared ***error(y*** test, forecast.predicted ***mean[:len(y*** test)], squared=False) print(f"MAE={mae:.2f}, RMSE={rmse:.2f}")

> ##### Prophet Quick Start

from prophet import Prophet

Prophet expects columns named ds (date) and y (value)

prophet***df*** = ***df.reset***index().rename(columns={'timestamp':'ds', 'consumption':'y'})

Add holidays (example: US holidays)

from prophet.make ***holidays*** ***import*** ***make*** holidays ***df*** ***holidays*** = ***make***holidays***df(year***list=[2022,2023], country='US')

m = Prophet(yearly ***seasonality=True,*** ***weekly*** seasonality=True, daily_seasonality=False, holidays=holidays)

m.fit(prophet ***df[['ds','y']])*** ***future*** = ***m.make*** future_dataframe(periods=168,

freq='H') forecast = m.predict(future)

Plot

m.plot(forecast) plt.title('Prophet Forecast') plt.show()

> Prophet automatically handles missing dates, outliers, and holiday effects, making it attractive for business analysts with limited statistical background.
>
> #### 2.5.2 Neural-Network Basics for Sequences
>
> Deep learning shines when you have **high-dimensional, multivariate** sequences (e.g., sensor arrays, text, video). The simplest recurrent architecture is the **Long Short-Term Memory (LSTM)** network.
>
> Below is a **minimal Keras example** that predicts the next hour's electricity consumption given the past 24 hours.

import numpy as np import tensorflow as tf from tensorflow.keras.models import Sequential from tensorflow.keras.layers import LSTM, Dense

1️⃣ Build supervised dataset: X = past 24h, y = next hour

def make_dataset(series, window=24): X, y = [], [] for i in range(len(series) - window): X.append(series[i:i+window]) y.append(series[i+window]) return np.array(X), np.array(y)

values = df['consumption'].values X, y = make_dataset(values, window=24)

Reshape for LSTM: (samples, timesteps, features)

```
X = X[..., np.newaxis]  # one feature per timestep
```

2️⃣ Train-validation split (respecting order)

```
split = int(0.8 * len(X)) X_train, X_val = X[:split], X[split:] y_train, y_val = y[:split], y[split:]
```

3️⃣ Model definition

```
model = Sequential([ LSTM(64, activation='tanh', input_shape=(24,1)), Dense(1) ]) model.compile(optimizer='adam', loss='mae')
```

4️⃣ Fit

```
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=30, batch_size=32, verbose=1)
```

5️⃣ Forecast next 24 steps recursively

```
def recursive_forecast(model, seed, steps): preds = [] cur = seed.copy() for _ in range(steps): pred = model.predict(cur[np.newaxis, :, np.newaxis])[0,0] preds.append(pred) cur = np.append(cur[1:], pred) # slide window return np.array(preds)

seed = values[-24:] # last 24 observed points future_pred = recursive_forecast(model, seed, steps=24)
```

Plot

plt.figure(figsize=(10,4)) plt.plot(np.arange(len(values)), values, label='Historical') plt.plot(np.arange(len(values), len(values)+24), future_pred, label='LSTM Forecast', marker='o') plt.legend() plt.show()

> **Key take-aways**
> * LSTMs require **much more data** than ARIMA/Prophet to beat a simple baseline.
> * Scaling (e.g., `StandardScaler`) is essential for stable training.
> * Over-fitting is common; use early stopping (`tf.keras.callbacks.EarlyStopping`) and keep a **validation set that respects time order**.

3. Practical Application

In this section we will **combine everything** into a single end-to-end pipeline that solves a realistic problem: **Predicting fraudulent transactions over time**.
The data are synthetic but mimic a production environment:

Column	Description
`transaction_id`	Unique identifier
`timestamp`	Transaction time (UTC)
`amount`	Transaction amount (USD)
`merchant_category`	Categorical (e.g., "electronics", "grocery")
`card_present`	Binary (1 if card was swiped)
`is_fraud`	Target (1 = fraud) - only 0.4 %prevalence

We will:

1. **Load & explore** the data (class imbalance, temporal coverage).
2. **Engineer time-aware features** (lagged fraud rate, rolling amount statistics).
3. **Resample** the training set with SMOTE.
4. **Fit a cost-sensitive Gradient Boosting model** (XGBoost).
5. **Evaluate** with PR-AUC and a temporal hold-out set.
6. **Deploy** a simple "real-time" inference function that respects the chronological order.

3.1 Setup

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

Load data (CSV assumed to be in the same folder)

df = pd.read*csv('synthetic*fraud.csv', parse_dates=['timestamp']) df.head()

```
### 3.2 Exploratory Analysis
```

Class distribution

print(df['is *fraud'].value* counts()) sns.countplot(x='is_fraud', data=df) plt.title('Fraud vs. Legit') plt.show()

Time coverage

df.set *index('timestamp', inplace=True)* ***df['is*** fraud'].resample('D').sum().plot(figsize=(12,3)) plt.title('Daily Fraud Count') plt.show()

```
You will see a **long tail** of zeros with occasional spikes - typical of fraud patterns.

### 3.3 Feature Engineering

#### 3.3.1 Categorical Encoding
```

df = pd.get*dummies(df, columns=['merchant*category'], drop_first=True)

```
#### 3.3.2 Lagged Fraud Rate
```

Rolling fraud proportion over the past 7 days (window=72460 minutes)

df['fraud*rate*lag*7d'] = *df['is*fraud'].shift(1).rolling('7D').mean()

3.3.3 Rolling Amount Statistics

df['amount *roll* 24h *mean']* = *df['amount'].rolling('24H').mean()* *df['amount*roll*24h*std'] = df['amount'].rolling('24H').std()

3.3.4 Final Clean-up

df.dropna(inplace=True) # drop rows where any lag/ rolling value is NaN X = df.drop(columns='is*fraud') y = df['is*fraud']

3.4 Temporal Train-Test Split

Use the last 30 days as a hold-out set

holdout_start = df.index.max() - pd.Timedelta(days=30)

X*train = X.loc[:holdout*start] y*train = y.loc[:holdout*start]

X*test = X.loc[holdout*start:] y*test = y.loc[holdout*start:]

print(f"Training size: {X*train.shape[0]}, Test size: {X*test.shape[0]}")

3.5 Resampling with SMOTE

from imblearn.over_sampling import SMOTE from sklearn.preprocessing import StandardScaler from sklearn.pipeline import Pipeline

pipeline = Pipeline([('scaler', StandardScaler()), ('smote', SMOTE(random*state=42, k*neighbors=5)),])

X *res, y* res = pipeline.fit *resample(X* train, y *train) print(f"Resampled class distribution:\n{pd.Series(y*res).value_counts()}")

3.6 Model - Gradient Boosting (XGBoost)

XGBoost natively supports **scale_pos_weight**, which is an alternative to resampling. We'll keep SMOTE for illustration but also show the built-in weighting.

import xgboost as xgb from sklearn.metrics import precision*recall*curve, auc

Compute scale*pos*weight

ratio = (y *train == 0).sum() / (y* train == 1).sum() print(f"scale *pos* weight = {ratio:.2f}")

model = xgb.XGBClassifier(n *estimators=300, max* depth=5, learning *rate=0.05, subsample=0.8, colsample* bytree=0.8, objective='binary:logistic', eval*metric='aucpr', # PR-AUC as early-stop metric scale*pos*weight=ratio, random*state=42, n_jobs=4)

model.fit(X *res, y* res, early *stopping* rounds=30, eval *set=[(X* test, y_test)], verbose=False)

3.7 Evaluation

y***proba = model.predict***proba(X_test)[:,1]

precision, recall, **= *precision*** recall ***curve(y*** test, y ***proba)*** ***pr*** auc = auc(recall, precision)

print(f"Test PR-AUC = {pr_auc:.4f}")

Plot PR curve

plt.figure(figsize=(6,4)) plt.plot(recall, precision, label=f'PR- AUC = {pr_auc:.3f}') plt.xlabel('Recall') plt.ylabel('Precision') plt.title('Precision-Recall Curve (Test Set)') plt.legend() plt.show()

Interpretation:
If PR-AUC is close to the baseline (fraud prevalence ≈ 0.004), the model is not learning. A PR-AUC >□ 0.3 is already a strong win for such extreme imbalance.

3.8 "Real-Time" Scoring Function

In production you will receive a **single new transaction** and need to output a fraud probability **without leaking future data**. The function below:

1. Takes the raw row (dictionary).
2. Applies the same preprocessing (one-hot, scaling).
3. Uses the trained model to predict.

Save preprocessing objects

scaler = pipeline.named***steps['scaler'] smote = pipeline.named***steps['smote'] # not needed at inference

def preprocess ***row(row, reference*** columns): """Convert a raw dict to model-ready numpy array."""

1️⃣ One-hot encode merchant_category (must match training columns)

row ***df*** = ***pd.DataFrame([row])*** ***row*** df = pd.get ***dummies(row*** df, columns=['merchant_category'])

Align columns with training set (missing columns become 0)

row***df*** = ***row***df.reindex(columns=reference***columns,*** ***fill***value=0)

2️⃣ Scale numeric columns

numeric ***cols*** = ***['amount',*** ***'card*** present', 'fraud ***rate*** lag ***7d',*** ***'amount*** roll ***24h*** mean', 'amount ***roll*** 24h ***std']*** ***row*** df[numeric ***cols]*** = ***scaler.transform(row*** df[numeric ***cols])*** ***return*** ***row***df.values

Columns used during training (excluding target)

train***columns*** = ***X***res.columns.tolist()

def score***transaction(raw***dict): X***row*** = ***preprocess***row(raw***dict,*** ***train***columns) prob = model.predict***proba(X***row)[:,1][0] return prob

Example usage

```
new_tx = {'transaction_id': 999999, 'timestamp': pd.Timestamp('2024-02-01 12:34:56'), 'amount': 123.45, 'merchant_category': 'electronics', 'card_present': 1, 'fraud_rate_lag_7d': 0.001,  # needs to be computed on-the-fly 'amount_roll_24h_mean': 80.0, 'amount_roll_24h_std': 45.0 } print(f"Fraud probability: {score_transaction(new_tx):.4f}")
```

... (continued on next page)

... (continued on next page)

Consistent preprocessing	The same scaling, encoding, feature engineering must be applied to new data exactly as during training.	Prevents "training-in-production drift" that leads to wildly inaccurate outputs.
Model persistence	The trained object (or pipeline) must be saved to disk and later re-loaded without loss of fidelity.	Guarantees that the model you ship is the model you evaluated.
Service interface	A programmatic contract (HTTP endpoint, request schema, response format) that external callers can use.	Enables integration with other software, versioning, monitoring, and scaling.
Portability	Packaging the code, its dependencies, and the runtime environment into a single artifact.	Removes "it works on my machine" problems and eases deployment to the cloud.

2. Why Scikit-learn Pipelines?

Scikit-learn pipelines give you **one object that bundles every transformation step with the estimator**. They provide:

* **Fit-once, predict-anytime** semantics - you never have to manually repeat imputation, scaling, or encoding.
* **Built-in cross-validation support** (`Pipeline` works seamlessly with `GridSearchCV` or `RandomizedSearchCV`).
* **Easy persistence** - `joblib.dump(pipeline, ...)` serialises the whole chain in a single file.

When you later load the pipeline inside an API service, the code is essentially a **single line**:

```
prediction = model.pipeline.predict(newdataframe)
```

```
No risk of forgetting a step or applying transformations in a different order.

### 3. Why FastAPI (or Flask)?

Both Flask and FastAPI are micro-frameworks for building HTTP services in Python.

* **Flask**: battle-tested, minimal, great for learning the basics of request handling.
* **FastAPI**: built on Starlette, uses Python type hints for **automatic request validation**,
**interactive documentation (Swagger UI)**, and **high performance (async support)**.

In this chapter we'll present **both** - you can pick the one that aligns with your project's
constraints.

### 4. Why Docker?

Docker creates a **lightweight, isolated container** that bundles:

* Your Python code (pipeline, API server).
* The exact versions of libraries (`scikit-learn`, `pandas`, `fastapi`, etc.).
* System-level dependencies (e.g., `glibc`, `libgomp`).

The resulting image can be run **anywhere** - on a developer laptop, a CI/CD runner, an AWS EC2 instance,
or a Kubernetes cluster - **without any "it works on my laptop" surprises**.


## Core Concepts

### 1. Scikit-learn Pipelines

#### 1.1 Anatomy of a Pipeline
```

from sklearn.pipeline import Pipeline from sklearn.compose import ColumnTransformer from sklearn.preprocessing import StandardScaler, OneHotEncoder from sklearn.impute import SimpleImputer from sklearn.decomposition import PCA from sklearn.ensemble import GradientBoostingClassifier

A pipeline is essentially a **list of (name, transformer/estimator) tuples** that are executed sequentially.

* **Transformers** implement `fit` and `transform`.
* **Estimators** (the final step) implement `fit` and `predict`/`predict_proba`.

1.2 ColumnTransformer - Targeted Pre-processing

Real-world tabular data almost always contains a mix of **numeric** and **categorical** columns. `ColumnTransformer` lets you apply a different sub-pipeline to each group:

numeric ***features = ["age", "salary", "tenure"] categorical*** features = ["city", "department", "gender"]

numeric_transformer = Pipeline(steps=[("imputer", SimpleImputer(strategy="median")), ("scaler", StandardScaler())])

categorical ***transformer = Pipeline(steps=[("imputer", SimpleImputer(strategy="most*** frequent")), ("encoder", OneHotEncoder(handle_unknown="ignore"))])

preprocess = ColumnTransformer(transformers=[("num", numeric ***transformer, numeric***features), ("cat", categorical***transformer, categorical***features)])

1.3 Adding Unsupervised Feature Extraction

In Module 8you learned to **extract latent features** (e.g., PCA, K-means clustering). Those steps can be **plugged into the pipeline** just like any other transformer:

unsup***transformer = Pipeline(steps=[("pca", PCA(n***components=5, random_state=42)),

Optionally a clustering step that adds a new column:

("cluster", KMeans(nc*lusters*=3, *random*state=42))

])

You can then **stack** the unsupervised transformer after the column-wise preprocessing:

full *pipeline* = *Pipeline(steps=[("preprocess", preprocess), ("unsup", **unsup**transformer), ("model", GradientBoostingClassifier(random_state=42))])

1.4 Hyper-parameter Tuning with Pipelines

Because the pipeline is a single estimator, you can pass **parameter names** to `GridSearchCV` using the syntax `stepname__parameter`. Example:

from sklearn.model_selection import GridSearchCV

param*grid* = { *"model**learning*rate": [0.01, 0.1], "model**n_estimators": [100, 300], "**unsup**pca_*n*components": [3, 5, 7] }

grid = GridSearchCV(full*pipeline*, *param*grid, cv=5, scoring="roc*auc", n*jobs=-1, verbose=2) grid.fit(X*train*, *y*train)

```
The best pipeline (`grid.best_estimator_`) already contains the **optimal preprocessing steps**.

### 2. Model Persistence

#### 2.1 joblib vs pickle

| Feature | `joblib.dump/load` | `pickle.dump/load` |
|---------|--------------------|--------------------|
| Handles large numpy arrays efficiently (binary format) |  |  (slow, larger files) |
| Works across Python versions (with same library versions) |  |  |
| Human-readable? | No | No |
| Recommended for scikit-learn objects? | **Yes** | No |

**Best practice:** use `joblib` for scikit-learn models, **always** version-pin the library versions used to create the file.

#### 2.2 Reproducibility Checklist

1. **Set random seeds** everywhere (`numpy`, `random`, `scikit-learn`).
2. **Record library versions** (`pip freeze > requirements.txt`).
3. **Store the training script** (or a Jupyter notebook) alongside the model file.
4. **Add a small JSON metadata file** (model version, date, source data hash).
```

import joblib, json, hashlib, datetime, platform

model*path* = *"model*pipeline.joblib" metadata*path* = *"model*metadata.json"

Save pipeline

joblib.dump(grid.best*estimator*, model_path)

Compute a hash of the training data (optional but handy)

data *hash* = *hashlib.sha256(pd.util.hash* pandas *object(X* train, index=True).values.tobytes()).hexdigest()

metadata = { "model *version":* *"1.0.0",* *"trained* at":

datetime.datetime.utcnow().isoformat(), "python***version": platform.python***version(), "libraries": { "scikit-learn": sklearn.**version**, "pandas": pd.**version**, "numpy": np.**version** }, "data***hash": data***hash, "seed": 42 }

with open(metadata_path, "w") as f: json.dump(metadata, f, indent=2)

2.3 Loading in Production

import joblib model = joblib.load("model_pipeline.joblib")

That's it - the loaded object is **ready to predict**.

3. Building a REST API

3.1 Request / Response Contract

A well-defined API schema makes the service **self-documenting** and prevents runtime errors. For tabular data you typically accept a **JSON list of records**:

{ "instances": [{"age": 35, "salary": 72000, "city": "Seattle", "department": "Engineering", "gender": "F"}, {"age": 28, "salary": 54000, "city": "Boston", "department": "Sales", "gender": "M"}] }

The response can include **probabilities** (for classification) and optionally the **raw model output**:

{ "predictions": [{"label": 1, "probability": 0.87}, {"label": 0, "probability": 0.23}] }

3.2 Flask Implementation

App_flask.py

from flask import Flask, request, jsonify import joblib import pandas as pd

app = Flask(**name**)

Load model once at startup

model = joblib.load("model_pipeline.joblib")

@app.route("/ health", methods=["GET"]) def health(): return jsonify({"status": "ok"}), 200

@app.route("/ predict", methods=["POST"]) def predict(): payload = request.get_json(force=True) # raises 400 if not JSON instances = payload.get("instances") if not instances: return jsonify({"error": "Missing 'instances' key"}), 400

Convert to DataFrame - order of columns does not matter

df = pd.DataFrame(instances)

Model expects the same column names used during training

try: preds = model.predict(df) probs = model.predict_proba(df) [:, 1] # probability of class 1 except Exception as e: return jsonify({"error": str(e)}), 500

results = [{"label": int(p), "probability": float(prob)} for p, prob in zip(preds, probs)] return jsonify({"predictions": results}), 200

if **name** == "**main**":

For development only - use a production WSGI server in prod

app.run(host="0.0.0.0", port=8080, debug=False)

Key points in the Flask version

* `force=True` to reject non-JSON bodies early.
* Minimal validation - you could add `marshmallow` schemas for stricter checks.
* Uses `predict_proba` to return a confidence score.

3.3 FastAPI Implementation

FastAPI shines when you want **automatic validation** and **interactive docs**.

App_fastapi.py

from fastapi import FastAPI, HTTPException from pydantic import BaseModel, Field, validator from typing import List, Dict, Any import joblib import pandas as pd import uvicorn

app = FastAPI(title="Predictive Model API", version="1.0.0")

Load model once

model = joblib.load("model_pipeline.joblib")

class Instance(BaseModel): age: float = Field(..., description="Age in years") salary: float = Field(..., description="Annual salary") city: str = Field(..., description="City of residence") department: str = Field(..., description="Department name") gender: str = Field(..., description="Gender (M/

F)")

@validator("*") def non_negative(cls, v, field): if isinstance(v, (int, float)) and v < 0: raise ValueError(f"{field.name} must be non-negative") return v

class PredictRequest(BaseModel): instances: List[Instance]

class Prediction(BaseModel): label: int probability: float

class PredictResponse(BaseModel): predictions: List[Prediction]

@app.get("/health", tags=["Health"]) def health(): return {"status": "ok"}

@app.post("/ predict", response_model=PredictResponse, tags=["Inference"]) def predict(req: PredictRequest):

Convert list of dicts into DataFrame

df = pd.DataFrame([instance.dict() for instance in req.instances])

try: preds = model.predict(df) probs = model.predict ***proba(df) [:, 1] except Exception as exc: raise HTTPException(status***code=500, detail=str(exc))

response = PredictResponse(predictions=[Prediction(label=int(p), probability=float(prob)) for p, prob in zip(preds, probs)]) return response

if **name** == "**main**":

Run with uvicorn - production ready (workers, reload, etc.)

uvicorn.run(app, host="0.0.0.0", port=8000)

```
**FastAPI highlights**

* **Pydantic models** (`Instance`, `PredictRequest`) automatically validate types and constraints.
* The `/docs` endpoint (Swagger UI) is generated for free - great for internal stakeholders.
* `response_model` ensures the output conforms to the declared schema, preventing accidental leakage of internal fields.

#### 3.4 Testing the API (curl)
```

Flask (port 8080)

```
curl -X POST http://localhost:8080/predict \ -H "Content-Type: application/json" \ -d '{"instances": [{"age":30,"salary":65000,"city":"Austin","department":"Product","gender":"F"}, {"age":45,"salary":120000,"city":"New York","department":"Finance","gender":"M"}]}'
```

```

```

FastAPI (port 8000)

```
curl -X POST http://localhost:8000/predict \ -H "Content-Type: application/json" \ -d '{"instances": [{"age":30,"salary":65000,"city":"Austin","department":"Product","gender":"F"}, {"age":45,"salary":120000,"city":"New York","department":"Finance","gender":"M"}]}'
```

```
Both commands should return a JSON payload with `label` and `probability` for each row.

### 4. Containerisation with Docker

#### 4.1 Why Docker?

* **Isolation** - the container has its own Python interpreter, libraries, and OS packages.
* **Portability** - the same image runs unchanged on any host with Docker installed.
* **Scalability** - containers can be orchestrated (Kubernetes, Docker Swarm) for auto-scaling.

#### 4.2 Minimal Dockerfile (FastAPI + Uvicorn)
```

Dockerfile

FROM python:3.11-slim

System dependencies (optional - for speed)

RUN apt-get update && apt-get install -y --no-install-recommends \ build-essential gcc && \ rm -rf /var/lib/apt/lists/*

Set working directory

WORKDIR /app

Copy only requirements first (leverages Docker caching)

COPY requirements.txt . RUN pip install --no-cache-dir -r requirements.txt

Copy the rest of the source code

```
COPY . .
```

Expose the port the API will run on

```
EXPOSE 8000
```

Command to run the API (Uvicorn)

```
CMD ["uvicorn", "app_fastapi:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
**Explanation of each layer**

| Layer | Purpose |
|-------|---------|
| `FROM python:3.11-slim` | Small base image with Python 3.11. |
| `apt-get install` | Installs low-level build tools needed for some packages (e.g., `numpy`, `scipy`). |
| `WORKDIR /app` | Sets the working directory inside the container. |
| `COPY requirements.txt` + `RUN pip install` | Leverages Docker's cache - if only code changes, the dependencies layer is reused (fast rebuild). |
| `COPY . .` | Copies the API code and the serialized model (`model_pipeline.joblib`). |
| `EXPOSE 8000` | Documents the port the service listens on (helpful for orchestration tools). |
| `CMD [...]` | Starts the service when the container runs. |

#### 4.3 Building and Running the Image
```

Build the image (tag it as ml-

api)

docker build -t ml-api:1.0 .

Run a container locally, mapping host port 9000 -> container port 8000

docker run -d -p 9000:8000 --name ml***api***container ml-api:1.0

Test (same `curl` as before, but now against port 9000):

curl -X POST http://localhost:9000/predict \ -H "Content-Type: application/json" \ - d '{"instances": [{"age":30,"salary":65000,"city":"Austin","department":"Product","gender":"F"}]}'

If you see a JSON response, the containerised service works!

4.4 Adding a `docker-compose.yml` for Development

Sometimes you want to spin up **multiple services** (e.g., the API + a monitoring side-car). `docker-compose` simplifies this:

Docker-compose.yml

version: "3.9"

services: api: build: . image: ml-api:dev container***name: ml***api ports:

- "9000:8000"

environment:

- LOG_LEVEL=INFO

volumes:

Mount the source code for hot-reloading (useful during dev)

- .:/app

restart: unless-stopped

> Run with:

docker compose up --build

```
Now you can edit `app_fastapi.py` locally and see changes reflected instantly (thanks to the bind mount).

#### 4.5 Production-Ready Tips

| Concern | Recommendation |
|---------|----------------|
| **Process manager** | Use `gunicorn` with `uvicorn.workers.UvicornWorker` (e.g., `gunicorn -k uvicorn.workers.UvicornWorker -w 4 app_fastapi:app`). |
| **Health checks** | Kubernetes liveness/readiness probes can call `/health`. |
| **Logging** | Route logs to `stdout`/`stderr` (Docker captures them). Use `structlog` or `loguru` for JSON-structured logs. |
| **Security** | Run the container as a non-root user (`USER 1000`). Enable HTTPS termination at a reverse proxy (NGINX, Traefik). |
| **Resource limits** | In `docker-compose.yml` or K8s spec, set `mem_limit` and `cpu_quota`. |
| **Model updates** | Store the model file in a mounted volume or a cloud bucket; reload the model without restarting the container (watch for file changes). |


## Practical Application

Below is a **complete, end-to-end walkthrough** that ties together everything discussed. The example uses the **UCI Credit Card Default** dataset (a classic imbalanced binary classification problem) and demonstrates:

* **Feature engineering** (unsupervised PCA).
* **Pipeline construction** (numeric + categorical preprocessing).
* **Model training + hyper-parameter tuning**.
* **Serialization** with `joblib`.
* **FastAPI serving** with request validation.
* **Docker containerisation**.

> **Note:** The code snippets are intentionally modular - you can copy-paste each block into a fresh Jupyter notebook or a Python script and run them sequentially.


### 1. Setup - Install Dependencies
```

Create a fresh virtual environment (optional but recommended)

python -m venv venv source venv/bin/activate # Windows: venv\Scripts\activate

Install core libraries

pip install --upgrade pip pip install \ pandas \ numpy \ scikit-learn \ fastapi \ "uvicorn[standard]" \ joblib \ python-multipart # for file uploads if needed

Create a `requirements.txt` for Docker later:

pandas==2.2.1 numpy==1.26.4 scikit- learn==1.5.0 fastapi==0.111.0 uvicorn[standard]==0.30.0 joblib==1.4.2

---

2. Load & Inspect the Data

import pandas as pd from sklearn.model***selection import train***test_split

Credit Card Default dataset (downloaded from UCI)

url = "https:// archive.ics.uci.edu/ ml/ machine- learning- databases/00350/ default%20of%20credit%20card%20clients.xls" df***raw = pd.read*** excel(url, header=1) # first sheet, skip first row

Rename target for clarity

df_raw.rename(columns={"default payment next month": "default"}, inplace=True)

Quick sanity check

print(df*raw.head()) print(df*raw["default"].value_counts())

> **Typical output**

ID LIMIT*BAL* *SEX* *EDUCATION* *MARRIAGE* *AGE* ... *PAY*5 BILL*AMT5* *PAY*AMT5 default
0 1 20000 2 2 1 24 ... 0 0.0 0.0
0 1 2 120000 2 2 1 26 ... 0 0.0 0.0
0 ...

> The dataset contains a mixture of **numeric** (limits, ages, bills) and **categorical** (sex, education, marriage) columns, plus a **highly imbalanced** target (`default` ≈ 22% positive).
>
> ### 3. Define Pre-processing & Feature Extraction

from sklearn.pipeline import Pipeline from sklearn.compose import ColumnTransformer from sklearn.preprocessing import StandardScaler, OneHotEncoder from sklearn.impute import SimpleImputer from sklearn.decomposition import PCA

Identify column groups

numeric*cols* = *["LIMIT*BAL", "AGE", "BILL*AMT1", "BILL*AMT2", "BILL*AMT3", "BILL*AMT4", "BILL*AMT5", "BILL*AMT6", "PAY*AMT1", "PAY*AMT2", "PAY*AMT3", "PAY*AMT4", "PAY*AMT5", "PAY*AMT6"] categorical_cols = ["SEX", "EDUCATION", "MARRIAGE"]

Numeric pipeline

numeric_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median")), ("scaler", StandardScaler())])

Categorical pipeline

categorical *pipe* = *Pipeline(steps=[("imputer", SimpleImputer(strategy="most* frequent")), ("onehot", OneHotEncoder(handle_unknown="ignore"))])

Column transformer

preprocess = ColumnTransformer(transformers=[("num", numeric*pipe, numeric*cols), ("cat", categorical*pipe, categorical*cols)])

Unsupervised feature extraction (PCA on the numeric+encoded space)

unsup*pipe = Pipeline(steps=[("pca", PCA(n*components=10, random_state=42))])

Full pipeline with Gradient Boosting

from sklearn.ensemble import GradientBoostingClassifier

full *pipe = Pipeline(steps=[("preprocess", preprocess), ("unsup", unsup* pipe), ("model", GradientBoostingClassifier(random_state=42))])

Why PCA?
Even after one-hot encoding, the feature space can be high-dimensional (especially with many categories). PCA reduces noise and can improve model stability. In practice you would compare with/without PCA - the pipeline makes this a one-line experiment.

4. Train / Validate

Separate features / target

X = df*raw.drop(columns=["ID", "default"]) y = df*raw["default"]

Train-test split (stratified for imbalanced data)

X*train, X*test, y*train, y*test = train*test*split(X, y, test*size=0.2, random*state=42, stratify=y)

Hyper-parameter grid (small for demonstration)

from sklearn.model_selection import GridSearchCV

param*grid = { "model*learning*rate": [0.05, 0.1], "model**n_estimators": [100, 200], "unsup**pca_*n*components": [5, 10] }

grid = GridSearchCV(estimator=full*pipe, param*grid=param*grid, cv=5, *scoring="roc*auc", n_jobs=-1, verbose=2)

grid.fit(X*train, y*train)

print("Best ROC-AUC:", grid.best*score) print("Best params:", grid.best*params)

After fitting, `grid.best_estimator_` is a **fully trained pipeline** that already includes the optimal PCA dimensionality, learning rate, and number of trees.

5. Evaluate on Hold-out Test Set

from sklearn.metrics import classification***report, roc***auc_score

best***pipe = grid.best***estimator_

y***pred = best***pipe.predict(X***test) y***proba = best***pipe.predict***proba(X_test)[:, 1]

print(classification ***report(y*** test, y ***pred)) print("Test ROC- AUC:", roc***auc***score(y***test, y_proba))

You should see a balanced precision/recall for the minority class (thanks to the tuned model). Keep the test metrics for your documentation - they will be the baseline you compare against after deployment.

6. Serialize the Pipeline

import joblib, json, hashlib, datetime, platform, sklearn

model***path = "model***pipeline.joblib" metadata***path = "model***metadata.json"

Save model

joblib.dump(best***pipe, model***path)

Optional: compute a hash of

the training data for traceability

data ***hash*** = ***hashlib.sha256(pd.util.hash*** pandas ***object(X*** train, index=True).values.tobytes()).hexdigest()

metadata = { "model ***version":*** ***"1.0.0",*** ***"trained*** at": datetime.datetime.utcnow().isoformat() + "Z", "python ***version":*** ***platform.python*** version(), "libraries": { "scikit- learn": sklearn. **version** , "pandas": pd.**version**, "numpy": np.**version** }, "data***hash":*** ***data***hash, "seed": 42 }

with open(metadata_path, "w") as f: json.dump(metadata, f, indent=2)

print("Model and metadata saved.")

You now have two files ready for production:

* `model_pipeline.joblib` - the **entire pipeline** (pre-processing + model).
* `model_metadata.json` - human-readable provenance.

7. Build the FastAPI Service

Create a file `app_fastapi.py` in the same directory:

App_fastapi.py

from fastapi import FastAPI, HTTPException from pydantic import BaseModel, Field, validator from typing import List import pandas as pd import joblib import uvicorn

app = FastAPI(title="Credit- Card- Default Prediction API", version="1.0.0", description="Serves a scikit-learn pipeline that predicts default risk.")

Load the pipeline at startup (global singleton)

model = joblib.load("model_pipeline.joblib")

--- Request schema ----------

class Instance(BaseModel): LIMIT ***BAL: float = Field(..., description="Credit limit") SEX: int = Field(..., ge=1, le=2, description="1=male, 2=female") EDUCATION: int = Field(..., description="Education level") MARRIAGE: int = Field(..., description="Marital status") AGE: int = Field(..., ge=0, description="Age in years") BILL***AMT1: float = Field(..., description="Bill amount month 1") BILL ***AMT2: float = Field(..., description="Bill amount month 2") BILL***AMT3: float = Field(..., description="Bill amount month 3") BILL***AMT4: float = Field(..., description="Bill amount month 4") BILL*** AMT5: float = Field(..., description="Bill amount month 5") BILL***AMT6: float = Field(..., description="Bill amount month 6") PAY***AMT1: float = Field(..., description="Payment amount month 1") PAY***AMT2: float = Field(..., description="Payment amount month 2") PAY***AMT3: float = Field(..., description="Payment amount month 3") PAY ***AMT4: float = Field(..., description="Payment amount month 4") PAY*** AMT5: float = Field(..., description="Payment amount month 5") PAY_AMT6: float = Field(..., description="Payment amount month 6")

class PredictRequest(BaseModel): instances: List[Instance]

class Prediction(BaseModel): label: int probability: float

class PredictResponse(BaseModel): predictions: List[Prediction]

--- Endpoints ----------

@app.get("/ health", tags=["Health"]) def health_check(): """Simple health check used by orchestrators.""" return {"status": "ok"}

@app.post("/ predict", response_model=PredictResponse, tags=["Inference"]) def predict(request: PredictRequest):

Convert list of Instance objects to a DataFrame

df = pd.DataFrame([inst.dict() for inst in request.instances])

try: labels = model.predict(df) probs = model.predict ***proba(df) [:, 1] # probability of default=1 except Exception as exc: raise HTTPException(status***code=500, detail=str(exc))

response = PredictResponse(predictions=[Prediction(label=int(lbl), probability=float(p)) for lbl, p in zip(labels, probs)]) return response

--- Run server ----------

if **name** == "**main**": uvicorn.run(app, host="0.0.0.0", port=8000)

Key points

* **Pydantic validation** catches missing fields, wrong types, or out-of-range values before the model is even hit.
* The endpoint returns **probabilities**, which are essential for downstream risk-scoring.
* The `/health` endpoint can be used by Docker/K8s liveness probes.

8. Test the API Locally

Run the service (development mode)

python app_fastapi.py

Open a browser at <http://localhost:8000/docs> - you'll see an interactive Swagger UI generated automatically. Click **/predict**, press **Try it out**, and paste the following JSON payload:

{ "instances": [{ "LIMIT***BAL": 50000, "SEX": 2, "EDUCATION": 2, "MARRIAGE": 1, "AGE": 30, "BILL***AMT1": 0, "BILL***AMT2": 0, "BILL***AMT3": 0, "BILL***AMT4": 0, "BILL***AMT5": 0, "BILL***AMT6": 0, "PAY***AMT1": 0, "PAY***AMT2": 0, "PAY***AMT3": 0, "PAY***AMT4": 0, "PAY***AMT5": 0, "PAY_AMT6": 0 }] }

You should receive a response similar to:

{ "predictions": [{ "label": 0, "probability": 0.12 }] }

```
If the probability is high (>0.7) you might flag the client for further review.


### 9. Containerise the Service

#### 9.1 Create a Dockerfile
```

```
# Dockerfile (placed in the same folder as app_fastapi.py)

FROM python:3.11-slim

# Install system build tools (required for some wheels)

RUN apt-get update && apt-get install -y --no-install-recommends \ build-essential && \ rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Copy only requirements first for caching

COPY requirements.txt . RUN pip install --no-cache-dir -r requirements.txt

# Copy the code and the serialized model
```

```
COPY . .
```

Expose the port FastAPI will listen on

```
EXPOSE 8000
```

Use gunicorn + uvicorn workers for production

```
CMD ["gunicorn", "-k", "uvicorn.workers.UvicornWorker", "-w", "4", "app_fastapi:app", "--bind", "0.0.0.0:8000"]
```

```
**Why `gunicorn`?**
`gunicorn` spawns multiple worker processes (here 4) that share the same model in memory, giving you better throughput on multi-core machines.

#### 9.2 Build & Run the Image
```

Build

```
docker build -t credit-default-api:1.0 .
```

Run (map host 9000 → container 8000)

```
docker run -d -p 9000:8000 --name credit_api credit-default-api:1.0
```

```
#### 9.3 Verify the Container
```

curl -X POST http://localhost:9000/predict \ -H "Content-Type: application/json" \ -d '{"instances": [{"LIMIT***BAL":200000,"SEX":1,"EDUCATION":1,"MARRIAGE":2,"AGE":45,"BILL***AMT1":5000,"BILL***AMT2":2000,"BILL***AMT3":0,"BILL_AMT4":0,"B

Ethical Considerations & Model Monitoring

> ***"A model that only works in the lab is useless in the real world - and a model that harms people is a failure of engineering." - Anonymous***

In the final module of this book we bring together everything you have learned so far - from data wrangling and feature engineering to handling imbalance and time-series quirks, all the way to production-grade pipelines and APIs. The last piece of the puzzle is **responsibility** : making sure the model you ship is **fair** , **transparent**, and **robust** long after the initial launch.

This chapter walks you through the ethical landscape of predictive modelling, shows you how to **measure** and **mitigate** bias, and then teaches you how to **monitor** models in production so that drift, degradation, or unexpected side-effects are caught early. You will leave with a concrete toolkit you can plug into any Python-based ML workflow.

Table of Contents

| Section | What you'll learn | |---------|-------------------| | **Introduction** | Why ethics and monitoring matter, and how they fit into the end-to-end ML lifecycle. | | **Core Concepts** | Sources of bias, fairness definitions, drift types,

monitoring architecture, documentation standards. | | **Practical Application** | Hands-on example: loan-approval model → bias audit → mitigation → drift-aware dashboard → governance artefacts. | | **Key Takeaways** | Concise recap of the most actionable points. |

Introduction

When you built the previous modules you were primarily concerned with **accuracy**: can the model predict the right label? In the real world, however, **accuracy alone is never enough**. A model that predicts loan defaults with 95% accuracy but systematically denies credit to a protected group is a **legal and reputational liability**. Likewise, a model that performs well today but silently degrades because the underlying data distribution has shifted can cause costly downstream errors.

Why Ethical Considerations Matter

Ethical Dimension	Business Impact	Real-World Example
Fairness	Avoid discrimination lawsuits, maintain brand trust	A hiring algorithm that penalises resumes containing "women's college" keywords.
Transparency	Enables auditors and regulators to understand decisions	Credit scoring that cannot be explained to a borrower.
Accountability	Clear ownership of model outcomes	A fraud-detection model that flags an entire demographic without justification.
Privacy	Compliance with GDPR, CCPA, etc.	Using raw IP addresses as features.

Why Ongoing Monitoring Is Essential

- **Concept drift** - the relationship between features and target changes (e.g., new fraud patterns).
- **Data drift** - the input feature distribution changes (e.g., a new product line adds a new feature value).

- **Performance decay** - model metrics (precision, recall, AUC) deteriorate over time.

If you **only test once** before deployment, you risk missing all three. Continuous monitoring gives you a **feedback loop** that lets you intervene- re- train, adjust thresholds, or roll back-before the model harms users or the business.

Core Concepts

Below we unpack the technical foundations you need to embed ethics and monitoring into any ML system.

1. Sources of Bias

| Source | Description | Typical Symptoms |
|--------|-------------|------------------| | **Sampling bias** | Training data not representative of the target population. | Over- representation of a demographic; poor performance on under- represented groups. | | **Label bias** | Human annotators embed their own prejudices. | Systematic under- reporting of certain outcomes (e.g., crime incidents in minority neighborhoods). | | **Feature bias** | Features encode protected attributes directly or indirectly (proxy variables). | "ZIP code" correlating strongly with race, leading to red-lining. | | **Algorithmic bias** | The learning algorithm optimises a global loss that favours the majority class. | Imbalanced classification where minority group errors dominate overall loss. | | **Evaluation bias** | Metrics chosen ignore subgroup performance. | High overall accuracy but low recall for a protected group. |

> ***Tip: Whenever you add a new feature, ask "Could this be a proxy for a protected attribute?" and document the answer.***

2. Fairness Definitions

Fairness is **multifaceted**; no single metric satisfies all contexts. The most

common quantitative definitions are:

Fairness Notion	Formal Definition	When It Makes Sense
Demographic Parity (Statistical Parity)	$P(\hat{Y}=1 \mid A=a) = P(\hat{Y}=1 \mid A=b)$ for all protected groups a,b.	When you want equal ***selection rates*** (e.g., loan offers).
Equalized Odds	$P(\hat{Y}=1 \mid Y=y, A=a) = P(\hat{Y}=1 \mid Y=y, A=b)$ for each true label y.	When you care about equal ***error rates*** (e.g., false-positive rates in policing).
Equal Opportunity	A special case of Equalized Odds focusing only on $Y=1$ (true positives).	When the cost of false negatives is high (e.g., disease screening).
Predictive Parity	$P(Y=1 \mid \hat{Y}=1, A=a) = P(Y=1 \mid \hat{Y}=1, A=b)$.	When you need equal ***positive predictive value*** across groups (e.g., credit scoring).
Individual Fairness	"Similar individuals should receive similar predictions."	When you can define a similarity metric (e.g., distance in feature space).

Important: These notions can be mutually exclusive. Achieving demographic parity may worsen equalized odds, especially when base rates differ across groups. The key is **choosing the right trade-off for your domain** and documenting the decision.

3. Measuring Fairness

A practical fairness audit typically follows these steps:

1. **Identify protected attributes** - gender, race, age, disability status, etc.
2. **Split the test set by group** - create a DataFrame for each protected value.
3. **Compute confusion matrices per group** - obtain TP, FP, FN, TN.
4. **Derive fairness metrics** - e.g., demographic parity difference, equalized odds disparity.

Below is a compact Python utility that computes the most common fairness metrics:

```
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix

def fairness_report(y_true, y_pred, protected, privileged_val):
    """
    y_true, y_pred : array-like binary labels
    protected      : array-like protected attribute (e.g., gender)
    privileged_val : value considered privileged (e.g., "Male")
    Returns a dict of fairness metrics.
    """
    df = pd.DataFrame({
        "y_true": y_true,
        "y_pred": y_pred,
        "protected": protected
    })
    groups = df["protected"].unique()
    metrics = {}
    for g in groups:
        sub = df[df["protected"] == g]
        tn, fp, fn, tp = confusion_matrix(sub["y_true"], sub["y_pred"]).ravel()
        pos_rate = sub["y_pred"].mean()
        tpr = tp / (tp + fn) if (tp + fn) > 0 else np.nan  # recall
        fpr = fp / (fp + tn) if (fp + tn) > 0 else np.nan
        ppv = tp / (tp + fp) if (tp + fp) > 0 else np.nan  # precision
        metrics[g] = {
            "selection_rate": pos_rate,
            "TPR": tpr,
            "FPR": fpr,
            "PPV": ppv,
            "support": len(sub)
        }
    # Demographic parity difference
    dp_diff = abs(metrics[privileged_val]["selection_rate"] -
                  metrics[[g for g in groups if g != privileged_val][0]]["selection_rate"])
    # Equalized odds disparity (max difference in TPR and FPR)
    eo_disp = max(
        abs(metrics[privileged_val]["TPR"] - metrics[[g for g in groups if g != privileged_val][0]]
["TPR"]),
        abs(metrics[privileged_val]["FPR"] - metrics[[g for g in groups if g != privileged_val][0]]
["FPR"])
    )
```

... (continued on next page)

```
    return {"group_metrics": metrics, "demographic_parity_diff": dp_diff,
            "equalized_odds_disp": eo_disp}
```

> ***Tip:** Store the output of this function alongside your model version (e.g., in a JSON file) to track fairness over time.*

4. Model Drift

Drift Type	Definition	Detection Technique
Data (Covariate) Drift	Input feature distribution changes while $P(Y\|X)$ stays the same.	Population Stability Index (PSI), Kolmogorov-Smirnov test, Wasserstein distance.
Concept Drift	The conditional distribution $P(Y\|X)$ itself changes.	Monitoring performance metrics (AUC, recall) on recent data; online learning error rates.
Label Drift	The prevalence of the target class changes (e.g., fraud rate spikes).	Track class balance over time; compute drift on the target variable if you have recent labels.

Why both matter: A model can retain high accuracy even when data drift occurs if the drift does not affect the decision boundary. Conversely, concept drift will almost always cause performance decay.

5. Monitoring Architecture

A robust monitoring stack typically consists of **four layers**:

1. **Data Ingestion Layer** - Captures raw input features and, when available, true labels.
2. **Metric Computation Layer** - Calculates performance (AUC, F1), fairness (DP, EO), and drift (PSI) on a sliding window (e.g., last 24 h).
3. **Alerting Layer** - Triggers alerts when any metric crosses a predefined threshold (e.g., DP diff > 0.1).

4. **Visualization Layer** - Dashboards for stakeholders (data scientists, product managers, compliance).

A minimal implementation can be built with **Python + FastAPI + Prometheus + Grafana** , but many organizations use managed services (AWS CloudWatch, Azure Monitor, GCP Vertex AI). The next section shows a **self- contained Streamlit dashboard** you can run locally or host on a minimal VM.

6. Documentation, Version Control, and Governance

Artefact	Purpose	Recommended Format
Model Card	Summarises model intent, performance, fairness, intended use, and limitations.	Markdown or JSON (see ***Model Card*** template below).
Data Sheet	Documents dataset provenance, collection process, and known biases.	Markdown, PDF, or Jupyter Notebook.
Training Pipeline Code	Reproducible steps from raw data to model artifact.	Git repository + DVC (Data Version Control) for large data/weights.
Change Log	Records every model version, reason for change, and impact assessment.	Structured YAML (e.g., `changelog.yaml`).
Governance Checklist	Ensures each release passes legal, ethical, and technical sign-offs.	Google Form or simple markdown checklist.

> ***Best Practice: Treat every model release as a software release - tag the Git commit, archive the Docker image, and store the model card alongside the binary artifact (e.g., in an S3 bucket with versioned prefixes).***

Practical Application

In this hands-on section we will:

1. **Load a public loan-approval dataset** (the "German Credit" data).
2. **Train a baseline classifier** and evaluate its fairness.
3. **Mitigate bias** using re-weighting and threshold adjustment.

4. **Deploy a simple monitoring pipeline** that tracks performance, fairness, and drift in real time.

5. **Generate governance artefacts** (model card, data sheet, changelog).

All code snippets are runnable in a fresh Conda environment. Feel free to copy-paste into a Jupyter notebook or VS Code.

1. Setup

```
conda create -n ethical-ml python=3.11 -y
conda activate ethical-ml

# Core ML stack
pip install pandas numpy scikit-learn matplotlib seaborn tqdm

# Fairness library (optional but handy)
pip install aif360==0.5.0  # will also install cvxpy, etc.

# Monitoring & dashboard
pip install streamlit plotly

# Version control for data & models
pip install dvc[gs]  # DVC with Google Cloud support (replace with s3/azure as needed)
```

> *Note: If you cannot install `aif360` due to platform constraints, the fairness utility we defined earlier works without it.*

2. Load & Explore the Data

```
import pandas as pd
from sklearn.model_selection import train_test_split

# German Credit dataset (UCI) - already cleaned in many tutorials
url = "https://raw.githubusercontent.com/ageron/handson-ml2/master/datasets/german_credit_data.csv"
df = pd.read_csv(url)

# Inspect
print(df.head())
print(df['Sex'].value_counts())
```

The dataset contains a binary target `Risk` (`1` = good, `2` = bad) and a protected attribute `Sex` (`Male`, `Female`). For illustration we treat `Sex` as the **protected group** and `Male` as the privileged class.

```
# Quick preprocessing
df['Risk'] = df['Risk'].map({1:0, 2:1})   # 0 = low risk, 1 = high risk (bad loan)
X = df.drop(columns=['Risk'])
y = df['Risk']
```

3. Train a Baseline Model

```
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import roc_auc_score, accuracy_score, classification_report

# Identify categorical vs numeric columns
cat_cols = X.select_dtypes(include=['object']).columns.tolist()
num_cols = [c for c in X.columns if c not in cat_cols]

preprocess = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), num_cols),
        ('cat', OneHotEncoder(handle_unknown='ignore'), cat_cols)
    ])

model = GradientBoostingClassifier(random_state=42)

pipeline = Pipeline(steps=[('preprocess', preprocess),
                           ('clf', model)])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42)

pipeline.fit(X_train, y_train)

# Predictions
y_pred = pipeline.predict(X_test)
y_proba = pipeline.predict_proba(X_test)[:,1]

# Basic performance
print("AUC:", roc_auc_score(y_test, y_proba))
print(classification_report(y_test, y_pred))
```

Result (example):

- AUC ≈ 0.78
- Accuracy ≈ 0.71

Now we must ask: **How does it treat men vs women?**

4. Fairness Audit

```
# Extract protected attribute from the test set (remember it was one-hot encoded)
protected_test = X_test['Sex'].values

fair_report = fairness_report(y_test, y_pred, protected_test, privileged_val='Male')
fair_report
```

Typical output (illustrative):


```
{
  "group_metrics": {
    "Male": {
      "selection_rate": 0.44,
      "TPR": 0.68,
      "FPR": 0.32,
      "PPV": 0.57,
      "support": 250
    },
    "Female": {
      "selection_rate": 0.28,
      "TPR": 0.53,
      "FPR": 0.22,
      "PPV": 0.61,
      "support": 100
    }
  },
  "demographic_parity_diff": 0.16,
  "equalized_odds_disp": 0.15
}
```


Interpretation

- **Demographic parity difference** (0.16) exceeds a common business tolerance of 0.10 - women are offered "good credit" far less often.
- **Equalized odds disparity** (0.15) shows women have lower true-positive and false-positive rates, indicating the model is **less sensitive** to their risk profile.

> ***Key Insight: Even though overall metrics look fine, the model is unfair toward female applicants.***

5. Bias Mitigation

We will try two complementary techniques:

1. **Re-weighting the training data** so that the loss gives more importance to the under-represented group.
2. **Threshold adjustment** to equalize selection rates post-hoc (a simple, deployment-friendly method).

5.1 Re-weighting

```
import numpy as np

# Compute group-wise weights (inverse of prevalence)
train_protected = X_train['Sex'].values
group_counts = pd.Series(train_protected).value_counts()
weights = np.where(train_protected == 'Male',
                   1.0 / group_counts['Male'],
                   1.0 / group_counts['Female'])

# Fit a new model with sample_weight
pipeline_rw = Pipeline(steps=[('preprocess', preprocess),
                              ('clf', GradientBoostingClassifier(random_state=42))])

pipeline_rw.fit(X_train, y_train, clf__sample_weight=weights)

# Evaluate
y_pred_rw = pipeline_rw.predict(X_test)
y_proba_rw = pipeline_rw.predict_proba(X_test)[:,1]
print("AUC (re-weighted):", roc_auc_score(y_test, y_proba_rw))
fair_rw = fairness_report(y_test, y_pred_rw, protected_test, privileged_val='Male')
fair_rw
```

Typical changes:

- AUC may dip slightly (e.g., from 0.78 → 0.75) because we sacrifice a bit of global accuracy for fairness.
- **Demographic parity diff** often drops dramatically (e.g., 0.04) - now the selection rates are much closer.

5.2 Threshold Adjustment

Even after re-weighting, you might still want to **equalize selection rates** at inference time. The idea is to compute a **group-specific decision threshold** that yields the same selection rate.

```
def find_threshold(probas, target_rate):
    """Return the probability cutoff that yields the given selection rate."""
    return np.percentile(probas, 100 * (1 - target_rate))

# Desired selection rate = average across groups
target_rate = (fair_rw["group_metrics"]["Male"]["selection_rate"] +
               fair_rw["group_metrics"]["Female"]["selection_rate"]) / 2

# Compute thresholds per group
male_thr = find_threshold(y_proba_rw[protected_test == 'Male'], target_rate)
female_thr = find_threshold(y_proba_rw[protected_test == 'Female'], target_rate)

# Apply thresholds
y_pred_adj = np.where(
    (protected_test == 'Male') & (y_proba_rw >= male_thr), 1,
    np.where((protected_test == 'Female') & (y_proba_rw >= female_thr), 1, 0)
)

# New fairness report
fair_adj = fairness_report(y_test, y_pred_adj, protected_test, privileged_val='Male')
fair_adj
```

Result: **Demographic parity diff** ≈ 0 (by construction) while **AUC** may be a bit lower because we are not using a universal threshold. This technique is **transparent** - you can expose the two thresholds in your API docs.

6. Deploying a Monitoring Pipeline

Next we set up a **lightweight monitoring service** that ingests new predictions, recomputes metrics, and surfaces them in a Streamlit dashboard.

6.1 Simulated Production Stream

We'll create a generator that mimics an online request flow.

```
import time, random
import streamlit as st
import plotly.express as px

# Load the final model (re-weighted + thresholds)
final_model = pipeline_rw  # already trained
final_thresholds = {"Male": male_thr, "Female": female_thr}

def predict_online(row):
    """Simulate a real-time prediction."""
    # row is a dict of raw features (including 'Sex')
    X_row = pd.DataFrame([row])
    proba = final_model.predict_proba(X_row)[0,1]
    thr = final_thresholds[row['Sex']]
    pred = int(proba >= thr)
    return pred, proba
```

6.2 Metric Buffer

We'll keep a **sliding window** of the last `N=500` predictions.

```
from collections import deque

WINDOW = 500
buffer = deque(maxlen=WINDOW)

def update_buffer(row, true_label, pred, proba):
    buffer.append({
        "Sex": row['Sex'],
        "y_true": true_label,
        "y_pred": pred,
        "y_proba": proba
    })
```

6.3 Drift Computation (PSI)

```
def psi(reference, current, buckets=10):
    """Population Stability Index for a single numeric feature."""
    ref_bins = np.histogram(reference, bins=buckets)[0] + 0.0001
    cur_bins = np.histogram(current, bins=buckets)[0] + 0.0001
    psi_val = np.sum((ref_bins - cur_bins) * np.log(ref_bins / cur_bins))
    return psi_val
```

We'll compute PSI for the **probability score** itself (a proxy for covariate drift).

6.4 Streamlit Dashboard

Create a file `monitor_dashboard.py`:

```
# monitor_dashboard.py
import streamlit as st
import pandas as pd
import numpy as np
import plotly.express as px
from collections import deque

# Load objects saved from training (you could pickle them)
import joblib, json, os
model = joblib.load("model_rw.pkl")
thresholds = json.load(open("thresholds.json"))

# Shared state
if "buffer" not in st.session_state:
    st.session_state.buffer = deque(maxlen=500)

st.title("🚦 Model Ethics & Drift Dashboard")
st.sidebar.header("Controls")
n_samples = st.sidebar.slider("Show last N predictions", 100, 500, 300)

# ------------------------------------------------------------------
# 1️⃣ Simulate incoming data (in real life you would read from a queue)
# ------------------------------------------------------------------
def simulate_batch():
    # Randomly draw from the original test set to keep distribution realistic
    idx = np.random.choice(X_test.index, size=50, replace=False)
    batch = X_test.loc[idx].copy()
    y_batch = y_test.loc[idx].values
    for i, row in batch.iterrows():
        pred, proba = predict_online(row.to_dict())
        update_buffer(row.to_dict(), y_batch[i], pred, proba)

# Button to generate new data
if st.button("🔄 Pull next batch"):
    simulate_batch()
    st.success("Added 50 new predictions to the window")

# ------------------------------------------------------------------
# 2️⃣ Compute metrics on the sliding window
# ------------------------------------------------------------------
```

... (continued on next page)

```
if len(st.session_state.buffer) > 0:
    df_win = pd.DataFrame(st.session_state.buffer)

    # Performance
    auc = roc_auc_score(df_win["y_true"], df_win["y_proba"])
    acc = (df_win["y_true"] == df_win["y_pred"]).mean()

    # Fairness (using the same utility)
    fair = fairness_report(df_win["y_true"], df_win["y_pred"],
                           df_win["Sex"], privileged_val='Male')

    # Drift (PSI on scores)
    psi_score = psi(df_win["y_proba"].values, np.random.rand(len(df_win)))  # placeholder

    # ------------------- Display -------------------
    st.subheader("📊 Performance")
    col1, col2 = st.columns(2)
    col1.metric("AUC (window)", f"{auc:.3f}")
    col2.metric("Accuracy (window)", f"{acc:.3%}")

    st.subheader("⚖️ Fairness")
    st.write("Selection rate (Male):", f"{fair['group_metrics']['Male']['selection_rate']:.2%}")
    st.write("Selection rate (Female):", f"{fair['group_metrics']['Female']['selection_rate']:.2%}")
    st.metric("Demographic Parity Δ", f"{fair['demographic_parity_diff']:.3f}")

    st.subheader("📉 Drift")
    st.metric("PSI (score distribution)", f"{psi_score:.3f}")

    # Visualisations
    st.subheader("📈 Detailed Plots")
    fig1 = px.histogram(df_win, x="y_proba", color="Sex", nbins=20,
                        title="Score Distribution by Sex")
    st.plotly_chart(fig1, use_container_width=True)

    fig2 = px.box(df_win, x="Sex", y="y_proba", points="all",
                  title="Score Boxplot by Sex")
    st.plotly_chart(fig2, use_container_width=True)

else:
    st.info("No data yet - click **Pull next batch** to start.")
```

Run the dashboard:

```
streamlit run monitor_dashboard.py
```

What you see

- Real-time AUC, accuracy, and demographic parity numbers that update as new batches flow in.
- A **PSI alert** (you could colour-code it red when > 0.2,a common drift threshold).
- Visual plots that let you spot if scores for a protected group are drifting away from the baseline.

> ***Production tip: Replace the `simulate_batch` function with a consumer of a Kafka or Pub/Sub topic that receives JSON payloads from your API endpoint.***

7. Governance Artefacts

7.1 Model Card (Markdown)

```
# Model Card: German Credit Risk Classifier (Fairness-Adjusted)

**Version**: 1.2.0
**Date**: 2026-02-13
**Owner**: credit-risk-team@example.com

## Intended Use
- **Primary**: Predict probability of loan default for personal credit applications in the EU.
- **Out-of-Scope**: Business loans, mortgages, or any non-consumer credit product.

## Model Details
| Component | Description |
|-----------|-------------|
| Algorithm | Gradient Boosting (n_estimators=200, learning_rate=0.05) |
| Training Data | German Credit dataset (UCI), 1 000samples, 20 features. |
| Protected Attribute | `Sex` (Male, Female) |
| Fairness Technique | Re-weighting of training instances + group-specific thresholds. |
| Performance (test set) | AUC = 0.75, Accuracy = 0.71 |
| Fairness (test set) | Demographic Parity Δ = 0.04, Equalized Odds Δ = 0.07 |

## Evaluation Data
- Split: 80 %train / 20 %hold-out (stratified).
- No leakage: All preprocessing fit only on training split.

## Ethical Considerations
- **Bias**: Original model exhibited a 0.16 demographic parity gap.
- **Mitigation**: Re-weighting reduced the gap to 0.04; threshold adjustment forced exact parity.
- **Residual Risk**: Slight performance loss (AUC ↓ 0.03). Continuous monitoring required.

## Monitoring Plan
- **Metrics Tracked**: AUC, Accuracy, DP Δ, EO Δ, PSI on score distribution.
- **Alert Thresholds**: DP Δ > 0.10, PSI > 0.20, AUC drop > 0.05.
- **Dashboard**: Streamlit app at `https://ml-dashboard.example.com/credit-risk`.

## Versioning & Reproducibility
- Code: `git commit 9f2c7e8` (tag `v1.2.0`).
- Data: Managed with DVC (`dvc pull data/credit.dvc`).
- Model artifact: `s3://ml-models/credit-risk/v1.2.0/model.pkl`.

## License
```

... (continued on next page)

```
- Dataset: UCI public domain.
- Model: Apache 2.0.

*Prepared by the Credit Risk ML Team*
```

7.2 Data Sheet (Markdown)

```
# Data Sheet: German Credit Dataset (UCI)

## Dataset Overview
- **Source**: UCI Machine Learning Repository (1994).
- **Number of Instances**: 1 000.
- **Number of Features**: 20 (mixed categorical & numeric).
- **Target**: `Risk` (0 = good, 1 = bad).

## Collection Process
- Collected from a German bank for a credit-scoring study.
- Applicants were screened by a human officer; demographic attributes were self-reported.

## Sensitive Attributes
- `Sex` (Male/Female) - **Protected** under EU anti-discrimination law.
- `Age` - not provided directly but can be inferred from `Duration of Credit` (potential proxy).

## Known Biases
- **Sampling bias**: Over-representation of male applicants (71 %male).
- **Label bias**: Risk assessments made by a single officer, possibly reflecting personal heuristics.

## Preprocessing Performed
- One-hot encoding of categorical variables.
- Standard scaling of numeric attributes.
- No imputation required (dataset complete).

## Suggested Use Cases
- Educational demos of credit scoring.
- Baseline for fairness research (protected attribute available).

## Limitations
- Small sample size → limited generalisation to modern credit markets.
- Historical societal norms may not reflect current legal frameworks.

*Compiled by Data Governance Team, 2026-02-10*
```

7.3 Change Log (YAML)

```
# changelog.yaml
- version: "1.0.0"
  date: "2025-12-01"
  author: "alice@example.com"
  description: "Initial baseline model (no fairness adjustments)."
  metrics:
    auc: 0.78
    dp_delta: 0.16
    eo_delta: 0.15
  notes: "Model passed performance review but flagged for fairness."

- version: "1.1.0"
  date: "2026-01-15"
  author: "bob@example.com"
  description: "Added re-weighting to address demographic parity."
  metrics:
    auc: 0.75
    dp_delta: 0.07
    eo_delta: 0.09
  notes: "Performance drop acceptable; fairness improved."

- version: "1.2.0"
  date: "2026-02-13"
  author: "carol@example.com"
  description: "Implemented group-specific thresholds; integrated monitoring dashboard."
  metrics:
    auc: 0.74
    dp_delta: 0.00
    eo_delta: 0.08
  notes: "Thresholds enforce exact parity; monitoring alerts set up."
```

Governance Checklist (example)

| ✅ Item | Owner | Status | |--------|-------|--------| | Data provenance documented (Data Sheet) | Data Engineer | ✅ | | Fairness metrics computed on hold-out set | Data Scientist | ✅ | | Bias mitigation technique approved by legal | Compliance | ✅ | | Model card versioned and stored with artifact | ML Engineer | ✅ | | Monitoring dashboard deployed to production environment | DevOps | ✅ | | Alert thresholds defined and tested | SRE | ✅ | | Annual re-audit scheduled | Risk Management | ☐ |

Key Takeaways

- **Bias is multi-source**: sampling, labeling, feature engineering, algorithmic loss functions, and evaluation metrics can all embed unfairness.
- **Fairness is a trade-off**: choose the definition (DP, EO, PPV, etc.) that aligns with your business policy and legal regime.
- **Quantify fairness** with **group-wise confusion matrices** and derived metrics; store these numbers for every model version.
- **Mitigation strategies** (re-weighting, adversarial debiasing, threshold adjustment, post-processing) can be combined; always re-evaluate performance after each step.
- **Drift is inevitable**: monitor data distribution (PSI, KS), model performance (AUC, recall), and fairness metrics continuously.
- **Monitoring architecture**: ingest raw predictions → compute rolling metrics → trigger alerts → visualise on a dashboard. A simple Streamlit-based UI is enough for prototypes; production systems often use Prometheus/Grafana or cloud-native observability stacks.
- **Governance matters**: a **Model Card**, **Data Sheet**, **Change Log**, and **Checklist** create a transparent audit trail, satisfy regulators, and enable rapid rollback if something goes wrong.
- **Version control** is not just code: use **Git** for scripts, **DVC** (or MLflow) for data & model artifacts, and store artefacts in immutable storage (S3, GCS).
- **Human-in-the-loop**: Even with automated monitoring, periodic manual reviews of fairness dashboards and drift reports are essential.

By embedding these practices into every stage- from data collection to post-deployment-you turn a **predictive model** into a **responsible AI system** that delivers value without compromising ethics or compliance. The journey doesn't end at "model deployed"; it continues as a **living service** that you watch, audit, and improve over time.

Congratulations! You have now completed the ten-module learning path. You are equipped not only to build accurate predictive models but also to do so

responsibly, transparently, and sustainably.

Summary

Course Summary - LearnMachine Learning for Building Predictive Models

---

1. Key Learning Outcomes

By the end of this program you will be able to:

1. **Frame real-world problems as machine-learning tasks** - identify whether a business question calls for classification, regression, clustering, or anomaly detection, and articulate the success metrics that matter to stakeholders.
2. **Explore, visualise and understand data** - use statistical summaries and interactive visualisations (histograms, pair-plots, correlation heat-maps, time-series plots) to uncover patterns, spot outliers, and generate hypotheses before any modelling begins.
3. **Prepare clean, model-ready data** - apply best-practice preprocessing pipelines (missing-value imputation, scaling, encoding, dimensionality reduction) and engineer high-impact features (interaction terms, lag variables, domain-specific aggregations).
4. **Build and interpret linear models** - fit ordinary least-squares regression, logistic regression, and regularised variants (Ridge, Lasso, Elastic Net); read coefficient magnitudes, confidence intervals, and odds ratios to explain model behaviour.
5. **Deploy powerful tree-based learners** - train decision-tree ensembles such as Random Forests, Gradient Boosting Machines (XGBoost, LightGBM, CatBoost) and understand their strengths (non-linearity, handling of categorical variables, built-in feature importance).
6. **Evaluate, validate and tune models rigorously** - split data correctly (train/validation/test, stratified or time-aware folds), compute appropriate metrics

(RMSE, MAE, AUC-ROC, Precision-Recall, F1, Brier score), and perform hyper-parameter optimisation using grid search, random search, or Bayesian optimisation.

7. **Exploit unsupervised learning for feature extraction** - apply Principal Component Analysis (PCA), t-SNE/UMAP, and clustering (K-means, DBSCAN, hierarchical) to discover latent structures, reduce dimensionality, and create new predictive attributes.

8. **Handle specialised data challenges** - work with imbalanced classes (SMOTE, class weighting, focal loss) and time-series data (rolling windows, lag features, seasonal decomposition, Prophet/ARIMA) while preserving temporal integrity.

9. **Package and serve models at scale** - construct reproducible pipelines with Scikit-learn, TensorFlow Extended(TFX) or PySpark, containerise with Docker, expose RESTful APIs via FastAPI/Flask, and orchestrate deployments on cloud platforms (AWS SageMaker, GCP AI Platform, Azure ML).

10. **Apply ethical AI principles and monitor performance** - recognise bias sources, conduct fairness audits (demographic parity, equal opportunity), document model cards, and set up continuous monitoring for data drift, concept drift, and degradation of key business metrics.

---

2. Important Concepts Recap

Module	Core Concepts & Tools	Why It Matters
Foundations & Problem Framing	Supervised vs unsupervised, loss functions, bias-variance trade-off, CRISP-DM, business KPI mapping	Guarantees you start with a clear, measurable objective rather than a vague "apply ML".
Data Exploration & Visualization	Pandas profiling, Seaborn/Matplotlib, Plotly, interactive dashboards	Early insights prevent costly downstream re-work and surface hidden data quality issues.
Data Preprocessing & Feature Engineering	Imputation (mean, KNN, iterative), scaling (Standard, MinMax, Robust), encoding (One-Hot, Target, Frequency), pipelines, feature crosses, temporal lags	A clean, well-engineered dataset is the single biggest lever for model performance.
Supervised Learning - Linear Models	Linear regression, logistic regression, regularisation (L1/L2), GLM

family, interpretation (coefficients, odds ratios) | Provides a transparent baseline, easy to debug, and often sufficient for tabular problems with linear relationships. | | **Supervised Learning - Tree-Based Models** | Decision trees, bagging, Random Forests, Gradient Boosting, XGBoost, LightGBM, CatBoost, SHAP values | Handles non-linear interactions automatically, robust to outliers, and yields powerful predictive power. | | **Model Evaluation, Validation & Hyper-parameter Tuning** | Train-test split, K-fold, time-series split, cross-validation, metric selection, confusion matrix, ROC/PR curves, GridSearchCV, Optuna, Hyperopt | Prevents over-fitting, ensures results generalise, and finds the sweet-spot of model complexity. | | **Unsupervised Learning for Feature Extraction** | PCA, ICA, Autoencoders, t-SNE/UMAP, clustering algorithms, silhouette score | Reduces dimensionality, uncovers hidden groupings, and creates compact representations for downstream models. | | **Working with Imbalanced & Time-Series Data** | Resampling (SMOTE, ADASYN), cost-sensitive learning, class weighting, rolling windows, lag/lead features, seasonality decomposition | Guarantees that rare but critical events (fraud, churn) are correctly learned and temporal leakage is avoided. | | **Deploying Predictive Models - Pipelines & APIs** | Scikit-learn Pipelines, MLflow tracking, Docker, FastAPI/Flask, CI/CD (GitHub Actions), monitoring (Prometheus, Grafana) | Turns a notebook prototype into a production-ready service that can be consumed by applications. | | **Ethical Considerations & Model Monitoring** | Bias detection, fairness metrics, model cards, data-drift detection (Kolmogorov-Smirnov, PSI), alerting, retraining strategies | Ensures responsible AI use, protects brand reputation, and keeps models accurate as the world changes. |

Take-away patterns

- **Iterative workflow:** Exploration → preprocessing → baseline → model upgrade → evaluation → deployment → monitoring.
- **Reproducibility is non-negotiable:** Use version-controlled code, data lineage tools (DVC, Delta Lake), and immutable pipelines.
- **Interpretability + performance:** Linear models give transparency; tree-based ensembles give accuracy. Pair them with SHAP/ELI5 to get the best of both worlds.
- **Domain knowledge fuels feature engineering:** The most effective features often arise from business logic, not from generic transformations.

3. Next Steps Guidance

A. Consolidate Your Portfolio

1. **Select three end-to-end projects** that showcase the full pipeline: (i) a classic tabular regression problem (e.g., house-price prediction), (ii) a classification task with severe class imbalance (e.g., fraud detection), and (iii) a time-series forecasting case (e.g., demand planning).

2. **Document each project** in a GitHub repository with a clear README, data-processing scripts, model-training notebooks, Dockerfile, and a short video demo of the API in action.

B. Deepen Technical Expertise

- **Advanced modelling:** Dive into deep learning for tabular data (TabNet, DeepGBM) and probabilistic modelling (Bayesian regression, Gaussian Processes).
- **Scale-out:** Learn Spark MLlib or Dask to handle millions of rows, and experiment with distributed training on GPUs (e.g., XGBoost on Amazon SageMaker).
- **AutoML:** Explore tools like AutoGluon, H2O AutoML, or Google AutoML Tables to understand automated pipeline generation and where human insight still adds value.

C. Strengthen Production Skills

- **CI/CD pipelines:** Build a GitHub Actions workflow that runs unit tests, lints code, builds a Docker image, and pushes it to a container registry.
- **Observability:** Implement logging (structured JSON), tracing (OpenTelemetry), and automated drift alerts using tools such as Evidently AI or WhyLabs.
- **Model governance:** Create model cards and data sheets for each model, and practice versioning with MLflow or DVC.

D. Embed Ethical AI Practices

- **Bias audits:** Run fairness dashboards on each model, experiment with mitigation techniques (re-weighting, adversarial debiasing).
- **Regulatory awareness:** Familiarise yourself with emerging standards (EU AI Act, US Algorithmic Accountability Act) and industry-specific guidelines (HIPAA for healthcare, GDPR for personal data).

E. Community & Continuous Learning

- **Join ML meet-ups, Kaggle competitions, and open-source projects** to stay current with new algorithms and tooling.
- **Subscribe to newsletters and podcasts** (e.g., "Import AI", "Data Skeptic") and follow key researchers on Twitter/X.
- **Consider certifications** (AWS Certified Machine Learning - Specialty, Google Professional ML Engineer) to validate your skills to employers.

---

4. Congratulations!

You have just completed an intensive, hands- on journey through the entire lifecycle of predictive modelling- from turning a vague business question into a data-driven solution, through rigorous experimentation and evaluation, all the way to production deployment and ethical stewardship.

This achievement marks more than just the acquisition of technical know- how; it demonstrates a mindset of curiosity, disciplined experimentation, and responsibility-qualities that distinguish a true machine-learning practitioner.

Carry forward the confidence you have earned, continue to iterate on the projects you built, and let the habit of asking the right question guide every new dataset you encounter. The world of AI is evolving at breakneck speed, and you are now equipped to not just keep pace, but to lead the conversation on building trustworthy, high-impact predictive models.

Well done, and welcome to the community of data-driven problem solvers!

Glossary

Algorithm: A step-by-step computational procedure (e.g., decision tree, XGBoost) that maps inputs to predictions or actions.

A/B testing: A controlled experiment that compares two versions of a model or system (treatment Avs. B)to measure which delivers better business metrics.

Bias-variance trade-off: The balance between systematic error (bias) from an overly simple model and sensitivity to training data (variance) from an overly complex model; optimal performance minimizes total error.

Classification: A supervised-learning task where the target variable is categorical, and the model assigns each instance to one of a finite set of classes (e.g., churn = 1/0).

Clustering: An unsupervised-learning technique that groups similar observations together without using labeled outcomes (e.g., K-means, DBSCAN).

Cross-entropy loss: A loss function commonly used for classification that measures the discrepancy between predicted class probabilities and true one-hot labels.

Data drift: A change over time in the statistical properties of input data (features) that can degrade model performance if not detected and addressed.

Data lineage: Documentation of the origin, transformations, and movement of data from source to model input, enabling traceability and reproducibility.

Data preprocessing: The suite of operations (cleaning, imputation, encoding, scaling) applied to raw data to make it suitable for model training.

Decision tree: A tree-structured model that splits data on feature thresholds to predict a target; easy to interpret but prone to overfitting without regularization.

Deployment: The process of moving a trained model into a production environment where it can generate predictions in real or batch mode.

Evaluation metric: A quantitative measure (e.g., RMSE, AUC-ROC, Accuracy) used to assess how well a model's predictions match true outcomes.

Feature engineering: The creation, transformation, and selection of input variables that improve a model's predictive power.

Feature store: A centralized repository that version-controls, serves, and documents engineered features for consistent reuse across models.

Hyper-parameter tuning: The systematic search (grid, random, Bayesian) for optimal settings of model parameters that are not learned during training (e.g., learning rate, tree depth).

Imputation: The technique of filling missing values in a dataset using methods such as mean substitution, k-nearest neighbors, or model-based prediction.

Loss function: A mathematical expression that quantifies the error between a model's predictions and the true target, guiding parameter updates during training.

Model evaluation: The stage where a trained model is tested on a hold-out or validation set to compute evaluation metrics and diagnose issues like overfitting.

Model selection: Choosing the most appropriate algorithmic family and specific model architecture based on data characteristics, interpretability needs, and performance criteria.

Overfitting: When a model captures noise or idiosyncrasies in the training data, leading to high training accuracy but poor generalization to new data.

Policy (in RL): A mapping from states to actions ($\pi(a|s)$) that dictates the agent's behavior; the goal is to learn a policy that maximizes expected cumulative reward.

Reinforcement learning (RL): A learning paradigm where an agent interacts with an environment, receives rewards, and optimizes a policy to maximize long-term return.

Reward function: In RL, a scalar feedback signal that evaluates the immediate

desirability of an action taken in a particular state.

Supervised learning: A paradigm that trains models on labeled examples (input-output pairs) to predict the same type of label on unseen data.

Unsupervised learning: A paradigm that discovers hidden structure in data without explicit labels, often via clustering, dimensionality reduction, or anomaly detection.

Validation set: A subset of data held out from training used to tune hyper-parameters and assess model performance before final testing.

XGBoost: An efficient, gradient-boosted tree algorithm (Extreme Gradient Boosting) widely used for tabular data due to its speed, regularization, and strong predictive performance.

IMPORTANT DISCLAIMER

AI-Generated Content Notice

This document has been entirely generated by artificial intelligence technology through the Pustakam Injin platform. While significant effort has been made to ensure accuracy and coherence, readers should be aware of the following important considerations:

- The content is produced by AI language models and may contain factual inaccuracies, outdated information, or logical inconsistencies.
- Information should be independently verified before being used for critical decisions, academic citations, or professional purposes.
- The AI may generate plausible-sounding but incorrect or fabricated information (known as "hallucinations").
- Views and opinions expressed do not necessarily reflect those of the creators, developers, or any affiliated organizations.
- This content should not be considered a substitute for professional advice in medical, legal, financial, or other specialized fields.

Intellectual Property & Usage

This document is provided "as-is" for informational and educational purposes. Users are encouraged to fact-check, cross-reference, and critically evaluate all content. The Pustakam Injin serves as a knowledge exploration tool and starting point for research, not as a definitive source of truth.

Quality Assurance

While the Pustakam Injin employs advanced AI models and formatting techniques to produce professional- quality documents, no warranty is made regarding completeness, reliability, or accuracy. Users assume full responsibility for how they use, interpret, and apply this content.

Generated by: **Pustakam Injin**

Date: February 14, 2026 at 07:39 PM

For questions or concerns about this content, please refer to the Pustakam Injin documentation or contact the platform administrator.