■ Machine Learning

15-Day Intensive Course

Complete Study Notes

2 Hours Per Day · 1 Hour Theory · 1 Hour Practical

Topics Covered:

Python · Statistics · Regression · Classification · Clustering

Neural Networks · CNN · RNN · Feature Engineering · Deployment

From Zero to ML Practitioner

Table of Contents

DAY 1

Introduction to Machine Learning

■ Learning Objectives

✓ Understand what Machine Learning is and why it matters

✓ Distinguish between AI, ML, and Deep Learning

✓ Identify types of ML: Supervised, Unsupervised, Reinforcement

■ THEORY (1 Hour)

What is Machine Learning?

Machine Learning is a subset of Artificial Intelligence where systems learn from data to improve performance without being explicitly programmed. Instead of writing rules manually, we let the algorithm discover patterns.

- AI → Machine Learning → Deep Learning (nested subsets)
- Traditional Programming: Data + Rules → Output
- Machine Learning: Data + Output → Rules (learned model)
- First coined by Arthur Samuel in 1959

```
y = f(x) → model learns f from (x, y) pairs
```

■ *ML is not magic — it finds statistical patterns. Garbage in = garbage out!*

Types of Machine Learning

ML is broadly categorised into three paradigms based on the nature of the learning signal.

- Supervised Learning: Labelled data, predict output (classification, regression)
- Unsupervised Learning: Unlabelled data, find structure (clustering, dimensionality reduction)
- Reinforcement Learning: Agent learns via reward/punishment signals
- Semi-supervised: Mix of labelled and unlabelled data

■ *Most real-world problems use Supervised Learning. Start there!*

ML Workflow

Every ML project follows a standard pipeline regardless of the algorithm used.

- 1. Define the problem & collect data
- 2. Explore & preprocess data (EDA)
- 3. Choose & train a model
- 4. Evaluate & tune the model
- 5. Deploy & monitor the model

■ PRACTICAL (1 Hour) — Setting Up Your ML Environment

Step 1: Install Python & Libraries

```
pip install numpy pandas matplotlib seaborn scikit-learn jupyter
# Verify installation
import sklearn; print(sklearn.__version__)
```

■ Run each command in your terminal.

Step 2: Your First ML Program

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

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

# Split data
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42)

# Train model
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)

# Evaluate
print(f'Accuracy: {model.score(X_test, y_test):.2f}')
```

■ Expected output: Accuracy: 1.00

■■ Practice Tasks

- Explore the Iris dataset — print shape, column names, class counts

- Try k=1, k=5, k=10 for KNN and compare accuracy
- Visualise the data using a scatter plot

DAY 2

Data Preprocessing & Exploratory Data Analysis

■ Learning Objectives

✓ Load and inspect datasets using Pandas

✓ Handle missing values, outliers, and categorical data

✓ Visualise distributions and correlations

■ THEORY (1 Hour)

Why Preprocessing Matters

Raw data is rarely ready for ML. Preprocessing transforms raw data into a clean, structured format that algorithms can effectively learn from.

- Missing values cause errors in most ML algorithms
- Outliers skew model training significantly
- Inconsistent scales lead to biased feature importance
- 'Garbage In, Garbage Out' — quality data = quality model

■ *Spend 60-80% of your project time on data cleaning — it's the most impactful step!*

Handling Missing Values

Choose a strategy based on the percentage of missing data and the feature's importance.

- Drop rows/columns: if > 40% data is missing
- Mean/Median imputation: for numerical features (median for skewed data)
- Mode imputation: for categorical features
- KNN/Model-based imputation: for complex relationships
- Flag + impute: add a binary 'was_missing' column

```
df.fillna(df.median()) # or SimpleImputer(strategy='median')
```

■ *Never impute before splitting train/test — it causes data leakage!*

Feature Scaling

Many algorithms (SVM, KNN, Neural Nets) are sensitive to the scale of features.

- Min-Max Normalization: scales to [0, 1] → (x - min)/(max - min)

- Standardization (Z-score): mean=0, std=1 → (x - μ)/σ
- Use StandardScaler for most algorithms
- Use MinMaxScaler when you need bounded output
- Tree-based methods (RF, XGBoost) don't need scaling

```
z = (x - mean) / std_deviation
```

■ *Fit the scaler ONLY on training data, then transform both train and test.*

■ PRACTICAL (1 Hour) — Data Cleaning & EDA on Titanic Dataset

Step 1: Load & Inspect Data

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

df = pd.read_csv('https://raw.githubusercontent.com/datasciencedojo/
datasets/master/titanic.csv')

print(df.shape) # (891, 12)
print(df.info()) # dtypes & null counts
print(df.describe()) # statistics
print(df.isnull().sum()) # missing values
```

■ Check for nulls in Age, Cabin, Embarked columns.

Step 2: Handle Missing Values & Encode

```
# Fill missing Age with median
df['Age'].fillna(df['Age'].median(), inplace=True)

# Drop Cabin (too many nulls)
df.drop('Cabin', axis=1, inplace=True)

# Encode Sex column
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})

# Correlation heatmap
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.show()
```

■ Note which features correlate with Survived.

■■ Practice Tasks

- Plot survival rate by gender and passenger class
- Detect outliers in Fare using a boxplot
- Create a new feature: FamilySize = SibSp + Parch + 1

DAY 3

Linear & Logistic Regression

■ Learning Objectives

✓ Understand the mathematics behind Linear Regression

✓ Implement Logistic Regression for binary classification

✓ Evaluate models using MSE, R^2, Accuracy, and Confusion Matrix

■ THEORY (1 Hour)

Linear Regression

Linear Regression models the relationship between input features X and a continuous output y as a straight line (or hyperplane in higher dimensions).

- Hypothesis: ■ = w■ + w■x■ + w■x■ + ... + w■x■
- Cost function: MSE = (1/n) Σ(y■ - ■■)²
- Goal: minimise MSE by finding optimal weights w
- Solved via Ordinary Least Squares (OLS) or Gradient Descent
- Assumptions: linearity, independence, homoscedasticity, normality of residuals

```
w = (X^T X)^(-1) X^T y (Normal Equation)
```

■ *Always plot residuals! A pattern in residuals = model assumptions violated.*

Gradient Descent

An iterative optimisation algorithm that updates weights in the direction of steepest descent of the loss function.

- Batch GD: uses all data per update — stable but slow
- Stochastic GD (SGD): uses 1 sample — fast but noisy
- Mini-batch GD: uses small batches — best of both worlds
- Learning rate α controls step size — too large: diverge; too small: slow

```
w := w - α · (∂L/∂w)
```

■ *Normalise features before gradient descent — it speeds up convergence drastically!*

Logistic Regression

Despite the name, Logistic Regression is a classification algorithm. It applies the sigmoid function to map linear output to probabilities [0, 1].

- Sigmoid: σ(z) = 1 / (1 + e^(-z))
- Predict class 1 if P(y=1|x) ≥ 0.5, else class 0
- Loss: Binary Cross-Entropy = -[y·log(■) + (1-y)·log(1-■)]
- Decision boundary is linear in feature space
- Can be extended to multi-class with Softmax (Multinomial LR)

$$P(y=1|x) = \sigma(w^T x) = 1/(1+e^{(-w^T x)})$$

■ *Logistic Regression is a great baseline — always try it first before complex models!*

■ PRACTICAL (1 Hour) — Predicting House Prices & Classifying Diabetes

Step 1: Linear Regression — Boston/California Housing

```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

data = fetch_california_housing()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42)

lr = LinearRegression()
lr.fit(X_train, y_train)
preds = lr.predict(X_test)

print(f'RMSE: {np.sqrt(mean_squared_error(y_test, preds)):.3f}')
print(f'R2: {r2_score(y_test, preds):.3f}')
```

■ R^2 close to 1 = good fit. RMSE = average error in target units.

Step 2: Logistic Regression — Diabetes Classification

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler

# Use Pima Indians dataset (or sklearn breast_cancer)
from sklearn.datasets import load_breast_cancer
X, y = load_breast_cancer(return_X_y=True)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(
X_scaled, y, test_size=0.2, random_state=42)

log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train, y_train)
preds = log_reg.predict(X_test)

print(classification_report(y_test, preds))
print(confusion_matrix(y_test, preds))
```

■ Read precision, recall, F1 from the classification report.

■■ Practice Tasks

- Plot predicted vs actual house prices
- Visualise the confusion matrix as a heatmap
- Try changing the decision threshold (default 0.5) and observe changes

DAY 4

Decision Trees & Model Evaluation

■ Learning Objectives

✓ Understand how Decision Trees split data using Information Gain / Gini

✓ Build and visualise a Decision Tree

✓ Master evaluation metrics: Accuracy, Precision, Recall, F1, ROC-AUC

■ THEORY (1 Hour)

Decision Trees — How They Work

A Decision Tree recursively splits the feature space into regions, creating a tree where each node tests a feature and each leaf holds a prediction.

- Root Node: entire dataset, best split feature chosen here
- Internal Nodes: test a feature against a threshold
- Leaf Nodes: final prediction (class or value)
- Splitting criterion for classification: Gini Impurity or Information Gain (Entropy)
- Splitting criterion for regression: MSE or MAE reduction
- Depth controls complexity — deep trees overfit!

$$Gini = 1 - \Sigma(p_i^2) \mid Entropy = -\Sigma(p_i \log_2 p_i)$$

■ *Decision Trees are very interpretable — great for explaining predictions to stakeholders.*

Overfitting & Pruning

An unconstrained decision tree memorises training data — it perfectly fits training but fails on new data.

- Overfitting: low training error, high test error
- Underfitting: high training error and high test error
- Regularisation techniques: max_depth, min_samples_split, min_samples_leaf
- Pre-pruning: stop growing early (hyperparameter constraints)
- Post-pruning: grow full tree, then remove unnecessary branches

■ *The bias-variance tradeoff is fundamental. Aim for the 'sweet spot' in model complexity.*

Evaluation Metrics Deep Dive

No single metric tells the full story. Choose metrics based on your problem and class balance.

- Accuracy = (TP+TN)/(TP+TN+FP+FN) — misleading with imbalanced classes
- Precision = TP/(TP+FP) — how many predicted positives are correct?
- Recall = TP/(TP+FN) — how many actual positives did we catch?
- F1 Score = 2·(P·R)/(P+R) — harmonic mean, balances P and R
- ROC-AUC: area under ROC curve, threshold-independent, 0.5=random 1.0=perfect

■ *For medical/fraud detection, prioritise Recall (catching all positives matters more).*

■ PRACTICAL (1 Hour) — Decision Tree on Iris & Advanced Evaluation

Step 1: Build & Visualise a Decision Tree

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42)

dt = DecisionTreeClassifier(max_depth=3, random_state=42)
dt.fit(X_train, y_train)

plt.figure(figsize=(15, 8))
plot_tree(dt, feature_names=iris.feature_names,
class_names=iris.target_names, filled=True)
plt.show()
print(f'Test Accuracy: {dt.score(X_test, y_test):.3f}')
```

■ Try max_depth=1, 3, 5, None and compare train vs test accuracy.

Step 2: ROC Curve & AUC

```
from sklearn.metrics import roc_curve, auc
from sklearn.datasets import load_breast_cancer

X, y = load_breast_cancer(return_X_y=True)
```

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

dt = DecisionTreeClassifier(max_depth=5)
dt.fit(X_train, y_train)
probs = dt.predict_proba(X_test)[:, 1]

fpr, tpr, _ = roc_curve(y_test, probs)
roc_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.3f}')
plt.plot([0,1],[0,1],'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```

■ AUC > 0.9 is excellent. Compare DT with Logistic Regression.

■■ Practice Tasks

- Plot feature importances from the Decision Tree
- Build a Decision Tree Regressor on California Housing
- Compare Accuracy vs F1 on an imbalanced dataset

DAY 5

Random Forests & Ensemble Methods

■ Learning Objectives

✓ Understand Bagging, Boosting, and Stacking ensemble techniques

✓ Build Random Forest and Gradient Boosting models

✓ Tune hyperparameters using cross-validation

■ THEORY (1 Hour)

Ensemble Learning

Ensemble methods combine predictions from multiple models to achieve better performance than any single model — the 'wisdom of crowds' principle.

- Bagging (Bootstrap Aggregating): train models on random subsets, average predictions
- Boosting: train models sequentially, each correcting previous errors
- Stacking: use meta-model to combine predictions of base models
- Random Forest = Bagging + Random feature selection at each split
- Reduces variance (Bagging) or bias (Boosting)

■ *Random Forests are often the best 'out-of-the-box' algorithm — start here for tabular data!*

Random Forest Deep Dive

Random Forest builds many decision trees on bootstrapped samples, using a random subset of features at each split, then aggregates (votes/averages) predictions.

- n_estimators: number of trees (more = better, diminishing returns after ~100-500)
- max_features: √n for classification, n/3 for regression (n = num features)
- max_depth: controls tree complexity — None means fully grown
- Out-of-Bag (OOB) score: free validation using samples not in bootstrap
- Feature importance: mean decrease in impurity across all trees

■ *OOB score eliminates the need for a separate validation set — enable with oob_score=True.*

Gradient Boosting (GBM)

Boosting builds trees sequentially, where each tree learns to correct residual errors of the combined previous trees.

• Learning rate (shrinkage): small lr (0.01-0.1) + many trees = better generalisation

• XGBoost, LightGBM, CatBoost: optimised, faster implementations

• Key params: n_estimators, learning_rate, max_depth, subsample

• Early stopping: stop when validation loss stops improving

• Regularisation: L1 (alpha) and L2 (lambda) to prevent overfitting

F(x) = F■(x) + α·h■(x) + α·h■(x) + ... (h = weak learner trees)

■ *LightGBM is ~10x faster than XGBoost for large datasets — use it in competitions!*

■ PRACTICAL (1 Hour) — Random Forest & XGBoost on Titanic

Step 1: Random Forest with Cross-Validation

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV
import numpy as np

# Using preprocessed Titanic data from Day 2
features = ['Pclass','Sex','Age','Fare','SibSp','Parch']
X = df[features]
y = df['Survived']

rf = RandomForestClassifier(n_estimators=100, random_state=42)
scores = cross_val_score(rf, X, y, cv=5, scoring='accuracy')
print(f'CV Accuracy: {scores.mean():.3f} ± {scores.std():.3f}')
```

■ 5-fold CV gives a more reliable estimate than a single train/test split.

Step 2: Hyperparameter Tuning with GridSearchCV

```
param_grid = {
'n_estimators': [50, 100, 200],
'max_depth': [None, 5, 10],
'max_features': ['sqrt', 'log2']
}

grid_search = GridSearchCV(
RandomForestClassifier(random_state=42),
param_grid, cv=5, scoring='accuracy', n_jobs=-1)
```

```
grid_search.fit(X, y)
```

```
print('Best params:', grid_search.best_params_)
```

```
print('Best score: ', grid_search.best_score_)
```

■ Use RandomizedSearchCV for large grids — much faster!

■■ Practice Tasks

- Plot feature importances from the Random Forest
- Compare RF vs GradientBoostingClassifier on the same dataset
- Try XGBoost (pip install xgboost) and compare performance

DAY 6

Support Vector Machines & Kernel Methods

■ Learning Objectives

✓ Understand the maximum-margin hyperplane concept

✓ Apply kernel trick for non-linear classification

✓ Tune SVM hyperparameters C and gamma

■ THEORY (1 Hour)

Support Vector Machines

SVMs find the hyperplane that maximises the margin between classes. Support vectors are the training points closest to the decision boundary.

- Hard-margin SVM: no misclassifications allowed (only linearly separable data)
- Soft-margin SVM: allows some misclassifications (parameter C controls trade-off)
- C (regularisation): high C = small margin, low bias; low C = large margin, high bias
- Support vectors: only these points define the decision boundary
- Works well in high-dimensional spaces and small datasets

Maximise: 2/■w■ subject to y■(w·x■ + b) ≥ 1

■ *SVMs are memory-efficient — only support vectors are stored after training!*

Kernel Trick

Kernels implicitly map data to higher dimensions where it becomes linearly separable, without computing the transformation explicitly.

- Linear kernel: K(x,z) = x·z (for linearly separable data)
- RBF/Gaussian kernel: K(x,z) = exp(-γ■x-z■²) (most commonly used)
- Polynomial kernel: K(x,z) = (x·z + c)^d
- gamma (γ): high = complex boundary, fits training closely; low = smoother boundary
- Rule: try RBF first, then tune C and gamma with GridSearchCV

K(x, z) = ϕ(x)·ϕ(z) (inner product in feature space)

■ *Scale your features! SVM is very sensitive to feature magnitude — always use StandardScaler.*

■ PRACTICAL (1 Hour) — SVM on Non-Linear Data

Step 1: SVM with RBF Kernel

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
import numpy as np

# Non-linear dataset
X, y = make_moons(n_samples=300, noise=0.2, random_state=42)

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

svm = SVC(kernel='rbf', C=1.0, gamma='scale')
svm.fit(X_scaled, y)
print(f'Accuracy: {svm.score(X_scaled, y):.3f}')
```

■ Try kernel='linear' and compare the decision boundary.

Step 2: Visualise Decision Boundary

```
def plot_boundary(model, X, y):
h = 0.02
x_min, x_max = X[:,0].min()-1, X[:,0].max()+1
y_min, y_max = X[:,1].min()-1, X[:,1].max()+1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(X[:,0], X[:,1], c=y, edgecolors='k')
plt.show()

plot_boundary(svm, X_scaled, y)
```

■ Notice how RBF creates a curved boundary unlike the linear kernel.

■■ Practice Tasks

- Tune C and gamma using GridSearchCV
- Apply SVM to the Breast Cancer dataset — compare with Logistic Regression
- Observe how increasing C affects the margin width

DAY 7

K-Nearest Neighbours & Naive Bayes

■ Learning Objectives

✓ Implement KNN for classification and regression

✓ Understand Bayes theorem and Naive Bayes classifier

✓ Apply NB to text classification problems

■ THEORY (1 Hour)

K-Nearest Neighbours (KNN)

KNN is a non-parametric, lazy learning algorithm. It classifies a new point by majority vote of its K nearest neighbours in the training set.

- Distance metrics: Euclidean (default), Manhattan, Minkowski
- k=1: very flexible, high variance; large k: smoother, higher bias
- Lazy learner: no training phase, all computation at prediction time
- Time complexity: O(n·d) per query — slow for large datasets
- Curse of dimensionality: performance degrades in high dimensions

$$d(x,z) = \sqrt{(\Sigma(x_i - z_i)^2)} \text{ (Euclidean distance)}$$

■ *Always normalise features for KNN! Distance is meaningless with inconsistent scales.*

Naive Bayes Classifier

Naive Bayes applies Bayes theorem with the 'naive' assumption that all features are conditionally independent given the class label.

- Bayes theorem: P(C|X) = P(X|C)·P(C) / P(X)
- Gaussian NB: features follow normal distribution (continuous data)
- Multinomial NB: feature counts (text classification, word frequencies)
- Bernoulli NB: binary features (presence/absence)
- Despite 'naive' assumption, works well in practice — especially for text

$$P(y|x_1..x_n) \propto P(y)\cdot\prod P(x_i|y)$$

■ *Naive Bayes is extremely fast and works well with small data — great for spam filters!*

■ PRACTICAL (1 Hour) — KNN on MNIST & Naive Bayes for Spam Detection

Step 1: KNN Digit Classification

```
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

digits = load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42)

# Find best k
for k in [1, 3, 5, 7, 10]:
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
print(f'k={k}: {knn.score(X_test, y_test):.3f}')
```

■ k=3 or k=5 usually gives best performance on this dataset.

Step 2: Naive Bayes SMS Spam Classifier

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline

# Sample messages
texts = ['Free prize click now!', 'Hi are you free tomorrow?',
'Win cash now limited offer', 'Meeting at 3pm today',
'URGENT claim your reward', 'Dinner plans for Friday?']
labels = [1, 0, 1, 0, 1, 0] # 1=spam, 0=ham

pipeline = Pipeline([
('vectorizer', CountVectorizer()),
('classifier', MultinomialNB())
])
pipeline.fit(texts, labels)

test = ['Congratulations you won!', 'See you at the meeting']
```

```
print(pipeline.predict(test)) # [1, 0]
```

■ This is a mini version of real spam filters — extend with real SMS dataset!

■■ Practice Tasks

- Elbow method: plot k vs accuracy to find optimal K
- Visualise sample digit images from the dataset
- Load the real SMS Spam dataset from UCI repository and train NB

DAY 8

Unsupervised Learning — Clustering

■ Learning Objectives

✓ Apply K-Means and Hierarchical clustering algorithms

✓ Evaluate clusters using Silhouette Score and Elbow Method

✓ Understand DBSCAN for density-based clustering

■ THEORY (1 Hour)

K-Means Clustering

K-Means partitions data into K clusters by iteratively assigning points to the nearest centroid and recomputing centroids until convergence.

- Step 1: Randomly initialise K centroids
- Step 2: Assign each point to nearest centroid (Euclidean distance)
- Step 3: Recompute centroids as cluster means
- Step 4: Repeat steps 2-3 until convergence (centroids stop moving)
- K-Means++: smarter initialisation — spread initial centroids far apart

```
WCSS = Σ Σ ■x■ - μ■■² (minimise Within-Cluster Sum of Squares)
```

■ *K-Means can get stuck in local minima — run multiple times (n_init) and take best!*

Choosing K & Evaluation

Selecting the right number of clusters is crucial. Use quantitative methods to guide this choice.

- Elbow Method: plot WCSS vs K — look for the 'elbow' where improvement slows
- Silhouette Score: measures how similar a point is to its own cluster vs others
- Score range: -1 (wrong cluster) to +1 (perfect cluster), 0 = overlapping
- Davies-Bouldin Index: lower is better
- Domain knowledge: sometimes the 'right' K is known from the business context

```
S(i) = (b(i) - a(i)) / max(a(i), b(i))
```

■ *Silhouette score > 0.5 is generally considered a good clustering result.*

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) finds clusters of arbitrary shape and automatically identifies outliers.

- eps (ε): maximum distance between two points to be considered neighbours
- min_samples: minimum points required to form a dense region (core point)
- Core points: ≥ min_samples neighbours within eps
- Border points: within eps of core point but fewer than min_samples neighbours
- Noise points: not reachable from any core point → labelled as -1

■ *DBSCAN is great for anomaly detection — noise points are potential outliers!*

■ PRACTICAL (1 Hour) — Customer Segmentation with K-Means

Step 1: K-Means Elbow & Silhouette

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np

# Generate synthetic customer data
from sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=300, centers=4,
cluster_std=0.6, random_state=42)

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

# Elbow method
inertias = []
silhouettes = []
for k in range(2, 11):
km = KMeans(n_clusters=k, random_state=42, n_init=10)
labels = km.fit_predict(X_scaled)
inertias.append(km.inertia_)
silhouettes.append(silhouette_score(X_scaled, labels))

plt.subplot(1,2,1)
```

```
plt.plot(range(2,11), inertias, 'bo-')
plt.xlabel('K'); plt.ylabel('WCSS'); plt.title('Elbow')
plt.subplot(1,2,2)
plt.plot(range(2,11), silhouettes, 'ro-')
plt.xlabel('K'); plt.ylabel('Silhouette'); plt.title('Silhouette')
plt.tight_layout(); plt.show()
```

■ Both methods should agree on the optimal K.

■■ Practice Tasks

- Colour-code the scatter plot by cluster assignment
- Apply DBSCAN to the same data — compare cluster shapes
- Use K-Means on the Mall Customer dataset (Annual Income vs Spending Score)

DAY 9

Dimensionality Reduction — PCA & t-SNE

■ Learning Objectives

✓ Understand and apply Principal Component Analysis (PCA)

✓ Visualise high-dimensional data using t-SNE and UMAP

✓ Use PCA as a preprocessing step for ML pipelines

■ THEORY (1 Hour)

Why Dimensionality Reduction?

High-dimensional data suffers from the 'curse of dimensionality' — data becomes sparse, distances become meaningless, and algorithms slow down.

- Removes multicollinearity between features
- Reduces storage and computation requirements
- Helps visualise data in 2D/3D
- Can improve model performance by removing noisy features
- Two main types: Linear (PCA, LDA) and Non-linear (t-SNE, UMAP, Autoencoders)

■ *PCA before ML can dramatically speed up training with minimal accuracy loss!*

Principal Component Analysis (PCA)

PCA finds orthogonal directions (principal components) that capture maximum variance in the data, allowing projection to fewer dimensions.

- Step 1: Standardise the data (zero mean, unit variance)
- Step 2: Compute covariance matrix
- Step 3: Compute eigenvectors and eigenvalues of covariance matrix
- Step 4: Sort by eigenvalue (= variance explained)
- Step 5: Project data onto top K eigenvectors
- Explained variance ratio: how much variance each PC captures

$$\Sigma = (1/n)\ X^T\ X \rightarrow \textbf{eigendecomposition} \rightarrow \textbf{top-k eigenvectors}$$

■ *Preserve 95% of variance by selecting enough components — check cumulative explained variance.*

t-SNE for Visualisation

t-SNE (t-Distributed Stochastic Neighbour Embedding) is a non-linear technique primarily used for visualising high-dimensional data in 2D/3D.

- Preserves local structure (nearby points stay nearby)
- Does NOT preserve global structure (distances between clusters not meaningful)
- Perplexity hyperparameter: controls local vs global structure trade-off (5-50)
- Computationally expensive — use on subsets of large datasets
- NOT suitable as a preprocessing step for ML (non-deterministic, no inverse transform)

■ *Always use PCA first to reduce to ~50 dims, then apply t-SNE — much faster!*

■ PRACTICAL (1 Hour) — PCA on MNIST Digits & t-SNE Visualisation

Step 1: PCA for Compression & Visualisation

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
import numpy as np

digits = load_digits()
X, y = digits.data, digits.target

# How many components for 95% variance?
pca_full = PCA()
pca_full.fit(X)
cum_var = np.cumsum(pca_full.explained_variance_ratio_)
n_95 = np.argmax(cum_var >= 0.95) + 1
print(f'Components for 95% variance: {n_95}')

# PCA to 2D for visualisation
pca_2d = PCA(n_components=2)
X_2d = pca_2d.fit_transform(X)

plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_2d[:,0], X_2d[:,1],
c=y, cmap='tab10', alpha=0.6)
plt.colorbar(scatter)
```

```
plt.title('PCA: Digits in 2D')
plt.show()
```

■ Notice some digit classes are well-separated even in 2D!

■■ Practice Tasks

- Plot explained variance ratio curve for all components
- Apply t-SNE with perplexity=30 and compare with PCA 2D plot
- Train a Logistic Regression on full data vs PCA-reduced data — compare accuracy

DAY 10

Introduction to Neural Networks

■ Learning Objectives

✓ Understand the biological inspiration and architecture of Neural Networks

✓ Implement forward pass and backpropagation mathematically

✓ Build a simple neural network with scikit-learn and then Keras

■ THEORY (1 Hour)

Neurons & Perceptrons

Artificial neurons are inspired by biological neurons. They receive weighted inputs, apply an activation function, and produce an output.

- Input layer: receives raw features
- Hidden layers: learn intermediate representations
- Output layer: produces final prediction
- Weights (W): learnable parameters connecting neurons
- Bias (b): allows shifting the activation function
- Activation: introduces non-linearity — without it, NN = linear model

```
z = w■x■ + w■x■ + ... + w■x■ + b → output = activation(z)
```

■ *Number of parameters = (inputs × neurons + neurons) for each layer. Track this!*

Activation Functions

Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond linear decision boundaries.

- Sigmoid: σ(x) = 1/(1+e^-x) → range (0,1), used in output layer for binary
- Tanh: range (-1,1), zero-centred, better than sigmoid for hidden layers
- ReLU: max(0,x) — fast, solves vanishing gradient, most popular for hidden layers
- Leaky ReLU: max(0.01x, x) — prevents 'dying ReLU' problem
- Softmax: converts logits to probabilities for multi-class output

```
ReLU(x) = max(0, x) — simple yet very effective!
```

■ *Use ReLU for hidden layers and Sigmoid/Softmax for output layers.*

Backpropagation

Backpropagation efficiently computes gradients of the loss w.r.t. all weights using the chain rule of calculus, enabling gradient descent to update weights.

- Forward pass: compute activations layer by layer, get loss
- Backward pass: propagate gradients from output to input using chain rule
- Gradient: direction to move weights to decrease loss
- Learning rate controls step size in gradient descent
- Epochs: number of full passes through training data
- Vanishing gradients: deep networks with sigmoid can have tiny gradients — use ReLU!

$$\partial L/\partial w = \partial L/\partial \blacksquare \cdot \partial \blacksquare/\partial z \cdot \partial z/\partial w \text{ (chain rule)}$$

■ *Use Adam optimizer in practice — it adapts learning rates automatically!*

■ PRACTICAL (1 Hour) — Neural Network with Keras on MNIST

Step 1: Build MLP with Keras

```
# pip install tensorflow (includes keras)
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Load MNIST
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess
X_train = X_train.reshape(-1, 784) / 255.0
X_test = X_test.reshape(-1, 784) / 255.0

# Build model
model = keras.Sequential([
keras.layers.Dense(256, activation='relu', input_shape=(784,)),
keras.layers.Dropout(0.3),
keras.layers.Dense(128, activation='relu'),
keras.layers.Dropout(0.3),
keras.layers.Dense(10, activation='softmax')
```

```
])

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

history = model.fit(X_train, y_train,
epochs=10, batch_size=128,
validation_split=0.1)

test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')
```

■ Expected accuracy: ~98% — significantly better than shallow methods!

■■ Practice Tasks

- Plot training & validation accuracy/loss curves
- Experiment: add more layers, change units, try different activations
- Visualise 9 misclassified examples

DAY 11

Convolutional Neural Networks (CNN)

■ Learning Objectives

✓ Understand convolution, pooling, and the CNN architecture

✓ Build a CNN for image classification with Keras

✓ Apply Transfer Learning with pre-trained models

■ THEORY (1 Hour)

Convolutional Layers

Convolutional layers apply learnable filters across the input to detect local features like edges, textures, and shapes, with parameter sharing across locations.

- Filter/Kernel: small matrix (e.g. 3×3) that slides across the image
- Feature map: output of applying a filter — highlights certain patterns
- Stride: how many pixels to move the filter each step (stride=1 or 2)
- Padding: 'same' padding keeps spatial dimensions; 'valid' reduces them
- Multiple filters: each detects a different feature (edge directions, colours, etc.)
- Parameters = filter_size × filter_size × in_channels × num_filters + num_filters

■ *Early CNN layers learn low-level features (edges), deeper layers learn high-level (faces, objects).*

Pooling & CNN Architecture

Pooling layers reduce spatial dimensions (downsampling), making the network more computationally efficient and providing translation invariance.

- Max Pooling: takes maximum value in each patch — preserves dominant features
- Average Pooling: takes average — smoother, less common
- Global Average Pooling: reduces each feature map to single value — used before output
- Typical CNN: Conv→ReLU→Pool→Conv→ReLU→Pool→Flatten→Dense→Output
- Batch Normalisation: normalises layer inputs — faster training, acts as regulariser

■ *Add BatchNormalization after Conv layers — it often replaces the need for Dropout in CNNs!*

Transfer Learning

Transfer Learning leverages features learned on large datasets (like ImageNet) for new tasks, dramatically reducing data and compute requirements.

- Feature Extraction: freeze pre-trained weights, only train final classifier
- Fine-tuning: unfreeze some/all pre-trained layers and train with very low LR
- Popular base models: VGG16, ResNet50, EfficientNet, MobileNet
- ImageNet: 1M+ images, 1000 classes — rich feature representations
- Rule of thumb: small dataset → feature extraction; large dataset → fine-tuning

■ *Transfer learning can achieve 90%+ accuracy with only hundreds of training images!*

■ PRACTICAL (1 Hour) — CNN for CIFAR-10 Image Classification

Step 1: Build CNN with Keras

```
import tensorflow as tf
from tensorflow import keras

(X_train, y_train),(X_test,y_test) = keras.datasets.cifar10.load_data()
X_train, X_test = X_train/255.0, X_test/255.0

model = keras.Sequential([
keras.layers.Conv2D(32, (3,3), activation='relu', padding='same',
input_shape=(32,32,3)),
keras.layers.BatchNormalization(),
keras.layers.Conv2D(32, (3,3), activation='relu', padding='same'),
keras.layers.MaxPooling2D(2,2),
keras.layers.Dropout(0.25),

keras.layers.Conv2D(64, (3,3), activation='relu', padding='same'),
keras.layers.BatchNormalization(),
keras.layers.Conv2D(64, (3,3), activation='relu', padding='same'),
keras.layers.MaxPooling2D(2,2),
keras.layers.Dropout(0.25),

keras.layers.Flatten(),
keras.layers.Dense(512, activation='relu'),
keras.layers.Dropout(0.5),
keras.layers.Dense(10, activation='softmax')
])
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=20, batch_size=64,
validation_data=(X_test, y_test))
```

■ Expected ~80-85% accuracy. Full training takes ~30 min on CPU.

■■ Practice Tasks

- Print model.summary() and count total parameters
- Apply data augmentation (flip, rotate, zoom) using ImageDataGenerator
- Load MobileNetV2 pre-trained on ImageNet and classify a custom image

DAY 12

Recurrent Neural Networks & NLP Basics

■ Learning Objectives

✓ Understand RNNs, LSTMs, and GRUs for sequential data

✓ Apply NLP techniques: tokenization, TF-IDF, word embeddings

✓ Build a sentiment analysis model

■ THEORY (1 Hour)

Recurrent Neural Networks

RNNs process sequential data by maintaining a hidden state that captures information from previous time steps — a 'memory' of past inputs.

- Hidden state: h■ = f(h■■■, x■) — updated at each time step
- Unrolled through time: each time step shares the same weights
- Applications: time series, text, speech, music generation
- Vanishing gradient problem: gradients diminish over long sequences
- Simple RNNs struggle with long-range dependencies (>10-20 steps)

```
h■ = tanh(W■·h■■■ + W■·x■ + b)
```

■ *Use LSTM/GRU instead of vanilla RNN for any real sequence modelling task.*

LSTM & GRU

Long Short-Term Memory (LSTM) networks solve the vanishing gradient problem with a gating mechanism that controls information flow.

- Cell state (C): long-term memory highway
- Forget gate: decides what to remove from cell state
- Input gate: decides what new information to add
- Output gate: decides what to output as hidden state
- GRU: simplified LSTM with only 2 gates — faster, similar performance
- Bidirectional LSTM: processes sequence forward and backward — richer context

■ *GRU trains ~30% faster than LSTM with comparable performance — prefer it for moderate tasks!*

NLP Preprocessing

Text data must be converted to numerical representations before feeding into ML models.

- Tokenization: splitting text into words or subwords
- Stop word removal: remove 'the', 'is', 'a' — low information words
- Stemming/Lemmatization: 'running' → 'run' (reduce to base form)
- Bag of Words (BoW): count vector of word frequencies
- TF-IDF: term frequency × inverse document frequency — down-weights common words
- Word Embeddings (Word2Vec, GloVe): dense vectors capturing semantic meaning

$$\text{TF-IDF}(t,d) = tf(t,d) \times \log(N/df(t))$$

■ *Word2Vec: words with similar meaning have similar embedding vectors — King - Man + Woman ≈ Queen!*

■ PRACTICAL (1 Hour) — Sentiment Analysis on IMDB Reviews

Step 1: LSTM for Sentiment Analysis

```
import tensorflow as tf
from tensorflow import keras

# Load IMDB dataset
max_features = 10000 # vocabulary size
maxlen = 200 # max review length

(X_train, y_train),(X_test,y_test) = keras.datasets.imdb.load_data(
num_words=max_features)

X_train = keras.preprocessing.sequence.pad_sequences(
X_train, maxlen=maxlen)
X_test = keras.preprocessing.sequence.pad_sequences(
X_test, maxlen=maxlen)

model = keras.Sequential([
keras.layers.Embedding(max_features, 128, input_length=maxlen),
keras.layers.LSTM(64, return_sequences=True),
keras.layers.LSTM(32),
keras.layers.Dense(1, activation='sigmoid')
```

```
])

model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, batch_size=128,
validation_data=(X_test, y_test))
```

■ Expected accuracy ~87-89%. Epoch 1 is slowest; subsequent are faster.

■■ Practice Tasks

- Try GRU instead of LSTM — compare speed and accuracy
- Use pre-trained GloVe embeddings as the Embedding layer weights
- Build a TF-IDF + Logistic Regression baseline and compare with LSTM

DAY 13

ML Pipelines, Feature Engineering & Regularisation

■ Learning Objectives

✓ Build end-to-end ML pipelines with scikit-learn

✓ Apply feature engineering techniques to improve model performance

✓ Understand L1, L2 regularisation and Dropout

■ THEORY (1 Hour)

Feature Engineering

Feature engineering is the process of creating new, more informative features from raw data — often the most impactful way to improve model performance.

- Polynomial features: x^2, x^3, $x_■ \times x_■$ — capture non-linear relationships
- Binning: convert continuous to categorical (age groups, price ranges)
- Log/Sqrt transform: reduce skewness in right-skewed distributions
- Interaction features: combine two features that jointly affect the target
- Date features: extract year, month, day, hour, day-of-week from datetime
- Target encoding: replace category with mean target value (use carefully!)

■ *Domain knowledge is the key to great feature engineering — talk to domain experts!*

Regularisation

Regularisation adds a penalty term to the loss function to discourage large weights, preventing overfitting by promoting simpler models.

- L1 (Lasso): penalty = $\lambda \Sigma |w_■|$ — drives some weights to exactly zero (feature selection!)
- L2 (Ridge): penalty = $\lambda \Sigma w_■^2$ — shrinks all weights toward zero, none exactly zero
- Elastic Net: combination of L1 and L2
- Dropout: randomly zero out neurons during training — prevents co-adaptation
- Early Stopping: stop training when validation loss stops improving
- Larger λ = stronger regularisation = simpler model

$$L = MSE + \lambda \Sigma w_■^2 \text{ (Ridge loss)}$$

■ *When in doubt, use L2 regularisation — it rarely hurts and often helps!*

scikit-learn Pipelines

Pipelines chain preprocessing steps and models together, preventing data leakage and making code cleaner and more reproducible.

- Ensures consistent preprocessing on train and test data
- Prevents data leakage (scaler fitted only on training data)
- Simplifies cross-validation and hyperparameter tuning
- Can include custom transformers using FunctionTransformer
- Pipeline.fit() → calls fit_transform on each step, then fit on estimator
- Pipeline.predict() → calls transform on each step, then predict

■ *Always use Pipelines in production — they prevent the most common ML bugs!*

■ PRACTICAL (1 Hour) — End-to-End Pipeline with Feature Engineering

Step 1: Complete Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

# Define feature types
num_features = ['Age', 'Fare', 'SibSp', 'Parch']
cat_features = ['Pclass', 'Sex', 'Embarked']

# Pipelines for each type
num_pipeline = Pipeline([
('imputer', SimpleImputer(strategy='median')),
('scaler', StandardScaler())
])

cat_pipeline = Pipeline([
('imputer', SimpleImputer(strategy='most_frequent')),
('encoder', OneHotEncoder(handle_unknown='ignore'))
])
```

```
# Combine with ColumnTransformer
preprocessor = ColumnTransformer([
('num', num_pipeline, num_features),
('cat', cat_pipeline, cat_features)
])

# Full pipeline
full_pipeline = Pipeline([
('preprocessor', preprocessor),
('model', RandomForestClassifier(n_estimators=100, random_state=42))
])

scores = cross_val_score(full_pipeline, X, y, cv=5)
print(f'CV Accuracy: {scores.mean():.3f} ± {scores.std():.3f}')
```

■ This pattern works for any tabular dataset — bookmark it!

■■ Practice Tasks

- Add PolynomialFeatures to the numerical pipeline
- Use Pipeline with Lasso regression — observe which coefficients go to zero
- Use GridSearchCV on the full pipeline to tune model hyperparameters

DAY 14

Model Deployment & MLOps Basics

■ Learning Objectives

✓ Save and load trained ML models

✓ Build a REST API for model serving with Flask

✓ Understand MLOps concepts: versioning, monitoring, CI/CD

■ THEORY (1 Hour)

Model Serialisation

After training, models must be saved (serialised) to disk so they can be loaded and used for predictions in production without retraining.

- Joblib: best for scikit-learn models (more efficient for large numpy arrays)
- Pickle: Python's built-in serialisation — works for any Python object
- ONNX: open format for cross-framework model exchange
- TensorFlow SavedModel / .h5: Keras/TF models
- PyTorch .pt / .pth: state_dict or full model
- Always save the preprocessing pipeline alongside the model!

■ *Pickle has security risks — never unpickle untrusted data! Use joblib for ML models.*

Model Serving with APIs

Production ML models are typically served as REST APIs that accept input data and return predictions over HTTP.

- REST API: stateless, uses HTTP methods (POST for predictions)
- JSON: standard data format for API requests/responses
- Flask: lightweight Python web framework, easy for ML APIs
- FastAPI: modern, faster, auto-generates OpenAPI docs
- Docker: containerise your model API for consistent deployment
- Cloud deployment: AWS SageMaker, GCP Vertex AI, Azure ML, Hugging Face Spaces

■ *FastAPI is replacing Flask for new ML APIs — it's faster and has automatic validation!*

MLOps Concepts

MLOps (Machine Learning Operations) applies DevOps principles to ML systems to enable reliable, scalable, and reproducible ML in production.

- Data versioning: track dataset changes (DVC — Data Version Control)
- Experiment tracking: log parameters, metrics, artifacts (MLflow, Weights & Biases)
- Model registry: version and stage models (dev → staging → production)
- CI/CD for ML: automate testing, retraining, and deployment pipelines
- Monitoring: detect data drift, model degradation, and concept drift
- Feature store: centralised repository for ML features

■ *Start with MLflow experiment tracking on day 1 of any ML project — you'll thank yourself later!*

■ PRACTICAL (1 Hour) — Saving a Model & Building a Flask API

Step 1: Save & Load Model

```
import joblib
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Train model
X, y = load_iris(return_X_y=True)
model = RandomForestClassifier(n_estimators=100)
model.fit(X, y)

# Save
joblib.dump(model, 'iris_model.pkl')
print('Model saved!')

# Load and predict
loaded_model = joblib.load('iris_model.pkl')
sample = [[5.1, 3.5, 1.4, 0.2]]
prediction = loaded_model.predict(sample)
print(f'Prediction: {prediction}') # [0] = setosa
```

■ The loaded model behaves exactly like the original.

Step 2: Flask REST API

```
# app.py
```

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

app = Flask(__name__)
model = joblib.load('iris_model.pkl')

@app.route('/predict', methods=['POST'])
def predict():
data = request.json
features = np.array(data['features']).reshape(1, -1)
prediction = model.predict(features)
return jsonify({'prediction': int(prediction[0])})

if __name__ == '__main__':
app.run(debug=True, port=5000)

# Test with curl:
# curl -X POST http://localhost:5000/predict \
# -H 'Content-Type: application/json' \
# -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

■ Test the API from terminal or use Postman/Thunder Client.

■■ Practice Tasks

- Add error handling for invalid inputs in the Flask API
- Install MLflow, train a model, and log parameters/metrics/model
- Containerise the Flask app with Docker (create Dockerfile)

DAY 15

Capstone — End-to-End ML Project

■ Learning Objectives

✓ Apply the complete ML workflow from data to deployment

✓ Build, evaluate, and compare multiple models

✓ Document and present your ML project professionally

■ THEORY (1 Hour)

Project Workflow Review

A complete ML project follows a structured process. Today we integrate everything learned over the past 14 days into one end-to-end project.

- 1. Problem Definition: What are we predicting? What metric matters?
- 2. Data Collection & EDA: Load, explore, visualise, understand the data
- 3. Preprocessing & Feature Engineering: Clean, encode, scale, create features
- 4. Model Selection & Training: Try multiple algorithms, use cross-validation
- 5. Hyperparameter Tuning: GridSearch/RandomSearch on best model(s)
- 6. Final Evaluation: Test set performance, confusion matrix, feature importance
- 7. Deployment: Save model, build API, document for handover

■ *Document everything! Future-you will thank present-you for clear comments and README files.*

Choosing the Right Algorithm

Algorithm selection depends on data size, feature types, interpretability needs, and latency requirements.

- Small data (<1K samples): SVM, Naive Bayes, Logistic Regression
- Medium data (1K-100K): Random Forest, XGBoost, LightGBM (usually best)
- Large data (>100K): Neural Networks, LightGBM, online learning
- Image data: CNN (or pre-trained ResNet/EfficientNet)
- Sequential/text data: LSTM, GRU, Transformers (BERT)
- Need interpretability: Logistic Regression, Decision Tree, SHAP on any model

■ *XGBoost + feature engineering wins ~80% of structured data Kaggle competitions!*

Next Steps After This Course

This 15-day course is just the beginning. Here's your roadmap for continued learning.

- Practice: Kaggle competitions — start with Titanic, House Prices, MNIST
- Mathematics: Linear Algebra, Calculus, Probability & Statistics (3Blue1Brown, MIT OCW)
- Deep Learning: fast.ai, DeepLearning.AI specialisation (Coursera)
- NLP: Hugging Face course — transformers, BERT, GPT fine-tuning
- MLOps: Full Stack Deep Learning course — production ML systems
- Research: Read papers on arxiv.org — start with classic papers (ResNet, Attention is All You Need)

■ *Consistency beats intensity. 1 hour of practice every day beats a 10-hour weekend session!*

■ PRACTICAL (1 Hour) — Capstone: Predicting Customer Churn

Step 1: Full Pipeline Implementation

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, roc_auc_score
import joblib

# 1. Load data (Telco Customer Churn - available on Kaggle)
# df = pd.read_csv('telco_churn.csv')

# 2. EDA
# print(df.head(), df.info(), df.describe())
# df['Churn'].value_counts(normalize=True).plot(kind='bar')

# 3. Feature Engineering
```

```
# df['TotalChargesPerMonth'] = df['TotalCharges'] / (df['tenure'] + 1)

# 4. Define features
# num_cols = ['tenure', 'MonthlyCharges', 'TotalCharges']
# cat_cols = ['Contract', 'InternetService', 'PaymentMethod']

# 5. Pipeline (same as Day 13 structure)

# 6. Compare models
models = {
'Logistic Regression': LogisticRegression(max_iter=1000),
'Random Forest': RandomForestClassifier(n_estimators=200),
'Gradient Boosting': GradientBoostingClassifier(n_estimators=200)
}

for name, model in models.items():
scores = cross_val_score(model, X, y, cv=5, scoring='roc_auc')
print(f'{name}: {scores.mean():.3f} ± {scores.std():.3f}')
```

■ Pick the best model, tune it, save it, and build a simple API!

Step 2: Project Documentation Template

```
# README.md template
## Customer Churn Prediction

### Problem
Predict which customers will cancel subscription (churn)

### Dataset
Telco Customer Churn — 7043 rows, 21 features

### Approach
1. EDA → 26.5% churn rate (class imbalance)
2. Feature engineering: TotalChargesPerMonth, tenure_group
3. Models tested: LR, RF, GBM
4. Best: GBM — AUC 0.854

### Results
| Model | AUC | F1 |
|-------|-----|----|
| Logistic Regression | 0.841 | 0.61 |
| Random Forest | 0.849 | 0.63 |
| Gradient Boosting | 0.854 | 0.65 |
```

```
### Deployment

Flask API available at /predict endpoint
```

■ A good README makes your project stand out on GitHub!

■■ Practice Tasks

- Complete the full churn prediction pipeline end-to-end
- Create a GitHub repository with your code and README
- Deploy to Hugging Face Spaces or Railway.app for a live demo

■ Quick Reference — 15 Days at a Glance

Day	Topic	Key Algorithms
1	Intro to ML	KNN, ML workflow
2	Data Preprocessing	Imputation, Scaling, EDA
3	Regression	Linear Reg, Logistic Reg
4	Decision Trees	CART, Gini, Entropy, ROC
5	Ensemble Methods	Random Forest, GBM, XGBoost
6	SVMs	SVC, RBF Kernel, C, gamma
7	KNN & Naive Bayes	KNN, GaussianNB, MultinomialNB
8	Clustering	K-Means, DBSCAN, Silhouette
9	Dimensionality	PCA, t-SNE, UMAP
10	Neural Networks	MLP, Backprop, Adam, Dropout
11	CNN	Conv2D, MaxPool, Transfer Learning
12	RNN & NLP	LSTM, GRU, TF-IDF, Embeddings
13	Pipelines & Reg.	Pipeline, L1/L2, Feature Eng.
14	Deployment	Flask, joblib, MLflow, Docker
15	Capstone Project	End-to-End, All techniques

Good luck on your Machine Learning journey! ■ Practice daily, experiment fearlessly, and never stop learning.