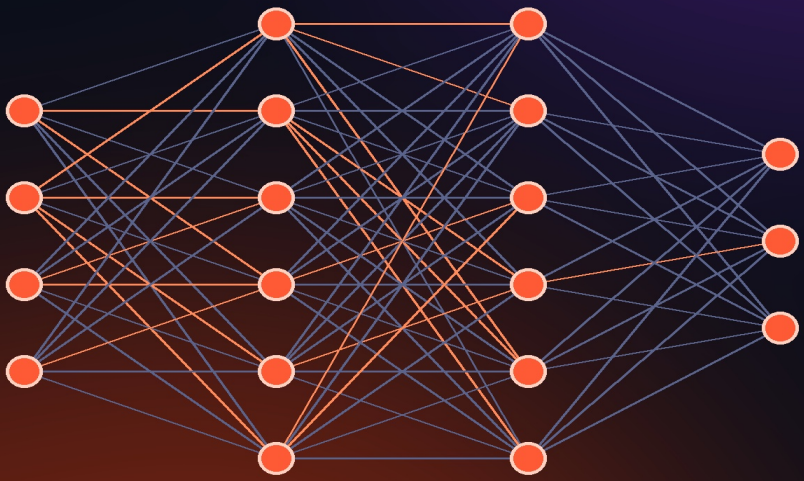
AI MODEL MAKING: BASIC TO ADVANCED · BOOK 1

Build Your First AI Models

A Hands-On Beginner's Guide to Machine Learning with Python



10 Projects

Full Python Code

Exercises + Answers

SACHIN TEKRIWAL

AI MODEL MAKING: BASIC TO ADVANCED · BOOK 1

Build Your First AI Models

A Hands-On Beginner's Guide to Machine Learning with Python, from Zero to a Working AI API

Sachin Tekriwal

AI Teacher · AI Engineer · Automation Expert
Founder, NKBWorks

Copyright © 2026 Sachin Tekriwal. All rights reserved.

No part of this book may be reproduced or shared in any form without written permission from the author, except for short quotations in reviews.

The code examples in this book may be used in your own personal and commercial projects. Republishing the code or text as tutorials, courses or books is not permitted.

The information in this book is provided for learning. Software libraries change over time; if a command does not work, check the library's current documentation. The author is not responsible for losses arising from the use of this material.

Website: https://sachintekriwal.com
Contact: sachintekriwalai@gmail.com

First edition, 2026.

Contents

Introduction 5

Chapter 1. What Is an AI Model? A Simple Explanation for Beginners 7

Chapter 2. Setting Up Your AI Workspace: Python, Jupyter and Google Colab 13

Chapter 3. Data Is Fuel: NumPy and Pandas Basics for AI 18

Chapter 4. Your First Machine Learning Model: Linear Regression 24

Chapter 5. Classification: Teaching a Model to Say Yes or No 29

Chapter 6. Train, Test, Validate: How to Know Your Model Really Works 34

Chapter 7. Decision Trees and Random Forests: Models You Can Explain 39

Chapter 8. Overfitting vs. Underfitting: The Most Important Idea in ML 44

Chapter 9. Feature Engineering: Making Your Data Smarter 49

Chapter 10. Save, Load and Serve Your Model as an API 55

What You Have Built 60

Appendix: Python Essentials 61

Glossary 64

About the Author 68

Introduction

Artificial intelligence is no longer only for big tech companies. A shop owner can forecast sales, a coaching institute can predict which students need help, and a developer in a small town can build and sell an AI service to the world. What they all need is the ability to build a model, test it honestly and put it to work.

That is what this book teaches. It is the first book in the series *AI Model Making: Basic to Advanced*. You will start from the question "What is an AI model?" and finish with your own trained model running as a web API that websites and apps can call.

Who this book is for

- Students and freshers who want a practical start in AI and machine learning.
- Developers who know some programming and want to add AI to their skills.
- Business owners and professionals who want to understand what AI can do with their data.

You need basic computer skills. Some Python helps, but every line of code is explained.

How each chapter works

1. **Concepts** in plain language, with real business examples.
2. **Starter code** to type and run.

3. **Full code**: a complete, more realistic version.
4. **Common mistakes** that beginners and even professionals make.
5. **An exercise with its answer**, and a **mini project** for your own data.
6. **Key terms** to remember. All of them are collected in the glossary at the end.

What you need

A computer with internet access. Chapter 2 shows two ways to set up: Google Colab in your browser (free, nothing to install), or Python on your own computer. A GPU is *not* needed for this book.

The series

1. **Build Your First AI Models** (this book): machine learning foundations
2. Neural Networks with PyTorch
3. Transformers and LLMs: GPT, LoRA and RAG
4. Generative AI: Images, Voice and Video Models
5. From Model to Money: Deploy and Sell AI as SaaS

Let's build your first model.

CHAPTER 1

What Is an AI Model? A Simple Explanation for Beginners

Understand what an AI model really is, how it learns from data, and the 3 parts every model has.

In this chapter you will learn:

- What a model is in plain words
- Training vs. using a model
- Inputs, weights and outputs

A model is a learned function

An AI model is a function that turns an input into an output. The difference from normal code is that we do not write the rules by hand. We show the model many examples, and it adjusts its internal numbers, called **weights**, until its answers match the examples.

Training and inference

There are two phases. **Training** is when the model learns from data and its weights change. **Inference** is when the trained model is used to answer new questions and the weights stay fixed. ChatGPT, a spam filter and a face unlock feature all work this way.

The three parts

Every model has: (1) an **architecture**, the shape of the function; (2) **weights**, the learned numbers; (3) a **loss function**, which measures how wrong the model is so training knows which way to improve.

Rules vs. learning

Traditional software is written as rules: *if the amount is above ₹50,000 and the city is new, flag the payment*. That works until the world changes or the rules become too many to write. Machine learning flips this. We give examples of inputs with the correct outputs, and the computer finds the rules itself. The finished rules are stored as numbers inside the model. This is why AI shines at problems humans find hard to describe, such as recognising a face, understanding speech or predicting which customer will leave.

Types of learning

Supervised learning uses labelled examples (a photo plus the word 'cat'). **Unsupervised learning** finds patterns in unlabelled data, such as grouping customers by behaviour. **Reinforcement learning** learns by trial and reward, like a game-playing agent. Modern large language models mix these: first they learn from huge amounts of unlabelled text (self-supervised), then they are tuned with human examples and feedback.

What 'learning' looks like in numbers

In the free code, the weight starts at 0. Each pass, it is nudged in the direction that reduces the error. This nudge-by-error idea is called **gradient descent**, and it is the same engine inside a model with billions of weights. The only differences are scale and the shape of the function.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
# The smallest "model": learn y = w * x from examples
xs = [1, 2, 3, 4]
ys = [2, 4, 6, 8]          # the hidden rule is y = 2x
w = 0.0                    # the weight starts as a guess
for step in range(100):
    for x, y in zip(xs, ys):
        error = w * x - y
        w -= 0.01 * error * x   # nudge w to reduce the error
print("learned w =", round(w, 3))   # about 2.0
```

Full Code: Learning y = w·x + b with a Loss Curve

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
# A complete mini training program: weight + bias, loss tracking, and a
prediction
xs = [1, 2, 3, 4, 5]
ys = [5, 7, 9, 11, 13]        # hidden rule: y = 2x + 3
w, b, lr = 0.0, 0.0, 0.01
history = []
for epoch in range(2000):
    total = 0.0
    for x, y in zip(xs, ys):
```

```
        pred = w * x + b
        err = pred - y
        total += err ** 2
        w -= lr * err * x       # gradient of squared error w.r.t. w
(up to a factor 2)
        b -= lr * err           # gradient w.r.t. b
    history.append(total / len(xs))
    if epoch % 400 == 0:
        print(f"epoch {epoch:4d}  loss {history[-1]:.4f}  w={w:.3f}
b={b:.3f}")
print(f"Final model: y = {w:.2f}x + {b:.2f}")
print("Prediction for x=10:", round(w * 10 + b, 2))   # about 23
```

How the Full Code Works

1. We store five example inputs (xs) and their correct outputs (ys). The hidden rule is y = 2x + 3, but the program is never told this.
2. w and b start at 0, so the first predictions are all 0 and very wrong. lr is the learning rate, the size of each nudge.
3. For each example we predict, measure the error (prediction minus truth), and add the squared error to a running total.
4. w is nudged by error × x, because a bigger input has more influence on the error. b is nudged by the error alone.
5. Every 400 epochs we print the average loss. Watch it fall towards zero while w moves towards 2 and b towards 3.
6. Finally we use the learned rule to predict x = 10, an input the model never saw. This is inference.

Real-World Case: A bakery's oven timer

A small bakery logged the weight of each cake and the minutes it needed to bake. Instead of guessing, the owner fitted the same kind of straight line you just trained: minutes = w × weight + b. The model learned that each extra 100 g needed about 4 more minutes. It is a tiny model, but it removed guesswork, reduced burnt cakes, and showed the staff that 'AI' can start very small.

Common Mistakes

- Thinking the model 'understands' like a human. It finds statistical patterns in data, nothing more.
- Using a learning rate that is too big, so the weight jumps around and never settles.
- Believing more code makes better AI. Better data usually matters more.

Exercise

Change ys to [3, 6, 9, 12] and run again. What value of w does the model learn?

> **Answer:** With ys = [3, 6, 9, 12] the hidden rule is y = 3x, so the model learns w ≈ 3.0.

Mini Project

Collect 10 pairs of numbers from your life, such as hours studied and test marks, and train the full code on them. Does a straight line fit?

Key Terms

Model
A function with learnable numbers that maps inputs to outputs.

Weight
A learned number inside a model.

Loss
A score of how wrong the model is.

Gradient descent
Repeatedly nudging weights in the direction that lowers the loss.

> **Go further:** The ready-to-use production version of this project, *AI Basics Starter Kit*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 2

Setting Up Your AI Workspace: Python, Jupyter and Google Colab

Get a free, ready-to-use environment for building AI models in 10 minutes, with or without a GPU.

In this chapter you will learn:

- Install Python and the key libraries
- Use Google Colab for a free GPU
- Organise an AI project folder

Two easy options

Option 1, Google Colab: open colab.research.google.com in a browser. Nothing to install, and you can switch on a free GPU from Runtime → Change runtime type. **Option 2, your own PC:** install Python 3.11 or newer, then create a virtual environment so each project keeps its own libraries.

The core libraries

NumPy for numbers, Pandas for tables, Matplotlib for charts, scikit-learn for classic ML, and PyTorch for deep learning. These five cover 90% of this blog series.

A clean project layout

Keep data/, notebooks/, src/ and models/ folders. Never commit large datasets or secret keys to Git. Use a requirements.txt file so anyone can reproduce your setup.

Virtual environments, explained

Each project can need different library versions. A virtual environment is a private folder of libraries for one project, so updating one project never breaks another. Always activate it before installing anything. If something gets badly broken, delete the folder and create it again; your code is safe.

GPU or CPU?

You do not need a GPU for Book 1 at all; scikit-learn runs on any laptop. GPUs matter from neural networks onwards. Colab gives a free GPU with time limits. For long training, rent cloud GPUs by the hour. Check which CUDA version your PyTorch build expects; the PyTorch website gives the exact install command for your system.

Notebooks vs. scripts

Notebooks are excellent for exploring: run one cell, look at the result, change and repeat. Scripts (.py files) are better for anything you will run again, schedule or deploy. A good habit: explore in a notebook, then move the working code into src/ as functions.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
# Run in a terminal (Windows: use "py" instead of "python3")
python3 -m venv ai-env
source ai-env/bin/activate        # Windows: ai-env\Scripts\activate
pip install numpy pandas matplotlib scikit-learn torch jupyter
python3 -c "import torch; print('GPU available:',
torch.cuda.is_available())"
```

Full Code: Environment Check Script

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
# check_env.py  -  run: python check_env.py
import importlib, platform, sys
print("Python :", sys.version.split()[0], "on", platform.system())
for name in ["numpy", "pandas", "matplotlib", "sklearn", "torch"]:
    try:
        mod = importlib.import_module(name)
        print(f"{name:11s}: OK  version {getattr(mod, '__version__',
'?')}")
    except ImportError:
        print(f"{name:11s}: MISSING  ->  pip install {'scikit-learn' if
name == 'sklearn' else name}")
try:
    import torch
    if torch.cuda.is_available():
        print("GPU    :", torch.cuda.get_device_name(0))
    else:
        print("GPU    : not available (fine for Book 1)")
except ImportError:
    pass
```

How the Full Code Works

1. importlib.import_module tries to load each library by name, so one script can check many libraries.
2. If a library is missing, the script prints the exact pip command to install it. Note that scikit-learn is imported as 'sklearn'.
3. getattr(mod, '__version__', '?') reads the version safely, even if a library does not expose one.
4. The last block checks whether PyTorch can see a GPU and prints its name. Book 1 runs fine without one.

Real-World Case: Setting up a computer lab

A training institute with 30 computers wanted every student to have the same AI setup. The trainer created one requirements.txt and this check script. On day one, students ran pip install -r requirements.txt and then the check. Any machine that printed MISSING was fixed in minutes, instead of hours of 'it works on my computer' problems during class.

Common Mistakes

- Installing libraries without activating the virtual environment, so they go to the wrong place.
- Mixing pip and conda in the same environment.
- Saving secret API keys inside notebooks that are then shared.

Exercise

Open Google Colab, enable the GPU, and run the last line. Does it print True?

> **Answer:** On Colab with GPU enabled it prints True. On a laptop without an NVIDIA GPU it prints False, which is fine for now.

Mini Project

Create the folder layout data/, notebooks/, src/, models/ for a project of your own and add a requirements.txt.

Key Terms

Virtual environment
A private set of libraries for one project.

Jupyter notebook
An interactive document mixing code, output and notes.

CUDA
NVIDIA's platform that lets software use the GPU.

requirements.txt
A list of libraries needed to rebuild an environment.

> **Go further:** The ready-to-use production version of this project, *AI Dev Environment Pack*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 3

Data Is Fuel: NumPy and Pandas Basics for AI

Learn how to load, clean and explore data, the most important step in building any AI model.

In this chapter you will learn:

- Arrays with NumPy
- Tables with Pandas
- Finding and fixing missing values

Why data matters most

A model can only be as good as its data. Most real AI work is spent collecting, cleaning and understanding data, not writing model code.

NumPy and Pandas

NumPy gives fast arrays and maths. Pandas gives DataFrames, which are like Excel sheets in Python. You load a CSV, look at the columns, remove bad rows and create new features.

Quick health check

Before training, always check: the number of rows, missing values, data types, and the balance of the label you want to predict.

Exploring data step by step

Start every dataset with five questions: How many rows and columns? What does each column mean? Which values are missing? What are the typical ranges? Is the thing I want to predict balanced? df.head(), df.info() and df.describe() answer most of these in seconds.

Cleaning choices

Missing numbers can be filled with the median, which is less affected by extreme values than the mean. Missing categories can get a new value such as 'Unknown'. If a column is mostly empty, dropping it is often better. Always write down what you changed, because the same cleaning must happen to new data in production.

Groupby: the business question tool

df.groupby('city')['sales'].sum() answers 'which city sells the most?' in one line. Grouping and comparing is how you find the patterns your model will later learn, and it often gives the client useful insights before any AI is built.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
import pandas as pd
df = pd.read_csv("https://raw.githubusercontent.com/mwaskom/seaborn-
data/master/titanic.csv")
print(df.shape)                    # rows, columns
print(df.isna().sum())             # missing values per column
df["age"] = df["age"].fillna(df["age"].median())
print(df["survived"].value_counts(normalize=True))   # label balance
```

Full Code: Data Health Report

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
import pandas as pd
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/
titanic.csv"
df = pd.read_csv(url)
print("Shape:", df.shape)
report = pd.DataFrame({
    "type": df.dtypes.astype(str),
    "missing_%": (df.isna().mean() * 100).round(1),
    "unique": df.nunique(),
})
print(report.sort_values("missing_%", ascending=False))
df["age"] = df["age"].fillna(df["age"].median())
df["embarked"] = df["embarked"].fillna(df["embarked"].mode()[0])
df = df.drop(columns=["deck"])             # mostly empty
print("\nSurvival rate by class and sex:")
print(df.groupby(["pclass", "sex"])
["survived"].mean().round(2).unstack())
df.to_csv("titanic_clean.csv", index=False)
```

How the Full Code Works

1. pd.read_csv loads the Titanic passenger list straight from the internet into a DataFrame.
2. We build a small report table: the data type of each column, the percentage of missing values, and the number of unique values.
3. Sorting by missing_% shows at once which columns need attention: 'deck' is mostly empty and 'age' has gaps.
4. Age is filled with the median and 'embarked' with its most common value (the mode). 'deck' is dropped.
5. groupby with unstack creates a survival table by class and sex. This is a business-style insight before any model is built.
6. The cleaned data is saved, so the next step starts from clean data and the cleaning is repeatable.

Real-World Case: A coaching institute's student records

An institute exported three years of student records: attendance, test marks, fee status and final result. The health report showed that 'previous school' was 60% empty and that marks were stored as text in one year's data. After fixing types and dropping the empty column, a simple groupby showed that students with attendance below 70% failed four times more often. That single insight changed the institute's follow-up policy, even before any model was trained.

Common Mistakes

- Filling missing values using statistics from the whole dataset, including the test part (data leakage).
- Deleting every row with any missing value and losing most of the data.
- Not checking data types, so numbers stored as text break the model.

Exercise

Find which column has the most missing values. Would you fill it or drop it? Why?

> **Answer:** 'deck' has the most missing values (about 77%). It is usually better to drop it, or to replace it with a simple 'has_deck' yes/no column.

Mini Project

Export a sales or student list from Excel as CSV and run the health report on it.

Key Terms

DataFrame
A table of data in Pandas.

Missing value (NaN)
An empty cell.

Median
The middle value; robust to outliers.

Class balance
How evenly the target labels are spread.

> **Go further:** The ready-to-use production version of this project, *Data Cleaning Toolkit*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 4

Your First Machine Learning Model: Linear Regression

Predict a number, such as a house price, with your first real machine learning model in scikit-learn.

In this chapter you will learn:

- What regression means
- Train a model in 4 lines
- Read the model's error

Predicting numbers

Regression predicts a number: price, sales or temperature. Linear regression draws the best straight line (or flat surface) through your data points.

Fit and predict

In scikit-learn every model follows the same pattern: create it, call fit(X, y) to train, and call predict(X) to use it. Learn this once and you can use hundreds of models.

Measuring error

Mean Absolute Error (MAE) is the average distance between predictions and real values, in the same unit as your target, so it is easy to explain to a client.

How the line is chosen

Linear regression chooses weights that make the total squared error as small as possible. Squaring punishes big mistakes more than small ones. For linear regression there is even a direct formula, so training is instant.

Reading the coefficients

Each input gets a coefficient. A coefficient of 0.4 on 'median income' means that, holding other inputs fixed, one more unit of income raises the predicted price by 0.4 units. This makes linear models easy to explain, which is valuable in finance and government projects.

When a straight line is not enough

Real prices do not rise in a straight line forever. Add features such as squared terms, or move to tree models (Chapter 7). Always compare against a simple baseline, such as predicting the average price; your model must beat it.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
X, y = fetch_california_housing(return_X_y=True)
model = LinearRegression().fit(X, y)
print("MAE:", mean_absolute_error(y, model.predict(X)))
```

Full Code: Honest Evaluation with a Baseline

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor
from sklearn.metrics import mean_absolute_error, r2_score
data = fetch_california_housing(as_frame=True)
X, y = data.data, data.target
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
random_state=42)
base = DummyRegressor(strategy="mean").fit(Xtr, ytr)
lin = LinearRegression().fit(Xtr, ytr)
for name, m in [("baseline", base), ("linear", lin)]:
    p = m.predict(Xte)
    print(f"{name:9s} MAE={mean_absolute_error(yte, p):.3f}
R2={r2_score(yte, p):.3f}")
for col, coef in sorted(zip(X.columns, lin.coef_), key=lambda t: -
abs(t[1])):
    print(f"{col:12s} {coef:+.3f}")
```

How the Full Code Works

1. as_frame=True returns the data as a DataFrame with column names, so the coefficients are readable.
2. train_test_split keeps 20% of the houses hidden for an honest test.
3. DummyRegressor always predicts the average price. It is the baseline our real model must beat.
4. Both models are scored with MAE and R^2 on the hidden test set only.
5. The coefficients are sorted by size, so you see which inputs move the prediction the most.

Real-World Case: Rent estimates for a property dealer

A property dealer wanted quick rent estimates for new listings. Using 400 past listings with area, floor, age of building and distance to the station, a linear regression gave estimates within about 10% for most flats. More importantly, the coefficients gave the dealer a simple story to tell owners: 'Each 100 sq ft adds about ₹2,000 a month; every kilometre from the station takes away about ₹1,200.' Explainability won the client's trust.

Common Mistakes

- Reporting error on training data only.
- Forgetting the target's unit (here, hundreds of thousands of dollars).
- Reading a coefficient as cause and effect; it shows association only.

Exercise

This measures error on the same data it trained on. Why might that be too optimistic? (Post 6 answers this.)

> **Answer:** Measuring on the training data is optimistic because the model has already seen those exact answers. Use a held-out test set, as in the full code.

Mini Project

Predict rent from area, floor and distance to the station using 30 listings you collect online.

Key Terms

Regression
Predicting a number.

MAE
Average absolute error.

R^2
Share of variation the model explains; 1.0 is perfect.

Baseline
A very simple model to beat.

> **Go further:** The ready-to-use production version of this project, *House Price Predictor App*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 5

Classification: Teaching a Model to Say Yes or No

Build a spam-or-not, pass-or-fail type model with logistic regression and understand probabilities.

In this chapter you will learn:

- Classification vs. regression
- Predicting probabilities
- The decision threshold

Predicting categories

Classification predicts a label: spam or not spam, fraud or safe, cat or dog. Logistic regression is the simplest classifier and is still widely used in banking and healthcare.

Probabilities, not just answers

The model outputs a probability such as 0.83. We then choose a **threshold** (often 0.5) to turn it into Yes or No. Changing the threshold trades missed cases against false alarms.

Where it is used

Loan approval, lead scoring, churn prediction and email filtering are all classification problems you can build for clients.

From line to probability

Logistic regression computes a weighted sum, like linear regression, and then passes it through the sigmoid function to get a number between 0 and 1. That number is read as a probability.

Choosing the threshold for the business

For fraud, missing a real fraud is costly, so you may flag at 0.3. For an expensive marketing call, you may only call leads above 0.8. The threshold is a business decision, not a technical one. Show the client a table of thresholds with the number of cases caught and false alarms.

Scaling helps

Logistic regression trains better when inputs are on similar scales. Put a StandardScaler in a pipeline, as in the full code.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
X, y = load_breast_cancer(return_X_y=True)
clf = LogisticRegression(max_iter=5000).fit(X, y)
```

```
print("Probability of class 1:", clf.predict_proba(X[:3])[:,
1].round(3))
```

Full Code: Threshold Table for the Client

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
X, y = load_breast_cancer(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
random_state=0, stratify=y)
clf = make_pipeline(StandardScaler(),
LogisticRegression(max_iter=1000)).fit(Xtr, ytr)
proba = clf.predict_proba(Xte)[:, 1]
print("threshold  predicted_yes  correct_yes  missed_yes")
for t in [0.2, 0.4, 0.5, 0.6, 0.8]:
    pred = proba >= t
    print(f"{t:9.1f}  {pred.sum():13d}  {(pred & (yte == 1)).sum():
11d}  {((~pred) & (yte == 1)).sum():10d}")
```

How the Full Code Works

1. stratify=y keeps the same share of each class in the training and test sets.
2. make_pipeline chains a StandardScaler and the logistic regression, so scaling is always applied the same way.
3. predict_proba(...)[:, 1] gives the probability of class 1 for each test case.
4. For each threshold we count how many cases are predicted Yes, how many of those are truly Yes, and how many real Yes cases were missed.

5. This table lets a non-technical manager choose the threshold based on cost, not on a default number.

Real-World Case: Loan follow-ups for a small finance company

A small lender wanted to know which borrowers might miss next month's payment. A logistic regression on payment history, income band and loan size produced a risk probability for each borrower. Calling everyone was too expensive, so the manager used the threshold table: at 0.6 the team made 120 calls a month and reached most of the borrowers who later struggled. The threshold, not the algorithm, made the project profitable.

Common Mistakes

- Always using 0.5 without thinking about costs.
- Reading predict_proba output as perfectly calibrated probabilities.
- Forgetting to scale inputs, leading to slow or failed training.

Exercise

Use predict_proba with a threshold of 0.8 instead of 0.5. How many more cases become 'No'?

> **Answer:** Raising the threshold to 0.8 turns borderline cases into 'No', so fewer Yes predictions are made: fewer false alarms, but more missed cases.

Mini Project

Build a 'will this student pass?' model from attendance and test scores.

Key Terms

Classification
Predicting a category.

Sigmoid
Function mapping any number into 0–1.

Threshold
Cut-off that turns a probability into Yes/No.

Pipeline
Chained preprocessing and model steps.

> **Go further:** The ready-to-use production version of this project*, Lead Scoring Model for Sales Teams*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 6

Train, Test, Validate: How to Know Your Model Really Works

Learn the train/test split, accuracy, precision, recall and the confusion matrix, explained simply.

In this chapter you will learn:

- Why we hide test data
- Accuracy vs. precision vs. recall
- Reading a confusion matrix

Never test on training data

A model that has seen the answers will look perfect. We keep a **test set** hidden during training, just like an exam paper the student has not seen.

The right metric

Accuracy can lie. If 99% of transactions are safe, a model that always says 'safe' is 99% accurate and useless. **Precision** asks: of those we flagged, how many were right? **Recall** asks: of the real cases, how many did we catch?

Validation set

For tuning settings, use a third split (validation) or cross-validation, so the test set stays truly unseen until the end.

The confusion matrix

A 2×2 table: true positives, false positives, true negatives and false negatives. Every other metric is calculated from these four numbers. Learn to read it and you can explain any classifier.

F1 and averages

F1 combines precision and recall into one score. With many classes, 'macro' averaging treats every class equally, while 'weighted' averaging follows class sizes. For rare but important classes, look at macro scores.

Cross-validation

Instead of one split, cross-validation trains and tests 5 times on different folds and averages the result. It gives a more reliable estimate, especially with small datasets.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
X, y = load_breast_cancer(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
clf = LogisticRegression(max_iter=5000).fit(Xtr, ytr)
print(classification_report(yte, clf.predict(Xte)))
```

Full Code: Confusion Matrix and Cross-Validation

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
X, y = load_breast_cancer(return_X_y=True)
model = make_pipeline(StandardScaler(),
LogisticRegression(max_iter=1000))
scores = cross_val_score(model, X, y, cv=5, scoring="recall")
print("5-fold recall:", scores.round(3), "mean",
scores.mean().round(3))
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
tn, fp, fn, tp = confusion_matrix(yte, model.fit(Xtr,
ytr).predict(Xte)).ravel()
print(f"TP={tp} FP={fp} TN={tn} FN={fn}")
print("precision", round(tp / (tp + fp), 3), "recall", round(tp / (tp +
fn), 3))
```

How the Full Code Works

1. cross_val_score runs 5-fold cross-validation and reports recall for each fold, giving a sense of how stable the model is.
2. A separate split is then used to train once and build a confusion matrix.
3. confusion_matrix(...).ravel() unpacks the four numbers: true negatives, false positives, false negatives and true positives.

4. Precision and recall are calculated by hand from those four numbers, so you can see exactly where they come from.

Real-World Case: Spotting defective products

A factory used a camera-based model to flag defective parts. In testing it showed 98% accuracy, but only 2% of parts were defective, so 'always say fine' would also score 98%. The confusion matrix revealed the model was missing half of the real defects (recall 50%). After rebalancing the training data and lowering the threshold, recall rose to 90% with a manageable number of false alarms. Without the right metric, the original model would have shipped and failed.

Common Mistakes

- Tuning settings again and again on the test set until it looks good.
- Using accuracy on imbalanced data.
- Not using stratify, so the test set has a different class balance.

Exercise

Which matters more for cancer screening, precision or recall? Explain in one line.

> **Answer:** Recall matters more for cancer screening: missing a real case is far worse than a false alarm that a doctor can re-check.

Mini Project

Write a one-page evaluation summary of one model for a non-technical manager.

Key Terms

Test set
Data kept hidden until final evaluation.

Precision
Of predicted positives, the share that are correct.

Recall
Of real positives, the share that were found.

Cross-validation
Repeated train/test on different folds.

> **Go further:** The ready-to-use production version of this project, *Model Evaluation Report Generator*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 7

Decision Trees and Random Forests: Models You Can Explain

Use tree-based models that are powerful, fast and easy to explain to non-technical people.

In this chapter you will learn:

- How a decision tree asks questions
- Why a forest beats one tree
- Feature importance

A tree of questions

A decision tree splits data by asking simple questions: 'Is age over 30?', 'Is income over 5 lakh?'. You can draw it on paper, which makes it easy to explain.

Many trees vote

A **random forest** trains hundreds of trees on random parts of the data and lets them vote. This reduces mistakes and is one of the strongest models for table data.

Which inputs matter

Forests tell you **feature importance**: which columns influenced decisions most. Business owners love this view.

How a split is chosen

At each step the tree tries possible questions and picks the one that best separates the classes, measured by Gini impurity or entropy. It keeps splitting until a stopping rule, such as maximum depth, is reached.

Why randomness helps

Each tree in a forest sees a random sample of rows and a random subset of columns at each split. The trees make different mistakes, and voting cancels many of them out.

Gradient boosting

Boosting builds trees one after another, each fixing the previous errors. Libraries like XGBoost, LightGBM and scikit-learn's HistGradientBoosting often win competitions on table data. Try them after a random forest.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
X, y = load_wine(return_X_y=True)
rf = RandomForestClassifier(n_estimators=300, random_state=0)
print("CV accuracy:", cross_val_score(rf, X, y, cv=5).mean().round(3))
rf.fit(X, y); print("Top feature index:",
rf.feature_importances_.argmax())
```

Full Code: Forest vs. Boosting with Feature Importance

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier,
HistGradientBoostingClassifier
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier, export_text
data = load_wine(as_frame=True); X, y = data.data, data.target
for name, m in [("tree", DecisionTreeClassifier(max_depth=3,
random_state=0)),
                ("forest", RandomForestClassifier(n_estimators=300,
random_state=0)),
                ("boosting",
HistGradientBoostingClassifier(random_state=0))]:
    print(f"{name:9s}", cross_val_score(m, X, y, cv=5).mean().round(3))
tree = DecisionTreeClassifier(max_depth=2, random_state=0).fit(X, y)
print(export_text(tree, feature_names=list(X.columns)))
rf = RandomForestClassifier(n_estimators=300, random_state=0).fit(X, y)
top = sorted(zip(rf.feature_importances_, X.columns), reverse=True)[:5]
for imp, col in top: print(f"{col:30s} {imp:.3f}")
```

How the Full Code Works

1. Three models are compared with 5-fold cross-validation: a single small tree, a random forest and gradient boosting.
2. export_text prints a depth-2 tree as readable if/else rules, which you can show to anyone.
3. The forest is trained on all data and its feature_importances_ are sorted to show the top five inputs.

4. Compare the scores: the single tree is easiest to explain, while the forest and boosting are usually more accurate.

Real-World Case: Which customers will reorder?

A wholesale supplier wanted to predict which shop owners would reorder within 30 days. A random forest using order frequency, last order size, days since last order and payment delays reached good accuracy. The feature importance chart showed 'days since last order' mattered most, so the sales team started calling shops on day 20. A depth-2 tree printed with export_text became a one-page rule sheet for staff without any AI tool.

Common Mistakes

- Letting a single tree grow without limits, which memorises the data.
- Treating feature importance as proof of cause.
- Ignoring class imbalance; use class_weight='balanced' when needed.

Exercise

Try n_estimators=10 and 1000. How much does accuracy change? How about training time?

> **Answer:** With 10 trees accuracy is usually slightly lower and less stable; 1000 trees is only a little better than 300 but much slower. Around 200–500 is a good balance.

Mini Project

Print the depth-2 tree for a dataset of your own and explain it to a friend.

Key Terms

Decision tree
A model of yes/no questions.

Random forest
Many trees voting.

Boosting
Trees built in sequence to fix errors.

Feature importance
How much each input influenced decisions.

> **Go further:** The ready-to-use production version of this project, *Explainable AI Dashboard*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 8

Overfitting vs. Underfitting: The Most Important Idea in ML

Why some models memorise instead of learning, and the simple tricks that fix it.

In this chapter you will learn:

- Spotting overfitting
- Bias vs. variance in plain words
- Regularisation and more data

Memorising vs. understanding

An **overfit** model scores very high on training data and badly on new data, like a student who memorised answers. An **underfit** model is too simple and scores badly everywhere.

How to spot it

Compare the training score with the validation score. A big gap means overfitting. Both being low means underfitting.

How to fix it

For overfitting: get more data, simplify the model, add regularisation, or stop training earlier. For underfitting: use a bigger model or better features.

The learning curve

Plot training and validation scores as you add more data. If the two curves are far apart, more data will help. If both are low and flat, you need a better model or features.

Regularisation

Regularisation adds a penalty for complexity. In linear models, L2 (Ridge) shrinks weights and L1 (Lasso) can set some to zero. In neural networks, dropout and weight decay do the same job.

Early stopping

For models trained in steps, stop when the validation score stops improving. You will use this constantly in deep learning.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
X, y = load_breast_cancer(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, random_state=1)
for depth in [1, 3, 10, None]:
    m = DecisionTreeClassifier(max_depth=depth,
random_state=0).fit(Xtr, ytr)
    print(depth, "train", round(m.score(Xtr, ytr), 3), "test",
round(m.score(Xte, yte), 3))
```

Full Code: Learning Curve in Numbers

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import learning_curve
from sklearn.tree import DecisionTreeClassifier
X, y = load_breast_cancer(return_X_y=True)
for depth in [2, None]:
    sizes, tr, va =
learning_curve(DecisionTreeClassifier(max_depth=depth, random_state=0),
X, y,
                                    cv=5, train_sizes=np.linspace(0.1,
1.0, 5))
    print(f"\nmax_depth={depth}")
    for s, a, b in zip(sizes, tr.mean(1), va.mean(1)):
        print(f"  {s:4d} rows  train {a:.3f}  valid {b:.3f}  gap {a -
b:.3f}")
```

How the Full Code Works

1. learning_curve trains the model on growing portions of the data (10% up to 100%) with 5-fold cross-validation.
2. For each size we print the average training score, validation score and the gap between them.
3. With max_depth=2, both scores are close: the model is simple and stable.
4. With max_depth=None, training stays near 1.0 while validation is lower: a clear sign of overfitting.

Real-World Case: Too good to be true

A student built a model to predict exam results and proudly reported 100% accuracy. The learning curve showed training at 100% and validation at 71%. The tree had memorised individual students. Limiting the depth and adding two years of extra data brought training to 86% and validation to 83%. That is a lower headline number, but a model the school could actually trust on new students.

Common Mistakes

- Celebrating 100% training accuracy.
- Testing many settings on one small validation split and overfitting to it.
- Adding complexity before collecting more or cleaner data.

Exercise

Which max_depth gives the best test score? Which one overfits most?

> **Answer:** A depth around 3–5 usually gives the best test score. max_depth=None overfits most: training accuracy 1.0 with a lower test score.

Mini Project

Draw the learning curve of one of your models and decide: more data, or a better model?

Key Terms

Overfitting
Memorising training data, failing on new data.

Underfitting
Too simple to capture patterns.

Regularisation
A penalty that discourages complexity.

Early stopping
Stopping when validation stops improving.

> **Go further:** The ready-to-use production version of this project, *Hyperparameter Tuning Masterclass*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 9

Feature Engineering: Making Your Data Smarter

Create new columns that help models learn faster and predict better, with real-world examples.

In this chapter you will learn:

- Encoding categories
- Scaling numbers
- Building pipelines

Better inputs, better model

Good features often beat fancy models. Turning a date into 'day of week' or 'is festival season' can improve a sales forecast more than switching algorithms.

Common techniques

One-hot encoding turns categories such as city names into 0/1 columns. **Scaling** puts numbers on a similar range. **Binning** groups ages into ranges.

Use pipelines

A scikit-learn Pipeline applies the same steps in training and in production, which prevents a very common bug.

Date and time features

From one date column you can create day of week, month, weekend flag, days to the next festival, and season. Retail and food businesses in India show strong Diwali, wedding-season and month-end patterns.

Ratios and interactions

Price per square foot, debt-to-income, and clicks per visit often carry more signal than the raw numbers. Domain knowledge from the client is the best source of such ideas.

Avoiding leakage

Never create a feature using information that would not exist at prediction time, such as 'total refunds this month' when predicting a purchase today. Leakage makes a model look excellent in testing and fail in real use.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
df = pd.DataFrame({"city": ["Mumbai","Varanasi","Mumbai","Delhi"],
"income": [9, 4, 12, 7], "buy": [1, 0, 1, 0]})
prep = ColumnTransformer([("cat",
OneHotEncoder(handle_unknown="ignore"), ["city"]), ("num",
StandardScaler(), ["income"])])
model = make_pipeline(prep,
LogisticRegression()).fit(df[["city","income"]], df["buy"])
```

```
print(model.predict(pd.DataFrame({"city": ["Lucknow"], "income":
[10]})))
```

Full Code: Date Features + Pipeline for a Shop

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import RandomForestRegressor
df = pd.DataFrame({"date": pd.date_range("2026-01-01", periods=120,
freq="D"),
                   "store": ["Mumbai", "Varanasi", "Delhi"] * 40})
df["sales"] = 100 + df["date"].dt.dayofweek * 8 + (df["store"] ==
"Mumbai") * 30
def add_features(d):
    d = d.copy()
    d["dow"] = d["date"].dt.dayofweek
    d["month"] = d["date"].dt.month
    d["is_weekend"] = (d["dow"] >= 5).astype(int)
    d["month_end"] = (d["date"].dt.days_in_month - d["date"].dt.day <
3).astype(int)
    return d
X = add_features(df)[["store", "dow", "month", "is_weekend",
"month_end"]]
prep = ColumnTransformer([("cat",
OneHotEncoder(handle_unknown="ignore"), ["store"])],
remainder=StandardScaler())
model = make_pipeline(prep,
RandomForestRegressor(random_state=0)).fit(X, df["sales"])
new = add_features(pd.DataFrame({"date":
pd.to_datetime(["2026-06-06"]), "store": ["Mumbai"]}))
print("Predicted sales:", model.predict(new[X.columns]).round(1))
```

How the Full Code Works

1. We create 120 days of sample sales for three stores, with weekday and city effects built in.
2. add_features turns one date column into day of week, month, a weekend flag and a month-end flag. The same function is used for training and prediction.
3. ColumnTransformer one-hot encodes the store name and scales the other numeric features (remainder=StandardScaler()).
4. The pipeline joins preprocessing and the random forest, so new data goes through exactly the same steps.
5. To predict, we build a new row with a date and store, add the same features and call predict.

Real-World Case: Festival demand for a sweet shop

A sweet shop chain's sales model was poor until the owner suggested two features: 'days until Diwali' and 'wedding-season month'. Adding them cut the forecast error by about a third. The model did not change; the inputs did. The owner's knowledge, turned into features, was worth more than any algorithm upgrade.

Common Mistakes

- Creating features from future information (leakage).
- One-hot encoding a column with thousands of unique values without grouping rare ones.

- Doing preprocessing outside the pipeline, so production differs from training.

Exercise

Add a column 'age' and include it in the scaler. Does the pipeline still work?

> **Answer:** Yes. Add 'age' to the DataFrame and to the scaler's column list (or leave it to remainder=StandardScaler()). The pipeline handles it automatically.

Mini Project

List 10 feature ideas for a business you know well, then ask the owner which ones they believe matter.

Key Terms

Feature
An input column.

One-hot encoding
Turning categories into 0/1 columns.

Scaling
Putting numbers on similar ranges.

Data leakage
Using information not available at prediction time.

> **Go further:** The ready-to-use production version of this project, *Feature Engineering Recipe Book*, is available at https://sachintekriwal.com/marketplace.

CHAPTER 10

Save, Load and Serve Your Model as an API

Turn your trained model into a web API with FastAPI so websites and apps can use it.

In this chapter you will learn:

- Saving models with joblib
- Building a FastAPI endpoint
- Calling it from a web page

From notebook to product

A model in a notebook helps nobody. Save it to a file, load it in a small server, and expose a URL that apps can call. This is the first step to selling AI as a service.

FastAPI

FastAPI is a fast, simple Python web framework. It also creates automatic API docs at /docs, which is great for showing clients.

Same steps everywhere

Save the whole pipeline (preprocessing plus model) so the API gets exactly the same transformations as training did.

Input validation with Pydantic

FastAPI uses Pydantic models to check incoming data. If a field is missing or has the wrong type, the API answers with a clear error instead of crashing. This protects your model and your users.

Versioning models

Save models with a version in the name, such as model_v3.joblib, and return the version in each API response. When a client reports a strange result, you will know which model produced it.

Calling the API from a web page

Any website can call your API with fetch(). Enable CORS for the domains you trust. This is how your models will power PWAs and client sites.

Starter Code

Type this in yourself rather than copying. Typing builds understanding.

```
# train_and_save.py
import joblib
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
X, y = load_iris(return_X_y=True)
joblib.dump(RandomForestClassifier().fit(X, y), "model.joblib")

# api.py  ->  run: uvicorn api:app --reload
from fastapi import FastAPI
import joblib
app = FastAPI(); model = joblib.load("model.joblib")
@app.get("/predict")
```

```
def predict(a: float, b: float, c: float, d: float):
    return {"class": int(model.predict([[a, b, c, d]])[0])}
```

Full Code: Production-Style Prediction API

This is the complete version, which goes further than the starter. Run it, then change one thing at a time and watch what happens.

```
# api.py  -  run: uvicorn api:app --reload   (train first with
train_and_save.py from the free lesson)
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field
import joblib
MODEL_VERSION = "iris-rf-v1"
model = joblib.load("model.joblib")
NAMES = ["setosa", "versicolor", "virginica"]
app = FastAPI(title="NKB Prediction API")
app.add_middleware(CORSMiddleware, allow_origins=["https://
sachintekriwal.com"], allow_methods=["*"], allow_headers=["*"])
class Flower(BaseModel):
    sepal_length: float = Field(gt=0, lt=20)
    sepal_width: float = Field(gt=0, lt=20)
    petal_length: float = Field(gt=0, lt=20)
    petal_width: float = Field(gt=0, lt=20)
@app.post("/predict")
def predict(f: Flower):
    x = [[f.sepal_length, f.sepal_width, f.petal_length,
f.petal_width]]
    probs = model.predict_proba(x)[0]
    best = int(probs.argmax())
    return {"label": NAMES[best], "confidence":
round(float(probs[best]), 3), "model_version": MODEL_VERSION}
@app.get("/health")
def health():
    return {"ok": True, "model_version": MODEL_VERSION}
```

How the Full Code Works

1. MODEL_VERSION is returned with every answer, so you can trace any prediction to the exact model.

2. The model is loaded once when the server starts, not on every request, so each call is fast.
3. CORSMiddleware allows only your own website to call the API from a browser.
4. The Pydantic class Flower defines the expected fields and their valid ranges. Bad input gets a clear error automatically.
5. /predict returns the label, the confidence and the model version. /health lets monitoring tools check that the service is alive.

Real-World Case: From notebook to product

A developer had a working crop-price model in a notebook but no way for farmers' co-operatives to use it. Wrapping it in a FastAPI service like this one took one afternoon. A simple Hindi web form called the API, and within a week three co-operatives were checking expected prices before going to the mandi. The same API later powered a WhatsApp bot. The model was the same; the API made it a product.

Common Mistakes

- Loading the model inside the request function, which makes every call slow.
- Allowing CORS from every website in production.
- Returning raw errors that reveal server paths.

Exercise

Open http://127.0.0.1:8000/docs and test the endpoint from the browser.

> **Answer:** At /docs you can click 'Try it out', enter the four numbers and see the predicted class instantly.

Mini Project

Wrap your Chapter 4 or 5 model in this API and call it from a simple HTML form.

Key Terms

API
A URL other programs call to use your model.

FastAPI
Python framework for building APIs.

Pydantic
Library that validates input data.

CORS
Browser rule controlling which sites may call an API.

> **Go further:** The ready-to-use production version of this project*, AI Model API Boilerplate (SaaS-ready),* is available at https://sachintekriwal.com/marketplace.

What You Have Built

Look back at what you can now do. You can explain what a model is and how it learns. You can set up an AI workspace, clean data, and build regression and classification models. You can evaluate them honestly with test sets, precision, recall and cross-validation. You can use trees, forests and boosting, recognise overfitting, and engineer better features. Finally, you can turn a trained model into an API that real applications can call.

These are the foundations used every day by professional data scientists. Everything in the rest of the series, from neural networks to video generation, builds on them.

Your 30-day practice plan

- **Week 1:** Repeat Chapters 3–6 on a dataset from your own work or city.
- **Week 2:** Try random forest and boosting on the same data and compare them honestly.
- **Week 3:** Add three new features from your own knowledge of the problem and measure the change.
- **Week 4:** Deploy your best model as an API and connect a simple web form to it.

Next: Book 2

Neural Networks with PyTorch takes you from single neurons to image classifiers, transfer learning, sequence models and embeddings, the building blocks of every modern AI system.

Appendix: Python Essentials for AI in 30 Minutes

If you are new to Python, read this appendix before Chapter 3. It covers only what you need for this book. Type each example and run it.

1. Variables and types

```
name = "Sachin"        # text (str)
age = 32               # whole number (int)
price = 499.50         # decimal number (float)
is_student = False     # True or False (bool)
print(name, age, price, is_student)
print(type(price))
```

2. Lists and dictionaries

A **list** holds items in order. A **dictionary** holds key-value pairs, like a phone book.

```
marks = [72, 85, 64, 90]
print(marks[0], marks[-1], len(marks))   # first, last, count
marks.append(78)
student = {"name": "Asha", "city": "Varanasi", "marks": 85}
print(student["city"])
student["passed"] = True
```

3. Conditions and loops

```
for m in marks:
    if m >= 80:
        print(m, "distinction")
    elif m >= 60:
        print(m, "pass")
    else:
        print(m, "needs help")

total = 0
```

```
for i in range(1, 11):     # 1 to 10
    total += i
print("sum:", total)
```

4. Functions

A function packs code you will reuse. Most of the full code in this book is built from small functions.

```
def grade(mark, pass_mark=60):
    return "pass" if mark >= pass_mark else "fail"

print(grade(72), grade(55), grade(55, pass_mark=50))
```

5. List comprehensions

```
squares = [x * x for x in range(5)]           # [0, 1, 4, 9, 16]
toppers = [m for m in marks if m >= 80]
print(squares, toppers)
```

6. Importing libraries

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
print(np.mean([72, 85, 64]))
```

`import x as y` gives a library a short name. `from x import y` brings in one item directly.

7. NumPy in one minute

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a * 10)          # maths on every item at once: [10 20 30 40]
print(a.mean(), a.max(), a.shape)
m = np.array([[1, 2], [3, 4]])
print(m.T)             # transpose: swap rows and columns
```

8. Pandas in one minute

```
import pandas as pd
df = pd.DataFrame({"city": ["Mumbai", "Varanasi", "Mumbai"], "sales":
[120, 80, 150]})
print(df.head())
print(df[df["sales"] > 100])               # filter rows
print(df.groupby("city")["sales"].sum())   # total by city
df["sales_k"] = df["sales"] / 1000         # new column
```

9. Reading errors

Errors are normal. Read the **last line** of the error first: it names the problem, such as `KeyError: 'Sales'` (wrong column name; Python is case-sensitive) or `ModuleNotFoundError` (library not installed; use pip install). The lines above it show where the problem happened.

10. Good habits

- Use clear names: `monthly_sales`, not `x1`.
- Run code often in small pieces.
- Write a short comment explaining *why*, not *what*.
- Keep data, notebooks and models in separate folders (Chapter 2).

Glossary

API
A URL other programs call to use your model.

Baseline
A very simple model to beat.

Boosting
Trees built in sequence to fix errors.

Class balance
How evenly the target labels are spread.

Classification
Predicting a category.

CORS
Browser rule controlling which sites may call an API.

Cross-validation
Repeated train/test on different folds.

CUDA
NVIDIA's platform that lets software use the GPU.

Data leakage
Using information not available at prediction time.

DataFrame
A table of data in Pandas.

Decision tree
A model of yes/no questions.

Early stopping
Stopping when validation stops improving.

FastAPI

Python framework for building APIs.

Feature
An input column.

Feature importance
How much each input influenced decisions.

Gradient descent
Repeatedly nudging weights in the direction that lowers the loss.

Jupyter notebook
An interactive document mixing code, output and notes.

Loss
A score of how wrong the model is.

MAE
Average absolute error.

Median
The middle value; robust to outliers.

Missing value (NaN)
An empty cell.

Model
A function with learnable numbers that maps inputs to outputs.

One-hot encoding
Turning categories into 0/1 columns.

Overfitting
Memorising training data, failing on new data.

Pipeline
Chained preprocessing and model steps.

Precision
Of predicted positives, the share that are correct.

Pydantic
Library that validates input data.

Random forest
Many trees voting.

Recall
Of real positives, the share that were found.

Regression
Predicting a number.

Regularisation
A penalty that discourages complexity.

requirements.txt
A list of libraries needed to rebuild an environment.

R^2
Share of variation the model explains; 1.0 is perfect.

Scaling
Putting numbers on similar ranges.

Sigmoid
Function mapping any number into 0–1.

Test set
Data kept hidden until final evaluation.

Threshold
Cut-off that turns a probability into Yes/No.

Underfitting
Too simple to capture patterns.

Virtual environment

A private set of libraries for one project.

Weight
A learned number inside a model.

About the Author

Sachin Tekriwal is an AI teacher, AI engineer and automation expert with 32 years of experience in software engineering. He is the founder of NKBWorks, a web development and digital services company, and has led products for companies in India, the UK and the UAE.

He teaches AI to students, professionals and business owners, and builds AI products used by real businesses. His mission is to make AI practical, affordable and understandable for everyone.

Website: https://sachintekriwal.com
Free lessons: https://sachintekriwal.com/blog
Ready-made AI projects: https://sachintekriwal.com/marketplace
Contact: sachintekriwalai@gmail.com

If this book helped you, please leave a review on Amazon. It helps other learners find it and helps the author write better books.