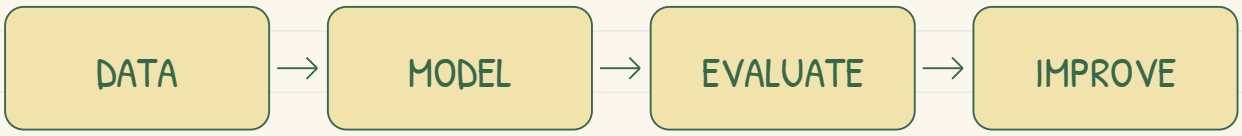
THE PRACTICAL NOTEBOOK

Python for AI & ML

Understand the data. Train carefully. Evaluate honestly.

64 pages / 20 guided lessons / 40 code files



Understand it. Try it. Explain it.

Learn machine-learning ideas through small, inspectable experiments. Begin with data and baselines, then explore regression, classification, clustering and text retrieval. The final lessons connect these foundations to AI applications and evaluation. Educational algorithms reveal the mechanics; established libraries are usually preferable for larger real projects.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 20-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	Define the problem first	4
02	Rows, features and targets	7
03	Arrays and shapes	10
04	Inspect before training	13
05	Train, validation and test	16
06	Baselines before models	19
07	Missing-value strategies	22
08	Scaling and feature units	25
09	Linear regression	28
10	Loss and gradient descent	31
11	Nearest-neighbour classification	34
12	Probabilities and thresholds	37
13	Metrics that match mistakes	40
14	Overfitting and model choice	43
15	Cross-validation and leakage	46
16	Clustering without labels	49
17	Text features and similarity	52
18	Retrieval before generation	55
19	Model cards and monitoring	58
20	Project: an honest experiment	61

Setup and study method · 3 References and next steps · 64

Make this book yours

BEFORE YOU START

Know Python functions, lists and basic arithmetic. The book introduces arrays and the required mathematical ideas gradually. Use fictional or public sample data. These small exercises are teaching experiments, not validated models for consequential decisions.

YOUR WORKSPACE

Use Python 3.11 or newer and a virtual environment. Install NumPy for numerical examples. Most examples also work with the standard library alone. Every dataset is included in the code, so no dataset download or paid AI API is needed. Save each example separately and run it before changing one assumption.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python -m pip install numpy
python lesson_01_example.py
python lesson_01_solution.py
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_20_example.py and matching solution files. Run them individually from the extracted folder. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

Define the problem first

THE GOAL

Choose a target and a useful success criterion.

Artificial intelligence is a broad field; machine learning learns patterns from examples. Supervised learning uses labelled targets, while unsupervised methods look for structure without a supplied target. Before training, define who will use the result, what action it supports and what a mistake costs.

lesson_01_example.py

```
problem = {
    "input": "support message text",
    "target": "billing or technical",
    "action": "suggest a queue for human review",
    "metric": "macro F1 plus per-class error review",
}
assert problem["target"]
assert "human" in problem["action"]
```

The design note limits the model to a suggestion rather than automatically resolving an account. It also makes the evaluation question concrete.

WATCH FOR THIS

Choosing a model before defining the problem often leads to impressive-looking numbers that answer the wrong question.

Your turn: Define the problem first

THE CHALLENGE

Classify three proposed tasks as regression, classification or clustering. Explain what the target is, if one exists.

ACCEPTANCE CHECKS

[ ] Predict delivery minutes -> regression

[ ] Label an email as spam/not spam -> classification

[ ] Group similar products without labels -> clustering

A NUDGE, IF YOU NEED ONE

Ask whether the desired output is a number, a supplied category or a discovered grouping.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DEFINE THE PROBLEM FIRST

lesson_01_solution.py

```
tasks = {
    "delivery_minutes": "regression",
    "spam_label": "classification",
    "product_groups": "clustering",
}
assert tasks["delivery_minutes"] == "regression"
assert tasks["spam_label"] == "classification"
assert tasks["product_groups"] == "clustering"
```

WHY IT WORKS

A task description is not enough to establish feasibility. You still need representative data, a meaningful label and a reason that the prediction helps.

TAKE IT FURTHER

Write a one-paragraph project brief with the user, target, baseline, metric and a case where the model must defer.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Rows, features and targets

THE GOAL

Keep inputs separate from what you want to predict.

A feature is an input available at prediction time. A target is the value you want to estimate. A table usually has one example per row, but define what a row means: one customer, one transaction or one day. Mixing different units of observation can make both training and evaluation misleading.

lesson_02_example.py

```
rows = [
    {"hours": 1, "practice": 2, "score": 45},
    {"hours": 3, "practice": 5, "score": 70},
    {"hours": 5, "practice": 8, "score": 90},
]
X = [[r["hours"], r["practice"]] for r in rows]
y = [r["score"] for r in rows]
assert len(X) == len(y) == 3
assert X[0] == [1, 2]
```

The score is excluded from X. Including it would reveal the answer during training and make evaluation meaningless.

WATCH FOR THIS

An identifier can accidentally encode the target or ordering. Do not assume every numeric column is a useful measurement.

Your turn: Rows, features and targets

THE CHALLENGE

Prepare features for predicting whether a delivery is late. Exclude fields available only after the delivery has finished.

ACCEPTANCE CHECKS

[ ] Distance and dispatch hour are eligible inputs

[ ] Actual arrival time is excluded

[ ] late is the target, not a feature

A NUDGE, IF YOU NEED ONE

Use an explicit feature allowlist rather than every column except a few.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ROWS, FEATURES AND TARGETS

lesson_02_solution.py

```
row = {"distance_km": 12, "dispatch_hour": 9,
       "arrival_hour": 11, "late": True}
feature_names = ["distance_km", "dispatch_hour"]
features = [row[name] for name in feature_names]
target = row["late"]
assert features == [12, 9]
assert target is True
assert "arrival_hour" not in feature_names
```

WHY IT WORKS

Feature availability must match the moment the model will be used. A column can be legitimate for reporting and still be invalid for prediction.

TAKE IT FURTHER

Write a data dictionary defining each feature, unit, missing-value rule and availability time.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Arrays and shapes

THE GOAL

Use dimensions to understand numerical operations.

NumPy arrays hold numeric data in structured shapes. A matrix with shape (rows, features) is different from a single feature vector. Elementwise operations and matrix multiplication have different meanings. Inspect shapes before debugging formulas; many mistakes are mismatched dimensions or unintended broadcasting.

lesson_03_example.py

```
import numpy as np
X = np.array([[1.0, 2.0], [3.0, 4.0]])
w = np.array([2.0, 1.0])
predictions = X @ w
assert X.shape == (2, 2)
assert predictions.shape == (2,)
assert np.allclose(predictions, [4.0, 10.0])
```

Each row is multiplied by the weight vector and summed. The first prediction is 1*2 + 2*1 = 4. The @ operator performs matrix multiplication.

WATCH FOR THIS

A successful array operation does not prove that feature order matches the order used during training.

Your turn: Arrays and shapes

THE CHALLENGE

Add an intercept of 3 to each prediction and verify the result. Then reject an input matrix with the wrong number of features.

ACCEPTANCE CHECKS

[ ] [[1,2],[3,4]], weights [2,1], bias 3 -> [7,13]

[ ] Wrong feature count -> ValueError

[ ] Output has one value per row

A NUDGE, IF YOU NEED ONE

Check X.shape[1] against the weight length before multiplying.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ARRAYS AND SHAPES

lesson_03_solution.py

```
import numpy as np

def predict(X, weights, bias):
    X = np.asarray(X, dtype=float)
    weights = np.asarray(weights, dtype=float)
    if X.ndim != 2 or X.shape[1] != len(weights):
        raise ValueError("Wrong feature shape")
    return X @ weights + bias
assert np.allclose(predict([[1, 2], [3, 4]], [2, 1], 3), [7, 13])
try:
    predict([[1, 2, 3]], [2, 1], 3)
except ValueError:
    pass
else:
    raise AssertionError("Shape mismatch accepted")
```

WHY IT WORKS

Explicit shape checks make failures easier to interpret. They also prevent some valid-but-unintended broadcasting operations.

TAKE IT FURTHER

Carry feature names beside the array and validate their order when loading a model.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Inspect before training

THE GOAL

Look for missing values, duplicates and impossible measurements.

Data quality problems can dominate model behaviour. Count rows, inspect ranges and review examples before choosing an algorithm. A missing value is not always zero. Duplicate rows may be legitimate repeated events or accidental copies; decide using the meaning of the data.

lesson_04_example.py

```
rows = [{"age": 20}, {"age": None}, {"age": -2}, {"age": 25}]
missing = sum(r["age"] is None for r in rows)
invalid = [r for r in rows if r["age"] is not None and r["age"] < 0]
assert missing == 1
assert invalid == [{"age": -2}]
```

The checks separate missing from impossible values. Automatically deleting every unusual value could remove valid rare cases.

WATCH FOR THIS

A clean schema does not prove labels are accurate or the sample represents future users.

Your turn: Inspect before training

THE CHALLENGE

Create a quality report for a numeric feature with None values. Report total, missing, valid count, minimum and maximum. Define empty valid data clearly.

ACCEPTANCE CHECKS

[ ] [1,None,3] -> total 3, missing 1, min 1, max 3

[ ] [None] -> min and max None

[ ] [] -> total 0

A NUDGE, IF YOU NEED ONE

Build the valid list first, then guard min and max on emptiness.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

INSPECT BEFORE TRAINING

lesson_04_solution.py

```
def quality(values):
    valid = [v for v in values if v is not None]
    return {"total": len(values), "missing": len(values)-len(valid),
            "valid": len(valid), "min": min(valid) if valid else None,
            "max": max(valid) if valid else None}
assert quality([1, None, 3])["missing"] == 1
assert quality([None])["min"] is None
assert quality([])["total"] == 0
```

WHY IT WORKS

This narrow helper assumes every non-missing value is numeric. Extend it deliberately for strings, infinities and invalid units rather than silently coercing everything.

TAKE IT FURTHER

Review five random examples and five extreme examples manually, recording questions about collection and labelling.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Train, validation and test

THE GOAL

Keep an honest measure of unseen performance.

Training data fits the model. Validation data helps choose settings. A final test set estimates performance after choices are finished. Split according to the real deployment: time-ordered problems need future-held-out data, while repeated records from one person often need group separation.

lesson_05_example.py

```
rows = list(range(10))
train = rows[:6]
validation = rows[6:8]
test = rows[8:]
assert set(train).isdisjoint(validation)
assert set(train).isdisjoint(test)
assert max(train) < min(test)
```

This chronological split is useful when indexes represent time. It is not a universal replacement for random or grouped splitting.

WATCH FOR THIS

Randomly splitting nearly identical records can make test performance look much better than real-world performance.

Your turn: Train, validation and test

THE CHALLENGE

Split records by customer so the same customer never appears in both training and test data.

ACCEPTANCE CHECKS

[ ] All records for customer C go to test

[ ] Customers A and B stay in training

[ ] No customer overlap

A NUDGE, IF YOU NEED ONE

Choose held-out customer identifiers, then filter entire groups.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TRAIN, VALIDATION AND TEST

lesson_05_solution.py

```
rows = [("A", 1), ("A", 2), ("B", 3), ("C", 4), ("C", 5)]
held_out = {"C"}
train = [r for r in rows if r[0] not in held_out]
test = [r for r in rows if r[0] in held_out]
assert {r[0] for r in train}.isdisjoint(r[0] for r in test)
assert len(train) == 3 and len(test) == 2
```

WHY IT WORKS

The split tests generalisation to unseen customers. If deployment instead predicts a future event for an existing customer, the evaluation design may differ.

TAKE IT FURTHER

Write down exactly what kind of unseen situation your chosen split represents.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Baselines before models

THE GOAL

Measure a simple strategy that any complex model must beat.

A baseline is a deliberately simple predictor. For classification it might always predict the most common training class. For squared-error regression it might predict the training mean. A baseline tells you whether added complexity creates value. Always compute it using training data only.

lesson_06_example.py

```
from collections import Counter
labels = ["A", "A", "B", "A"]
majority = Counter(labels).most_common(1)[0][0]
predictions = [majority] * 3
assert predictions == ["A", "A", "A"]
```

The baseline uses no features. If a trained model cannot improve a relevant metric over it, inspect the data and problem before adding complexity.

WATCH FOR THIS

A baseline computed from the test targets leaks information about the evaluation set.

Your turn: Baselines before models

THE CHALLENGE

Predict the training mean for a test set and compute mean absolute error. Reject empty training data.

ACCEPTANCE CHECKS

[ ] Train [2,4,6] -> mean 4

[ ] Test targets [3,5] -> MAE 1

[ ] Empty training -> ValueError

A NUDGE, IF YOU NEED ONE

Compute the prediction once from training, then compare it with each test target.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BASELINES BEFORE MODELS

lesson_06_solution.py

```
def mean_baseline(train, test):
    if not train or not test:
        raise ValueError("Non-empty train and test required")
    prediction = sum(train) / len(train)
    mae = sum(abs(y - prediction) for y in test) / len(test)
    return prediction, mae
assert mean_baseline([2, 4, 6], [3, 5]) == (4.0, 1.0)
```

WHY IT WORKS

The mean is the natural constant predictor for squared error, while the median minimises absolute error. This exercise uses the mean baseline and evaluates it with MAE intentionally.

TAKE IT FURTHER

Compare mean and median baselines on training data with one extreme outlier.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Missing-value strategies

THE GOAL

Learn replacement values from training data only.

Imputation replaces missing values according to a policy. A mean can be reasonable for some numeric features but inappropriate for categories or informative absence. Add a missingness indicator when absence itself may matter. Fit replacement values on training data and reuse them unchanged for validation and test.

lesson_07_example.py

```
train = [2.0, None, 6.0]
known = [x for x in train if x is not None]
fill = sum(known) / len(known)
clean = [fill if x is None else x for x in train]
assert clean == [2.0, 4.0, 6.0]
```

The replacement value is 4. If future data includes a missing value, it receives that same fitted value rather than a mean recalculated from the new batch.

WATCH FOR THIS

Replacing every missing value with zero can create a false measurement that the model treats as real.

Your turn: Missing-value strategies

THE CHALLENGE

Write fit_fill and transform functions. Reject training data containing no known values. Return transformed values plus missing indicators.

ACCEPTANCE CHECKS

[ ] Train [2,None,6] -> fill 4

[ ] Transform [None,10] -> [4,10] and [1,0]

[ ] All-missing training -> ValueError

A NUDGE, IF YOU NEED ONE

Separate learning the fill value from applying it.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

MISSING-VALUE STRATEGIES

lesson_07_solution.py

```
def fit_fill(values):
    known = [v for v in values if v is not None]
    if not known:
        raise ValueError("No observed training values")
    return sum(known) / len(known)
def transform(values, fill):
    return ([fill if v is None else v for v in values],
            [int(v is None) for v in values])
fill = fit_fill([2, None, 6])
assert transform([None, 10], fill) == ([4.0, 10], [1, 0])
```

WHY IT WORKS

The two functions make leakage harder to introduce accidentally. Keep the fitted fill value with the model so prediction-time preprocessing matches training.

TAKE IT FURTHER

Compare two policies on validation data and record which errors change, not only the overall metric.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Scaling and feature units

THE GOAL

Prevent arbitrary units from dominating distance-based models.

Features measured on very different scales can dominate distance and optimisation. Standardisation subtracts a training mean and divides by a training standard deviation. Constant features need special handling to avoid division by zero. Tree-based models often need different preprocessing choices.

lesson_08_example.py

```
import numpy as np
X = np.array([[1.0, 10.0], [3.0, 30.0], [5.0, 50.0]])
mean = X.mean(axis=0)
scale = X.std(axis=0)
scale = np.where(scale == 0, 1.0, scale)
Z = (X - mean) / scale
assert np.allclose(Z.mean(axis=0), [0, 0])
```

axis=0 computes one statistic per feature column. The transformation is fitted on X here; a real experiment uses only its training portion.

WATCH FOR THIS

Fitting scaling on the full dataset before splitting leaks test-distribution information into training.

Your turn: Scaling and feature units

THE CHALLENGE

Scale a constant training feature and transform a new value without producing infinity or NaN.

ACCEPTANCE CHECKS

[ ] Training [5,5,5] -> all zeros

[ ] Stored scale becomes 1

[ ] New value 7 -> transformed value 2

A NUDGE, IF YOU NEED ONE

Replace a zero standard deviation with one before division.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SCALING AND FEATURE UNITS

lesson_08_solution.py

```
import numpy as np
train = np.array([5.0, 5.0, 5.0])
mean = train.mean()
scale = train.std()
if scale == 0:
    scale = 1.0
assert np.allclose((train - mean) / scale, 0)
assert (7.0 - mean) / scale == 2.0
```

WHY IT WORKS

This fallback prevents a numerical failure but does not make a constant training feature informative. You may choose to remove such features after inspecting the problem.

TAKE IT FURTHER

Explain why centimetres versus metres can change nearest-neighbour results when scaling is omitted.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Linear regression

THE GOAL

Fit a line and understand what it assumes.

Linear regression models a target as a weighted sum of features plus an intercept. Least squares chooses weights that minimise squared residuals. A good fit on training examples does not prove causation or reliable extrapolation. Inspect residuals and evaluate on held-out data.

lesson_09_example.py

```
import numpy as np
x = np.array([1.0, 2.0, 3.0, 4.0])
y = np.array([3.0, 5.0, 7.0, 9.0])
X = np.column_stack([np.ones(len(x)), x])
weights, *_ = np.linalg.lstsq(X, y, rcond=None)
assert np.allclose(weights, [1.0, 2.0])
assert np.isclose(np.array([1.0, 5.0]) @ weights, 11.0)
```

The first column represents the intercept. The fitted relationship is y = 1 + 2x for this synthetic dataset. Real measurements rarely fit perfectly.

WATCH FOR THIS

A linear relationship in observed data can fail outside the observed range.

Your turn: Linear regression

THE CHALLENGE

Fit a line to y = 4 + 3x using three examples and predict at x = 4. Verify the training residuals are near zero.

ACCEPTANCE CHECKS

[ ] Weights near [4,3]

[ ] Prediction at 4 near 16

[ ] Residuals near zero

A NUDGE, IF YOU NEED ONE

Use a column of ones and np.linalg.lstsq rather than explicitly inverting a matrix.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LINEAR REGRESSION

lesson_09_solution.py

```
import numpy as np
x = np.array([1.0, 2.0, 3.0])
y = np.array([7.0, 10.0, 13.0])
X = np.column_stack([np.ones(len(x)), x])
w, *_ = np.linalg.lstsq(X, y, rcond=None)
assert np.allclose(w, [4.0, 3.0])
assert np.isclose(np.array([1.0, 4.0]) @ w, 16.0)
assert np.allclose(y - X @ w, 0)
```

WHY IT WORKS

This exercise checks mechanics on known data. A real regression project needs uncertainty discussion, unit checks and evaluation on examples not used to fit the line.

TAKE IT FURTHER

Add noise and one outlier, then inspect how the fitted line and absolute errors change.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Loss and gradient descent

THE GOAL

Update parameters in a direction that reduces error.

A loss measures prediction error. Gradient descent changes parameters opposite the loss gradient. The learning rate controls step size: too large can diverge, too small can be slow. Track loss rather than assuming more iterations always improve a model.

lesson_10_example.py

```
weight = 0.0
rate = 0.1
losses = []
for _ in range(30):
    loss = (weight - 3.0) ** 2
    losses.append(loss)
    gradient = 2 * (weight - 3.0)
    weight -= rate * gradient
assert losses[-1] < losses[0]
assert abs(weight - 3.0) < 0.01
```

The target is the minimum of a simple quadratic. This is a transparent optimisation exercise, not a complete model-training implementation.

WATCH FOR THIS

One stable learning rate on a toy problem is not a universal setting for every model.

Your turn: Loss and gradient descent

THE CHALLENGE

Optimise the same quadratic from two different initial weights. Confirm both approach 3 using the same stable learning rate.

ACCEPTANCE CHECKS

[ ] Start -5 -> near 3

[ ] Start 10 -> near 3

[ ] Loss decreases by the final iteration

A NUDGE, IF YOU NEED ONE

Put the loop into a function that returns the final weight.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LOSS AND GRADIENT DESCENT

lesson_10_solution.py

```
def optimise(start, rate=0.1, steps=100):
    weight = start
    for _ in range(steps):
        weight -= rate * 2 * (weight - 3)
    return weight
assert abs(optimise(-5) - 3) < 1e-6
assert abs(optimise(10) - 3) < 1e-6
assert abs(optimise(-5, steps=0) + 5) < 1e-9
```

WHY IT WORKS

The zero-step check confirms the function does not change a parameter without an update. Real losses can be non-convex and have more complicated optimisation behaviour.

TAKE IT FURTHER

Try an intentionally excessive learning rate and plot or print the loss trajectory to diagnose divergence.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Nearest-neighbour classification

THE GOAL

Predict from similar labelled examples.

k-nearest neighbours finds nearby training examples and combines their labels. It makes few explicit training calculations but prediction can be expensive on large data. Distances depend on feature scale and the distance rule. Choose k and tie handling using validation data.

lesson_11_example.py

```
def nearest_label(points, query):
    if not points:
        raise ValueError("Training examples required")
    return min(points, key=lambda p: abs(p[0]-query))[1]
points = [(1.0, "low"), (5.0, "high"), (8.0, "high")]
assert nearest_label(points, 2.0) == "low"
assert nearest_label(points, 7.0) == "high"
```

This one-dimensional, one-neighbour example chooses the first minimum in input order when distances tie. That tie rule is part of its behaviour.

WATCH FOR THIS

High accuracy on training points can be trivial for one-neighbour classification and says little about generalisation.

Your turn: Nearest-neighbour classification

THE CHALLENGE

Implement k-neighbour majority voting for one-dimensional data. Break label-count ties alphabetically for a deterministic teaching rule.

ACCEPTANCE CHECKS

[ ] Three nearest labels A,A,B -> A

[ ] k larger than data -> reject

[ ] Empty data -> reject

A NUDGE, IF YOU NEED ONE

Sort by distance, take k and count labels. Sort counts by negative count then label.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

NEAREST-NEIGHBOUR CLASSIFICATION

lesson_11_solution.py

```
from collections import Counter

def classify(points, query, k):
    if not 1 <= k <= len(points):
        raise ValueError("Invalid k")
    near = sorted(points, key=lambda p: abs(p[0]-query))[:k]
    counts = Counter(label for _, label in near)
    return min(counts, key=lambda label: (-counts[label], label))
assert classify([(1,"A"),(2,"A"),(3,"B")], 2, 3) == "A"
assert classify([(1,"B"),(3,"A")], 2, 2) == "A"
```

WHY IT WORKS

The deterministic tie policy makes tests reproducible, but it is not necessarily the best business policy. Equal-distance neighbour selection can also need a defined rule.

TAKE IT FURTHER

Create a held-out set and compare k = 1 and k = 3 using the same split.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Probabilities and thresholds

THE GOAL

Separate a score from the action chosen from it.

A classifier may produce a score between zero and one. A threshold converts it into a decision. Lowering the threshold usually catches more positives but may create more false positives. A score is not necessarily a well-calibrated probability; calibration must be evaluated on representative data.

lesson_12_example.py

```
from math import exp

def sigmoid(value):
    if value >= 0:
        return 1 / (1 + exp(-value))
    z = exp(value)
    return z / (1 + z)
assert sigmoid(0) == 0.5
assert 0 < sigmoid(-10) < sigmoid(10) < 1
```

The branch avoids an unnecessarily large exponential for very negative values. This is a numeric building block used in logistic models.

WATCH FOR THIS

A model confidence number should not be presented as a guarantee that a prediction is correct.

Your turn: Probabilities and thresholds

THE CHALLENGE

Convert scores into positive labels at thresholds 0.5 and 0.8. Count how many predictions change.

ACCEPTANCE CHECKS

[ ] Scores [0.2,0.6,0.9] at 0.5 -> [0,1,1]

[ ] At 0.8 -> [0,0,1]

[ ] One decision changes

A NUDGE, IF YOU NEED ONE

Define whether equality with the threshold counts as positive.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROBABILITIES AND THRESHOLDS

lesson_12_solution.py

```
def decisions(scores, threshold):
    if not 0 <= threshold <= 1:
        raise ValueError("Threshold outside [0,1]")
    return [int(score >= threshold) for score in scores]
scores = [0.2, 0.6, 0.9]
a, b = decisions(scores, 0.5), decisions(scores, 0.8)
assert a == [0, 1, 1] and b == [0, 0, 1]
assert sum(x != y for x, y in zip(a, b)) == 1
```

WHY IT WORKS

Threshold selection should reflect the cost of mistakes and available review capacity. Choose it using validation data, then report final test results once.

TAKE IT FURTHER

Add an abstain region for scores between two thresholds and measure how many cases need human review.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Metrics that match mistakes

THE GOAL

Inspect false positives and false negatives separately.

Accuracy counts correct labels, but can hide failure on a rare class. Precision asks how many predicted positives are truly positive. Recall asks how many actual positives were found. F1 combines precision and recall. Always state which label is considered positive and how zero denominators are handled.

lesson_13_example.py

```
truth = [1, 1, 0, 0]
pred = [1, 0, 1, 0]
tp = sum(a == b == 1 for a, b in zip(truth, pred))
fp = sum(a == 0 and b == 1 for a, b in zip(truth, pred))
fn = sum(a == 1 and b == 0 for a, b in zip(truth, pred))
assert (tp, fp, fn) == (1, 1, 1)
```

The classifier finds one positive, misses one and incorrectly flags one negative. The same accuracy could come from different operationally important errors.

WATCH FOR THIS

Changing the positive class changes precision and recall interpretation.

Your turn: Metrics that match mistakes

THE CHALLENGE

Compute precision, recall and F1 from tp, fp and fn. Return zero for a metric whose denominator is zero in this exercise.

ACCEPTANCE CHECKS

[ ] tp=1,fp=1,fn=1 -> all 0.5

[ ] tp=0,fp=0,fn=0 -> all 0

[ ] Perfect positives -> all 1

A NUDGE, IF YOU NEED ONE

Use a helper for safe division and document the zero convention.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

METRICS THAT MATCH MISTAKES

lesson_13_solution.py

```
def metrics(tp, fp, fn):
    precision = tp / (tp + fp) if tp + fp else 0.0
    recall = tp / (tp + fn) if tp + fn else 0.0
    f1 = 2*precision*recall/(precision+recall) if precision+recall else 0.0
    return precision, recall, f1
assert metrics(1, 1, 1) == (0.5, 0.5, 0.5)
assert metrics(0, 0, 0) == (0.0, 0.0, 0.0)
assert metrics(3, 0, 0) == (1.0, 1.0, 1.0)
```

WHY IT WORKS

For multiclass problems, report per-class results and explain macro versus weighted averaging. A single summary number cannot show every failure pattern.

TAKE IT FURTHER

Construct a dataset with 99 negatives and one positive, then show why always-negative accuracy is misleading.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Overfitting and model choice

THE GOAL

Prefer evidence from held-out data over training perfection.

Overfitting occurs when a model learns details that do not generalise. More flexible models can reduce training error while increasing validation error. Regularisation, simpler models, better data and careful validation can help. The final test set should not become a repeated tuning tool.

lesson_14_example.py

```
experiments = [
    {"depth": 1, "train_error": 0.20, "valid_error": 0.22},
    {"depth": 3, "train_error": 0.10, "valid_error": 0.15},
    {"depth": 9, "train_error": 0.00, "valid_error": 0.30},
]
best = min(experiments, key=lambda row: row["valid_error"])
assert best["depth"] == 3
```

The deepest model fits training perfectly but validates worst. These are illustrative experiment records, not measured performance claims for a specific dataset.

WATCH FOR THIS

Trying hundreds of settings on one validation set can overfit the validation process itself.

Your turn: Overfitting and model choice

THE CHALLENGE

Choose between two models whose validation scores are effectively tied. Use a declared simplicity preference rather than looking at the final test set.

ACCEPTANCE CHECKS

[ ] Error difference <= 0.01 -> prefer lower complexity

[ ] Larger error difference -> prefer lower validation error

[ ] No test scores consulted

A NUDGE, IF YOU NEED ONE

First find the minimum error, then choose the simplest eligible model within tolerance.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

OVERFITTING AND MODEL CHOICE

lesson_14_solution.py

```
def choose(models, tolerance=0.01):
    best = min(m["error"] for m in models)
    eligible = [m for m in models if m["error"] <= best+tolerance]
    return min(eligible, key=lambda m: m["complexity"])
models = [{"name":"simple","error":0.151,"complexity":1},
          {"name":"large","error":0.15,"complexity":10}]
assert choose(models)["name"] == "simple"
```

WHY IT WORKS

The tolerance is a project policy, not a statistical significance test. Repeated trials and uncertainty estimates may be needed to judge whether differences are meaningful.

TAKE IT FURTHER

Log every tried setting and decide a tuning budget before looking at results.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Cross-validation and leakage

THE GOAL

Repeat evaluation while keeping preprocessing inside each fold.

Cross-validation rotates held-out subsets so one split does not dominate the estimate. Every learned preprocessing step must fit within the training portion of each fold. Group and time constraints still apply; ordinary shuffled folds are inappropriate for many dependent datasets.

lesson_15_example.py

```
def folds(n, k):
    if not 2 <= k <= n:
        raise ValueError("Invalid fold count")
    return [list(range(i, n, k)) for i in range(k)]
parts = folds(9, 3)
assert sorted(x for part in parts for x in part) == list(range(9))
assert all(len(part) == 3 for part in parts)
```

This deterministic splitter teaches disjoint folds. It is not stratified, grouped or shuffled, so use it only when those omissions match the experiment.

WATCH FOR THIS

Calling a scaler on the full dataset before cross-validation makes the folds less independent.

Your turn: Cross-validation and leakage

THE CHALLENGE

For each fold, construct training indexes from every other fold and prove they are disjoint from validation indexes.

ACCEPTANCE CHECKS

[ ] Each index is validated once

[ ] No overlap in any train/validation pair

[ ] Training plus validation covers all indexes

A NUDGE, IF YOU NEED ONE

Build the full index set, then subtract the held-out fold.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CROSS-VALIDATION AND LEAKAGE

lesson_15_solution.py

```
all_ids = set(range(9))
folds = [set(range(i, 9, 3)) for i in range(3)]
for valid in folds:
    train = all_ids - valid
    assert train.isdisjoint(valid)
    assert train | valid == all_ids
assert set.union(*folds) == all_ids
assert sum(len(f) for f in folds) == len(all_ids)
```

WHY IT WORKS

Index isolation is necessary but not sufficient: duplicate people, future-derived fields and preprocessing fitted globally can still leak information.

TAKE IT FURTHER

Implement a fold-specific mean imputer and compare its fitted values across folds.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Clustering without labels

THE GOAL

Discover groups while resisting unsupported interpretations.

Clustering groups observations according to a distance or similarity rule. k-means alternates assigning points to centres and recomputing those centres. Results depend on scaling, initialisation and k. A cluster number is not automatically a meaningful customer type or scientific category.

lesson_16_example.py

```
import numpy as np
points = np.array([0.0, 1.0, 9.0, 10.0])
centres = np.array([0.0, 10.0])
labels = np.abs(points[:, None] - centres).argmin(axis=1)
updated = np.array([points[labels == i].mean() for i in range(2)])
assert np.allclose(updated, [0.5, 9.5])
```

Broadcasting creates one distance per point-centre pair. argmin selects the closest centre for each point.

WATCH FOR THIS

Renaming clusters with human descriptions can imply evidence that the algorithm never supplied.

Your turn: Clustering without labels

THE CHALLENGE

Run several k-means updates on the same one-dimensional points. Keep an old centre if a cluster becomes empty.

ACCEPTANCE CHECKS

[ ] Two groups converge near 0.5 and 9.5

[ ] Every point gets one label

[ ] Empty clusters do not create NaN

A NUDGE, IF YOU NEED ONE

Check whether a cluster has members before taking its mean.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CLUSTERING WITHOUT LABELS

lesson_16_solution.py

```
import numpy as np
points = np.array([0.0, 1.0, 9.0, 10.0])
centres = np.array([0.0, 10.0])
for _ in range(5):
    labels = np.abs(points[:, None] - centres).argmin(axis=1)
    for i in range(len(centres)):
        members = points[labels == i]
        if len(members):
            centres[i] = members.mean()
assert np.allclose(sorted(centres), [0.5, 9.5])
assert np.isfinite(centres).all()
```

WHY IT WORKS

The toy example has obvious separation. Real clustering needs sensitivity analysis, domain review and a reason why the groups help an actual decision.

TAKE IT FURTHER

Try a different initialisation and an outlier, then describe how the result changes.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Text features and similarity

THE GOAL

Turn words into a representation you can inspect.

Text models need a representation. A bag-of-words count ignores word order but is easy to understand. Tokenisation and normalisation determine what counts as a word. Fit vocabulary choices on training data. More advanced embeddings capture other patterns but also require evaluation.

lesson_17_example.py

```
from collections import Counter

def bag(text):
    return Counter(text.casefold().split())
a = bag("python web python")
b = bag("python data")
assert a["python"] == 2
assert set(a) & set(b) == {"python"}
```

This simple tokeniser treats punctuation as part of a token. That limitation is acceptable only when documented and tested for the intended data.

WATCH FOR THIS

High lexical similarity does not prove that a document answers the user's question.

Your turn: Text features and similarity

THE CHALLENGE

Compute Jaccard similarity between two sets of words. Define two empty sets to have similarity one for this exercise.

ACCEPTANCE CHECKS

[ ] {python,web} and {python,data} -> 1/3

[ ] Identical sets -> 1

[ ] Disjoint nonempty sets -> 0

A NUDGE, IF YOU NEED ONE

Divide intersection size by union size, guarding an empty union.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TEXT FEATURES AND SIMILARITY

lesson_17_solution.py

```
def jaccard(left, right):
    union = left | right
    return len(left & right) / len(union) if union else 1.0
assert jaccard({"python","web"}, {"python","data"}) == 1/3
assert jaccard({"a"}, {"a"}) == 1
assert jaccard({"a"}, {"b"}) == 0
assert jaccard(set(), set()) == 1
```

WHY IT WORKS

The score measures token overlap, not semantic truth. Two sentences can share words while expressing opposite meanings.

TAKE IT FURTHER

Compare whitespace tokenisation with a documented punctuation-normalisation rule on five examples.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Retrieval before generation

THE GOAL

Find supporting material and preserve its source.

An AI document assistant can retrieve relevant passages before drafting an answer. Retrieval and generation are separate steps. First verify that the search returns useful evidence; adding a language model cannot repair missing or irrelevant sources automatically. Treat retrieved text as data, not instructions to the application.

lesson_18_example.py

```
documents = {"git": "git branches commits history",
             "web": "python flask routes templates",
             "data": "python arrays statistics"}
def retrieve(query):
    words = set(query.casefold().split())
    scores = [(len(words & set(text.split())), ident)
              for ident, text in documents.items()]
    score, ident = max(scores)
    return ident if score > 0 else None
assert retrieve("flask routes") == "web"
assert retrieve("ocean") is None
```

The baseline returns one source ID and abstains when there is no overlap. Ties follow tuple ordering here; a real search system should define a better tie policy.

WATCH FOR THIS

A document saying ignore previous instructions must not gain authority over your application's rules or credentials.

Your turn: Retrieval before generation

THE CHALLENGE

Return both source ID and source text, and include an explicit no-evidence result instead of inventing an answer.

ACCEPTANCE CHECKS

[ ] Matching query -> source plus passage

[ ] No match -> evidence_found False

[ ] Returned text is exactly the stored source

A NUDGE, IF YOU NEED ONE

Keep the source mapping available and construct a structured retrieval result.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

RETRIEVAL BEFORE GENERATION

lesson_18_solution.py

```
def evidence(query, docs):
    words = set(query.casefold().split())
    ranked = sorted(docs, key=lambda k:
                    (-len(words & set(docs[k].split())), k))
    if not ranked or not words.intersection(docs[ranked[0]].split()):
        return {"evidence_found": False}
    ident = ranked[0]
    return {"evidence_found": True, "source": ident,
            "passage": docs[ident]}
docs = {"a": "python flask routes"}
assert evidence("flask", docs)["passage"] == docs["a"]
assert evidence("ocean", docs) == {"evidence_found": False}
```

WHY IT WORKS

This is a retrieval component, not a language-model call. A later generation step should cite the source and be evaluated for unsupported claims.

TAKE IT FURTHER

Build a small question set with answerable and unanswerable cases and measure retrieval success before adding generation.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Model cards and monitoring

THE GOAL

Describe limits and watch for changing conditions.

A model card records purpose, data, evaluation, limitations and intended use. Monitoring checks whether inputs, error patterns or operating conditions change after release. A model can keep running technically while becoming less useful. Plan when to review, retrain or disable it.

lesson_19_example.py

```
card = {"purpose": "Suggest support categories",
        "data": "Fictional teaching messages",
        "limitations": ["Tiny sample", "English only"],
        "human_review": True,
        "version": "demo-1"}
assert card["limitations"]
assert card["human_review"] is True
```

The card makes the demonstration's limits visible. It should report measured results when available and distinguish them from intended goals.

WATCH FOR THIS

Monitoring only request latency cannot detect declining prediction quality.

Your turn: Model cards and monitoring

THE CHALLENGE

Compare a feature's missing-value rate between two batches and flag a large increase. Treat the threshold as an operational rule, not a proof of model failure.

ACCEPTANCE CHECKS

[ ] Old 10%, new 40%, threshold 20 percentage points -> flag

[ ] Unchanged rate -> no flag

[ ] Empty batch -> reject

A NUDGE, IF YOU NEED ONE

Compute each rate separately and subtract old from new.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

MODEL CARDS AND MONITORING

lesson_19_solution.py

```
def missing_rate(values):
    if not values:
        raise ValueError("Empty batch")
    return sum(v is None for v in values) / len(values)
old = [None] + [1] * 9
new = [None] * 4 + [1] * 6
assert missing_rate(new) - missing_rate(old) > 0.2
assert missing_rate(old) - missing_rate(old) == 0
```

WHY IT WORKS

A flagged shift prompts investigation. It may reflect collection changes, a new user population or a legitimate seasonal pattern.

TAKE IT FURTHER

Add an error-review process with a sample size, review frequency and named action when performance crosses a threshold.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Project: an honest experiment

THE GOAL

Compare a fitted model with a baseline on untouched examples.

Finish with a reproducible regression experiment using synthetic data. The purpose is to practise separation, fitting and evaluation, not to claim a useful business model. Save the feature definition, split and metric with the results. In a real project, repeat evaluation under realistic data conditions.

lesson_20_example.py

```
import numpy as np
x_train = np.array([1., 2., 3., 4.])
y_train = np.array([3., 5., 7., 9.])
x_test = np.array([5., 6.])
y_test = np.array([11., 13.])
X = np.column_stack([np.ones(len(x_train)), x_train])
w, *_ = np.linalg.lstsq(X, y_train, rcond=None)
pred = np.column_stack([np.ones(len(x_test)), x_test]) @ w
mae = np.abs(y_test - pred).mean()
baseline = np.abs(y_test - y_train.mean()).mean()
assert mae < 1e-10
assert baseline > mae
```

The line is exact because the synthetic data follows the same formula. This easy result is evidence that the code works on the constructed case, not evidence of real-world predictive power.

WATCH FOR THIS

Do not tune repeatedly on the final held-out targets and still call them an untouched test set.

Your turn: Project: an honest experiment

THE CHALLENGE

Package the experiment into a function returning baseline MAE, model MAE and coefficients. Add a shape check and a held-out evaluation.

ACCEPTANCE CHECKS

[ ] Exact synthetic line -> near-zero model MAE

[ ] Baseline is worse on this dataset

[ ] One prediction per held-out row

[ ] Results include the fitted coefficients

A NUDGE, IF YOU NEED ONE

Keep all training calculations inside the fit section and never use y_test to choose weights.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROJECT: AN HONEST EXPERIMENT

lesson_20_solution.py

```
import numpy as np

def experiment(x_train, y_train, x_test, y_test):
    x_train, y_train = map(np.asarray, (x_train, y_train))
    x_test, y_test = map(np.asarray, (x_test, y_test))
    if len(x_train) != len(y_train) or len(x_test) != len(y_test):
        raise ValueError("Mismatched rows")
    X = np.column_stack([np.ones(len(x_train)), x_train])
    w, *_ = np.linalg.lstsq(X, y_train, rcond=None)
    T = np.column_stack([np.ones(len(x_test)), x_test])
    return {"weights": w.tolist(),
            "model_mae": float(np.abs(y_test-T@w).mean()),
            "baseline_mae": float(np.abs(y_test-y_train.mean()).mean())}
r = experiment([1,2,3,4], [3,5,7,9], [5,6], [11,13])
assert r["model_mae"] < 1e-10
assert r["baseline_mae"] > r["model_mae"]
assert np.allclose(r["weights"], [1, 2])
```

WHY IT WORKS

The portfolio version should add non-empty and finite-value validation, a realistic dataset, a documented split and error analysis. Use a maintained modelling library for broader estimators and pipelines rather than extending every toy algorithm yourself.

TAKE IT FURTHER

Write a one-page experiment report: problem, dataset limits, baseline, split, metrics, worst errors and a recommendation about whether the model should be used.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

NumPy learning resources

https://numpy.org/learn/

scikit-learn getting started

https://scikit-learn.org/stable/getting_started.html

Avoiding leakage

https://scikit-learn.org/stable/common_pitfalls.html

Model evaluation

https://scikit-learn.org/stable/modules/model_evaluation.html

Text feature extraction

https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

Model persistence

https://scikit-learn.org/stable/model_persistence.html

WHAT THE CHECKS ESTABLISH

All 40 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.