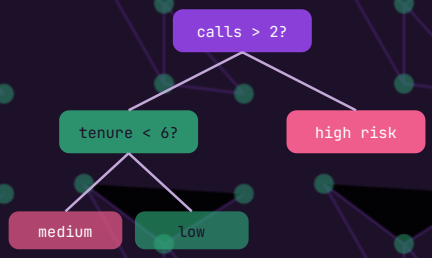
STUDY PAGE · AREA 04 OF 14

Supervised Learning with scikit-learn

Train, read and compare the models that solve most tabular problems at work: a baseline, linear and logistic regression, decision trees, random forests and gradient boosting.




```
model.fit(X_train, y_train)
model.predict_proba(X_test)
```

04

18 h of study · **4** modules · **8** diagrams · **3** interactives

Free tools · scikit-learn, or Google Colab · **20**-question test · 70% to pass

Lab: baseline vs logreg vs forest

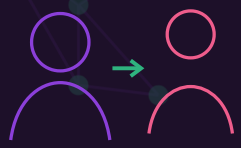
1 · Linear models → 2 · Decision trees → 3 · Ensembles → 4 · The API → Lab · churn models → Area test · 70%

regression · logistic | Gini · depth | forest · boosting | fit · Pipeline · search | three bugs | 20 questions

BEFORE YOU START · A

Who this area is for

This is the core ML toolbox. Most production models on tables — churn, fraud, pricing, demand — are one of these five.



CS / data / stats students

Students preparing for ML internships and new-grad roles who want the practical path, not only theory.

Software engineers

Backend and full-stack developers moving into ML engineering, who already write good code and want the ML on top.

Data analysts

Analysts who know SQL and spreadsheets and want to build and ship models, not just report on data.

QA & test engineers

Test automation engineers heading into ML: evaluation, data checks and model testing use the same mindset.

Data engineers

Pipeline builders who want to own the models their data feeds, and the MLOps around them.

Career switchers

Engineers from other fields (EE, mechanical, finance) with some Python, aiming for an ML engineer job.

Prerequisites

You need	Level	Where to get it
Data prep and leak-free pipelines	Comfortable	Area 03
Gradient descent, sigmoid, log-loss	Basic	Area 02
pandas	Comfortable	Area 01

Where this area sits in the ML lifecycle — and the roles that use it



Highlighted: this area is about training models

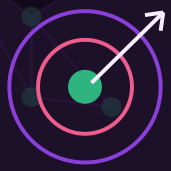
Training turns prepared features into a model. Area 05 then judges it.

Role	How you use this area on the job	Typical first title
ML engineer	Train, compare and ship tabular models with reproducible pipelines	ML Engineer I / Machine Learning Engineer
Data scientist	Choose and explain models; read coefficients and importances	Data Scientist / Applied Scientist
AI / LLM engineer	Use classic models as cheap baselines and routers around LLMs	AI Engineer / LLM Engineer
MLOps / platform	Package fitted pipelines with joblib and track hyperparameters	MLOps Engineer / ML Platform Engineer
Data engineer	Provide features in the shape models need; understand training cost	Data Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.

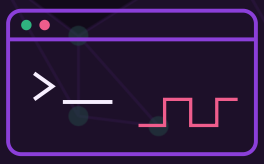


- Build a baseline every model must beat.
- Fit and interpret linear and logistic regression.
- Explain how a decision tree picks splits with Gini impurity.
- Diagnose overfitting from the train–test gap and control tree depth.
- Train random forests and gradient boosting and say how they differ.
- Measure feature importance with permutation importance.
- Use fit / predict / predict_proba and chain steps in a Pipeline.
- Tune hyperparameters with GridSearchCV and save a model with joblib.

BEFORE YOU START · C

Tools & software

scikit-learn covers every model in this area with one API.



Tool	What it does	Used in this area for	Cost
Python 3.10 – 3.13	The language of ML: every library on this page is a Python package	Every example and the lab	Free
venv + **pip**	An isolated environment per project and the package installer	Installing the area's packages	Free
pandas	Tables (DataFrames): load, clean, group and join data	Loading data and results tables	Free
NumPy	Fast n-dimensional arrays and maths	Arrays for features and predictions	Free
scikit-learn	Classic ML: preprocessing, models, metrics, pipelines	Models, Pipeline, GridSearchCV	Free
Matplotlib	Charts: histograms, scatter plots, learning curves	Plotting fits and depth curves	Free
JupyterLab	Notebooks: run code cell by cell and see tables and plots inline	Comparing models cell by cell	Free
Google Colab	Hosted notebooks in the browser, with a free GPU on request	No-install option	Free

Platforms you will meet at work

Platform	Vendor	What teams use it for
SageMaker	AWS	Managed notebooks, training jobs, model registry and endpoints
Vertex AI	Google Cloud	Training, pipelines, feature store, model serving and Gemini APIs
Azure Machine Learning	Microsoft	Workspaces, MLflow tracking, managed endpoints
Databricks	Databricks	Spark + notebooks + MLflow on a lakehouse
Weights & Biases	W&B	Experiment tracking and model evaluation dashboards

i

Same code, bigger machines

XGBoost and LightGBM follow the same fit/predict API; scikit-learn's HistGradientBoostingClassifier is built in.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, Google Colab works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows) — the same commands as Linux, cloud servers and most ML teams.

1

Turn on WSL and install Ubuntu

Open **PowerShell as Administrator**, run the command, then restart.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2

Install Python's venv support

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip git make
```

3

Create an environment and install the packages

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install pandas numpy scikit-learn matplotlib joblib jupyterlab
```

Run `source ~/ml/bin/activate` in every new terminal.

4

Editor

Install VS Code, the **WSL**, **Python** and **Jupyter** extensions, then type `code .` in Ubuntu.

5

Check it works

```
python -c "import sklearn; print(sklearn.__version__)"
```

i

No WSL on your laptop?

Install Python 3.12 from python.org (tick *Add python.exe to PATH*), then in PowerShell: `py -m venv %USERPROFILE%\ml` , `%USERPROFILE%\ml\Scripts\activate` and the same `pip install` line. Every lab runs the same.

macOS

1 **Install Apple's command-line tools**

```
xcode-select --install
```

2 **Install Homebrew and Python**

Follow the one-line command on brew.sh, then:

```
brew install python@3.12 git make
```

3 **Create an environment and install the packages**

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install pandas numpy scikit-learn matplotlib joblib jupyterlab
```

4 **Editor**

Install VS Code with the **Python** and **Jupyter** extensions. Apple-silicon Macs run every lab natively.

5 **Check it works**

```
python -c "import sklearn; print(sklearn.__version__)"
```

Linux

1 **Ubuntu / Debian**

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip git make
```

2 **Fedora**

```
sudo dnf install -y python3 python3-pip git make
```

3 **Arch**

```
sudo pacman -S python python-pip git make
```

4 **Create an environment and install the packages**

```
python3 -m venv ~/ml
source ~/ml/bin/activate
pip install --upgrade pip
pip install pandas numpy scikit-learn matplotlib joblib jupyterlab
```

5 **Check it works**

```
python -c "import sklearn; print(sklearn.__version__)"
```

Browser only

1 **Open Google Colab**

Go to colab.research.google.com and sign in with a Google account.

2 **New notebook**

File → New notebook. NumPy, pandas, scikit-learn, Matplotlib and PyTorch are already installed.

3 **Upload the lab**

Click the folder icon, upload the lab zip, then run `!unzip lab.zip && cd lab && python check.py` in a cell (prefix shell commands with `!`).

4 **Check the versions**

```
python -c "import sklearn; print(sklearn.__version__)"
```

BEFORE YOU START · E

First run in 5 minutes

Train a logistic regression churn model on the lab data and read its probabilities.



1 **Get the kit**

Download the lab kit (it contains `prep.py` and `make_data.py`) and run `python make_data.py`.

2 **Create `first.py` next to them**

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

model = Pipeline([("pre", preprocessor()), ("clf", LogisticRegression(ma
model.fit(X_tr, y_tr)
print("test accuracy:", round(model.score(X_te, y_te), 3))
print("P(churn), first 5 customers:", model.predict_proba(X_te.head())[
```

3 **Run it**

```
python first.py
```

Output (captured from a real run):

```
test accuracy: 0.743
P(churn), first 5 customers: [0.06 0.26 0.35 0.27 0.17]
```

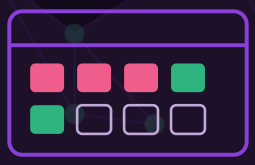
4 **What to notice**

Accuracy looks fine, but always guessing “stays” already scores 0.70. The probabilities are the useful output, which is why Module 1 starts with a baseline.

BEFORE YOU START · F

Study plan

18 hours at about 9 hours a week.



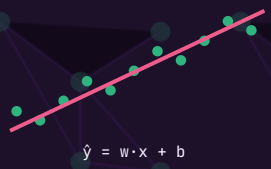
Session	Do this	Time
Day 1	Install · first run · Module 1 (linear & logistic) + line-fit widget	4 h
Day 2	Module 2 (trees, Gini, overfitting) + tree widget	4 h
Day 3	Module 3 (forests, boosting, importance)	4 h
Day 4	Module 4 (API, Pipeline, GridSearchCV, joblib)	3.5 h
Day 5	Lab: churn models · area test	2.5 h

MODULE 1 OF 4

Linear & logistic regression

The simplest models that work — and the baseline that keeps you honest.

- Baselines
- Linear regression and least squares
- Logistic regression and probabilities
- Reading coefficients



⏱ 4 hours | Area 04 · Supervised ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

1 Learning from answered examples

≈ In plain words

Show the computer many rows where you already know the answer. It learns a rule that maps inputs to the answer, then applies it to new rows.

✳ Think of it like

Learning to price used cars by looking at hundreds of past sales.

✎ Worked example

tenure	calls	churned?
2	4	yes
30	0	no
5	3	?

Aa The word for it	**Supervised learning**: **regression** for numbers, **classification** for categories.
! Why it matters	Most business ML — churn, fraud, demand, pricing — is supervised.

2 STEP 2 Beat the dumbest guess first

≈ In plain words	Before building a model, see how well “always guess the most common answer” does.
✳ Think of it like	A weather forecaster who always says “same as yesterday” — surprisingly hard to beat.
✎ Worked example	1. 70% stay, 30% churn 2. always guess “stay” → 70% accuracy 3. a model scoring 72% adds very little
Aa The word for it	**Baseline**.
! Why it matters	It tells you whether a model's score means anything.

3 STEP 3 A straight line, then a squashed line

≈ In plain words	Linear regression draws the best straight line through points. Logistic regression bends that line's output into a 0–1 probability.
✳ Think of it like	A dimmer switch: the linear part turns the knob, the sigmoid keeps the light between off and full.

Worked example

```
import numpy as np
z = np.array([-3, 0, 3])
print((1 / (1 + np.exp(-z))).round(3))
```

```
# output
[0.047 0.5   0.953]
```

The word for it — **Linear regression**, **logistic regression**, **sigmoid**.

Why it matters — They're fast, explainable and often all you need.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Supervised learning
Learning a mapping from inputs X to a known answer y, from labelled examples.

Regression
Predicting a number: price, demand, minutes.

Classification
Predicting a category: churn yes/no, spam, digit 0–9.

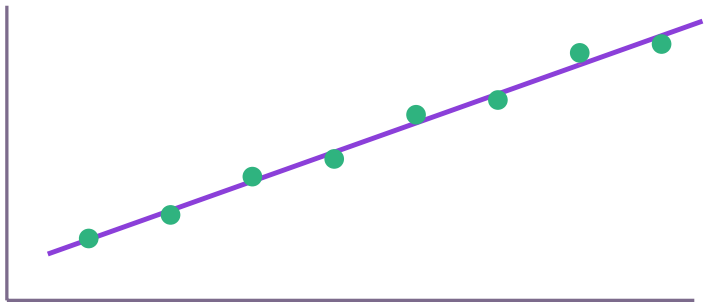
Linear regression
$\hat{y} = w \cdot x + b$, with w and b chosen to minimise squared error.

Coefficient
A learned weight; on standardized features its size compares importance and its sign gives direction.

Logistic regression
A linear score passed through the sigmoid to give a probability; a classifier.

Regularization
A penalty on large weights (L2 ridge, L1 lasso) that reduces overfitting; C = 1/strength in scikit-learn.

Baseline
The simplest honest model — predict the mean, or the most frequent class. Every model must beat it.



Least squares

$\hat{y} = 0.99 \cdot x + 1.08$

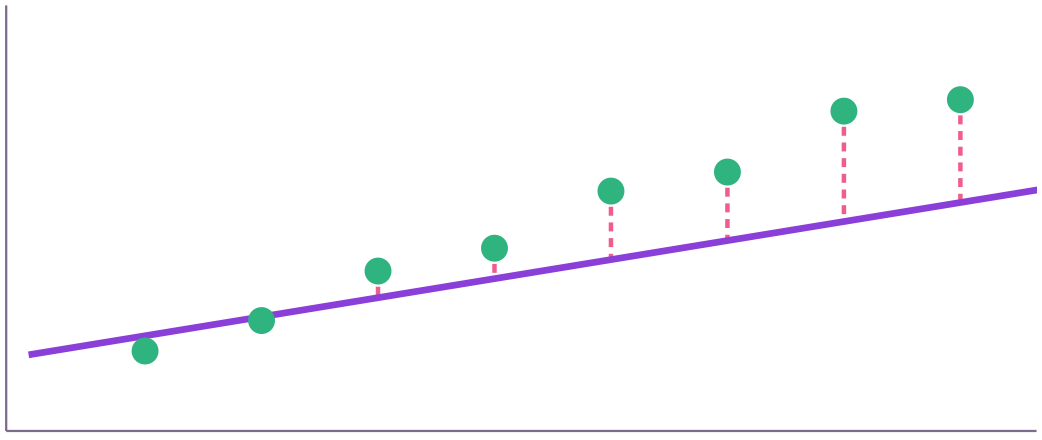
dashed = residual $(y - \hat{y})$

fit = choose w, b to minimise Σ residual² (the MSE of Area 02)

Linear regression is gradient descent on MSE — or a closed-form solve. Either way: the line with the smallest squared residuals.

Try it: fit a line by hand

slope w intercept b

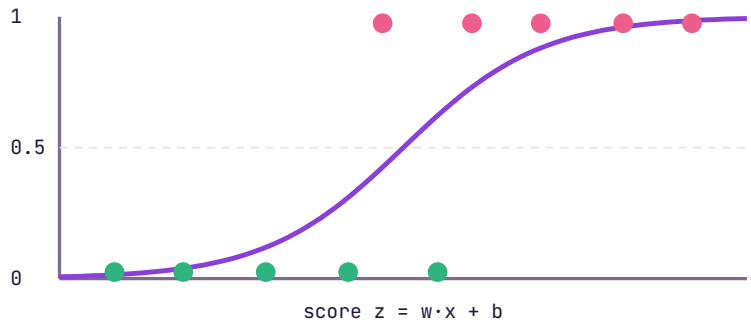


```
your line: ŷ = 0.50·x + 2.00   MSE = 2.935
best fit:  ŷ = 0.99·x + 1.08   MSE = 0.059
Your MSE is 50.1× the best — shrink the dashed residuals.
```

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

x = np.array([1, 2, 3, 4, 5, 6, 7, 8]).reshape(-1, 1)     # X must be 2-D: (n_sa
y = np.array([2.1, 2.9, 4.2, 4.8, 6.3, 6.8, 8.4, 8.7])
reg = LinearRegression().fit(x, y)
print("w =", reg.coef_.round(3), " b =", round(reg.intercept_, 3))
print("prediction for x = 10:", reg.predict([[10]]).round(2))
print("R² =", round(reg.score(x, y), 4))
```

```
# output
w = [0.988]  b = 1.079
prediction for x = 10: [10.96]
R² = 0.9887
```



p = σ(z) = 1 / (1 + e⁻ᶻ)

p ≥ 0.5 → predict churn

pink = churned, green = stayed

trained with log-loss
(cross-entropy, Area 02)

Despite the name: a classifier.

Logistic regression = linear score + sigmoid. The output is a probability; the 0.5 threshold is a business choice (Area 05).

```python
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

base = DummyClassifier(strategy="most_frequent").fit(X_tr, y_tr)
print("baseline accuracy:", round(base.score(X_te, y_te), 3))

logreg = Pipeline([("pre", preprocessor()), ("clf", LogisticRegression(max_iter=
print("logreg accuracy  :", round(logreg.score(X_te, y_te), 3))
names = logreg.named_steps["pre"].get_feature_names_out()
coefs = logreg.named_steps["clf"].coef_[0]
for n, c in sorted(zip(names, coefs), key=lambda t: -abs(t[1]))[:5]:
    print(f"  {n:24} {c:+.3f}")
print("P(churn) first 3 test rows:", logreg.predict_proba(X_te.head(3))[:, 1].ro
```

```
# output
baseline accuracy: 0.7
logreg accuracy  : 0.743
  num__tenure_months       -0.693
  num__support_calls       +0.692
  cat__plan_basic          +0.461
  cat__plan_pro            -0.273
  cat__plan_plus           -0.187
P(churn) first 3 test rows: [0.055 0.258 0.351]
```

i

Read the coefficients

Positive coefficients raise churn probability; on standardized inputs bigger magnitude = bigger effect. More support calls → more churn; longer tenure → less. A model you can explain is easier to ship.

Model	Predicts	Loss	scikit-learn
Linear regression	a number	squared error	`LinearRegression`, `Ridge`, `Lasso`
Logistic regression	a probability → class	log-loss	`LogisticRegression(C=1.0)`
Baseline (regression)	the training mean	—	`DummyRegressor()`
Baseline (classification)	the majority class	—	`DummyClassifier(strategy="most_frequent")`

Common mistakes

Skipping the baseline (planted bug 1 makes it look worse than it is) · passing a 1-D array as X · reading coefficients of unscaled features as importance · treating logistic regression as a regression model · forgetting `max_iter` and ignoring convergence warnings.

INTERVIEW CORNER

Q. Why is logistic regression called regression? **A.** It regresses the log-odds linearly on the features; the sigmoid turns that into a probability for classification.

Q. What does regularization do? **A.** Penalises large weights so the model can't fit noise; L1 can also zero out features.

Key takeaways

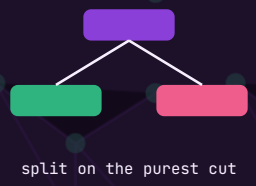
1. Always start with a baseline.
2. Linear regression: numbers; logistic regression: probabilities.
3. Coefficients explain the model — on scaled inputs.

MODULE 2 OF 4

Decision trees

Models made of readable rules, and the overfitting they invite.

- Splits and Gini impurity
- Leaves and predictions
- max_depth and min_samples_leaf
- Underfitting vs overfitting



⏱ 4 hours | Area 04 · Supervised ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1 STEP 1 — Twenty questions with data

≈ In plain words

A tree asks yes/no questions about the features, one after another, until it reaches an answer.

☀ Think of it like

A doctor's checklist: fever? → cough? → rash? → likely diagnosis.

✎ Worked example

1. support_calls ≥ 3? yes → high churn risk
2. no → tenure < 6 months? yes → medium risk
3. no → low risk

Aa The word for it

Decision tree, **split**, **leaf**.

! Why it matters

Trees handle mixed data with no scaling and produce rules people can read.

2 STEP 2 — The best question makes the purest groups

In plain words The tree tries many questions and keeps the one that sorts churners and stayers into the cleanest groups.

Think of it like Sorting laundry: “is it white?” separates the pile better than “does it have buttons?”.

Worked example

group	churners / rows	Gini
before	30 / 100	0.42
calls < 3	10 / 70	0.24
calls ≥ 3	20 / 30	0.44

The word for it **Gini impurity**; the weighted average drops from 0.42 to 0.30.

Why it matters Impurity decrease is how every tree-based model picks its splits.

PAUSE & TRY **A tree scores 100% on training data and 60% on test data. What happened?** Answer

It overfit — memorised training rows. Limit its depth.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Decision tree
A flowchart of yes/no questions on features ending in leaves that hold a prediction.

Split
A question like tenure_months < 6 that divides rows into two groups.

Gini impurity

$1 - \Sigma\, p_k^2$: 0 when a node is pure; 0.5 for a 50/50 binary node.

Information gain

How much a split lowers impurity, weighted by group sizes.

Leaf

A final node; it predicts the majority class (or the class share as probability).

max_depth

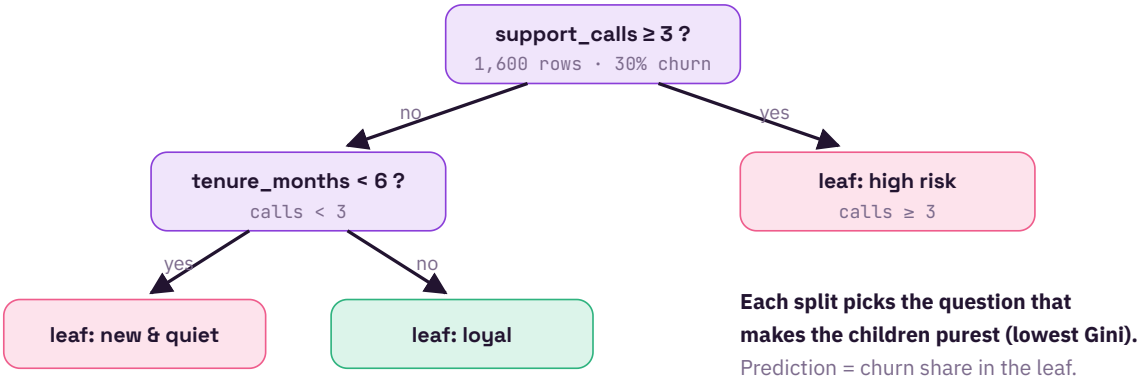
How many questions deep the tree may go — the main overfitting control.

min_samples_leaf

The fewest training rows a leaf may hold; bigger = smoother.

Overfitting

Memorising training noise: great on training data, worse on new data.



Each split picks the question that makes the children purest (lowest Gini).
Prediction = churn share in the leaf.

Trees learn rules you can read. No scaling needed, interactions captured for free — but one deep tree overfits easily.

How a split is chosen

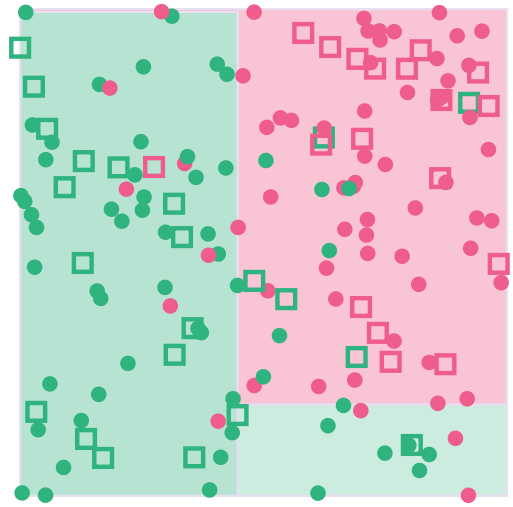
```python
def gini(pos, n):
    p = pos / n
    return 1 - p ** 2 - (1 - p) ** 2

# parent: 100 rows, 30 churners. Candidate split: calls >= 3
left_n, left_pos = 70, 10      # calls < 3
right_n, right_pos = 30, 20    # calls >= 3
parent = gini(30, 100)
child = left_n / 100 * gini(left_pos, left_n) + right_n / 100 * gini(right_pos,
print(f"parent gini {parent:.3f}  after split {child:.3f}  gain {parent - child:
```

```
# output
parent gini 0.420  after split 0.305  gain 0.115
```

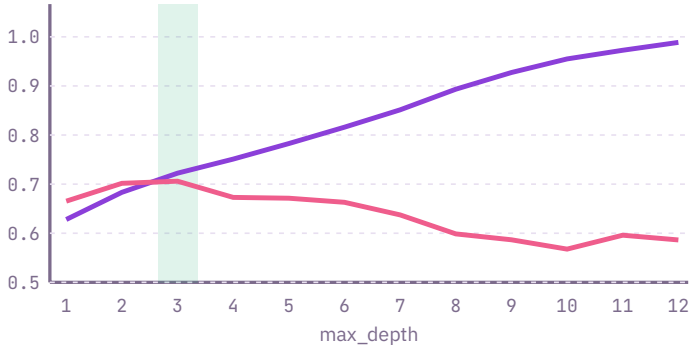
Try it: grow a tree

max_depth **depth 2**



● train □ test
pink = churn, green = stay

```
max_depth = 2 · 4 leaves
train accuracy 87.5%   test accuracy 85.0%   gap 2.5 pts
A reasonable balance between fitting and generalising.
```



train AUC: 0.63 → 0.99

test AUC: best 0.71 at depth 3, 0.59 at 12

Left of the peak: underfitting
(too simple to capture the pattern).

Right of the peak: overfitting
(memorising noise in the training rows).

Real numbers from the lab's churn data.

Depth is a bias–variance dial. On the churn data a single tree peaks at depth 3; deeper trees memorise the training rows.

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.metrics import roc_auc_score
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier, export_text

for depth in (2, 4, None):
    tree = Pipeline([("pre", preprocessor()), ("clf", DecisionTreeClassifier(max
    tr = roc_auc_score(y_tr, tree.predict_proba(X_tr)[:, 1])
    te = roc_auc_score(y_te, tree.predict_proba(X_te)[:, 1])
    print(f"max_depth={str(depth):4}  train AUC {tr:.3f}  test AUC {te:.3f}  lea

small = Pipeline([("pre", preprocessor()), ("clf", DecisionTreeClassifier(max_de
print(export_text(small[-1], feature_names=list(small[0].get_feature_names_out()
```

```
# output
max_depth=2     train AUC 0.683  test AUC 0.702  leaves 4
max_depth=4     train AUC 0.751  test AUC 0.673  leaves 16
max_depth=None  train AUC 1.000  test AUC 0.563  leaves 393
|--- num__support_calls <= -0.13
|   |--- num__tenure_months <= 0.13
|   |   |--- class: 0
|   |--- num__tenure_months >  0.13
|   |   |--- class: 0
|--- num__support_calls >  -0.13
|   |--- num__tenure_months <= -0.48
|   |   |--- class: 1
|   |--- num__tenure_months >  -0.48
|   |   |--- class: 0
```

★ **Watch the train–test gap**

Always print train and test scores side by side. A big gap means overfitting (limit depth, raise min_samples_leaf, add data); both low means underfitting (a richer model or better features). The lab's forest bug is exactly this.

× **Common mistakes**

Leaving `max_depth=None` on small data · judging a tree by its training score · believing deep-tree rules generalise · scaling features for trees (harmless but pointless).

INTERVIEW CORNER

Q. How does a decision tree choose a split? **A.** It tries candidate thresholds on every feature and picks the one with the largest drop in impurity (Gini or entropy).

Q. How do you stop a tree overfitting? **A.** max_depth, min_samples_leaf, min_samples_split, pruning (ccp_alpha) — or use an ensemble.

Key takeaways

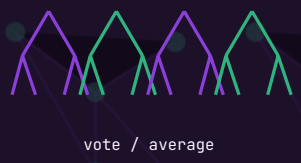
1. Trees split on the purest question, recursively.
2. Depth trades underfitting for overfitting.
3. Compare train vs test to see which side you're on.

MODULE 3 OF 4

Random forests & gradient boosting

Combine many trees into the strongest tabular models.

- Bagging and random forests
- Boosting and learning rate
- Permutation importance
- Choosing a model



⏱ 4 hours | Area 04 · Supervised ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1

STEP 1

Ask a crowd, not one expert

≈ **In plain words** — Train many different trees and average their answers. Individual mistakes cancel out.

✳ **Think of it like** — Guessing the number of sweets in a jar: the crowd's average beats most individuals.

✎ **Worked example**

tree	P(churn)
1	0.8
2	0.5
3	0.6
average	0.63

Aa **The word for it** — **Random forest** (**bagging**).

! **Why it matters** — A robust, low-tuning default for tabular data.

2 STEP 2 — Each new tree fixes the last one's mistakes

≈ **In plain words** — Build small trees one after another; each focuses on what the previous trees still get wrong.

☀ **Think of it like** — Editing a draft in passes: each pass fixes what the last one missed.

✎ **Worked example**

1. tree 1 predicts 0.5 for everyone
2. tree 2 learns the errors: +0.3 for frequent callers
3. sum: 0.5 + 0.1 × 0.3 = 0.53 … repeat 200×

Aa **The word for it** — **Gradient boosting** with a **learning rate**.

! **Why it matters** — Boosting (XGBoost, LightGBM) wins most tabular competitions.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Ensemble
Many models combined so their errors partly cancel.

Bagging
Bootstrap aggregating: train each model on a random resample of rows, then average.

Random forest
Bagged trees that also consider a random subset of features at each split.

Boosting
Train models in sequence, each one focusing on the errors left by the previous ones.

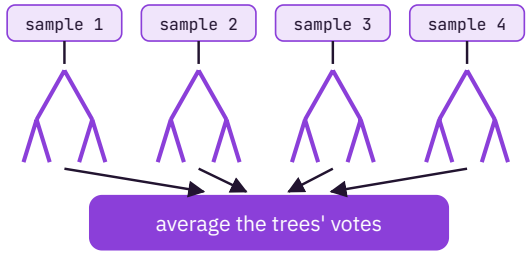
Gradient boosting
Boosting where each new tree fits the gradient of the loss (the residuals for MSE).

Learning rate (boosting)
Shrinks each tree's contribution; smaller needs more trees but generalises better.

Feature importance
How much each feature reduced impurity across the trees (or, better, permutation importance).

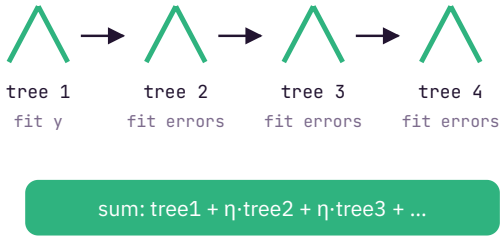
Hyperparameter
A setting you choose before training: n_estimators, max_depth, learning_rate.

Random forest · bagging, in parallel



Reduces variance: deep, noisy trees average out.

Gradient boosting · in sequence



Reduces bias: each tree fixes what's left. XGBoost, LightGBM.

Two ways to combine trees. Forests average independent deep trees; boosting adds shallow trees that correct each other.

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.ensemble import HistGradientBoostingClassifier, RandomForestClassif
from sklearn.metrics import roc_auc_score
from sklearn.pipeline import Pipeline

models = {
    "random forest": RandomForestClassifier(n_estimators=300, max_depth=4, rando
    "grad. boosting": HistGradientBoostingClassifier(max_depth=3, learning_rate=
}
for name, clf in models.items():
    m = Pipeline([("pre", preprocessor()), ("clf", clf)]).fit(X_tr, y_tr)
    tr = roc_auc_score(y_tr, m.predict_proba(X_tr)[:, 1])
    te = roc_auc_score(y_te, m.predict_proba(X_te)[:, 1])
    print(f"{name:15} train AUC {tr:.3f}  test AUC {te:.3f}")
```

```
# output
random forest   train AUC 0.793  test AUC 0.758
grad. boosting  train AUC 0.850  test AUC 0.728
```

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance
from sklearn.pipeline import Pipeline

forest = Pipeline([("pre", preprocessor()), ("clf", RandomForestClassifier(n_est
imp = permutation_importance(forest, X_te, y_te, scoring="roc_auc", n_repeats=10
for col, m in sorted(zip(X_te.columns, imp.importances_mean), key=lambda t: -t[1
    print(f"{col:15} AUC drop when shuffled: {m:.3f}")
```

```
# output
support_calls    AUC drop when shuffled: 0.159
tenure_months    AUC drop when shuffled: 0.049
plan             AUC drop when shuffled: 0.030
monthly_fee      AUC drop when shuffled: 0.007
age              AUC drop when shuffled: -0.002
region           AUC drop when shuffled: -0.005
```

Model	Strength	Watch out for	Key hyperparameters
Logistic regression	fast, explainable, strong baseline	needs scaling; linear boundary	`C`, `penalty`
Decision tree	readable rules, no scaling	overfits	`max_depth`, `min_samples_leaf`
Random forest	robust default, little tuning	large, slower to predict	`n_estimators`, `max_depth`, `max_features`
Gradient boosting	best accuracy on most tables	overfits without early stopping; needs tuning	`learning_rate`, `max_iter`, `max_depth`

i

Simple won here — and that's normal

On this churn data logistic regression ties or beats the ensembles because the true pattern is nearly linear. Gradient boosting usually wins on bigger, messier tables. Always compare against the simple model before shipping the complex one.

×

Common mistakes

Trusting impurity-based `feature_importances_` for high-cardinality features (use permutation importance) · boosting with a high learning rate and many trees (overfits) · comparing models on different splits · tuning on the test set.

INTERVIEW CORNER

Q. Random forest vs gradient boosting? **A.** Forest: parallel deep trees averaged — lowers variance, robust. Boosting: sequential shallow trees fitting residuals — lowers bias, usually more accurate but needs tuning.

Q. Why do random forests sample features at each split? **A.** To decorrelate the trees, so averaging cancels more error.

Key takeaways

1. Forests average; boosting corrects.
2. Permutation importance beats impurity importance.
3. The fanciest model doesn't always win — measure.

MODULE 4 OF 4

The scikit-learn API & pipelines

One consistent way to fit, tune, save and serve any model.

- fit / predict / predict_proba
- Pipelines end to end
- GridSearchCV
- Saving with joblib

```
.fit(X, y)
.predict(X)
.predict_proba(X)
.score(X, y)
```

⏱ 3.5 hours | Area 04 · Supervised ML

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1

STEP 1

Every model has the same buttons

≈ In plain words

In scikit-learn every model is created, then `fit` on training data, then asked to `predict`.

☀ Think of it like

Every car has a key, a pedal and a wheel — learn them once, drive any car.

✎ Worked example

```
from sklearn.tree import DecisionTreeClassifier
m = DecisionTreeClassifier(max_depth=2)
m.fit([[0], [1], [2], [3]], [0, 0, 1, 1])
print(m.predict([[0.4], [2.6]]), m.predict_proba([[2.6]]))
```

```
# output
[0 1] [[0. 1.]]
```

Aa The word for it	**Estimator API**: fit, predict, predict_proba.
! Why it matters	Swapping models becomes a one-line change.

2 STEP 2

Settings you choose vs things it learns

≈ In plain words	You pick settings like tree depth. The model learns the actual splits or weights during fit.
☀ Think of it like	You choose the oven temperature; the cake does the rising.
✎ Worked example	

you set	it learns
max_depth=4	the split questions
C=1.0	coef_ weights

Aa The word for it	**Hyperparameters** vs **parameters**.
! Why it matters	Tuning hyperparameters with cross-validation is how you get the best honest model.

≡

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Estimator

Any scikit-learn object with `fit`; models also have `predict`.

Transformer

An estimator with `fit` and `transform`: scalers, encoders, imputers.

Parameters vs hyperparameters

Parameters are learned in fit (coef_, tree splits); hyperparameters are set by you.

Pipeline

Chains transformers and a model into one estimator.

GridSearchCV

Tries every hyperparameter combination with cross-validation and keeps the best.

RandomizedSearchCV

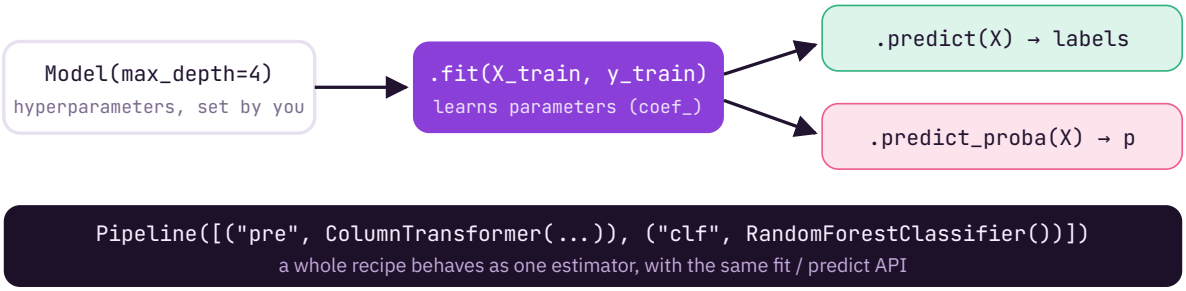
Samples combinations — cheaper for large search spaces.

Persisting a model

Saving the fitted Pipeline with joblib so serving uses the exact same recipe.

random_state

Seed that makes random steps reproducible.



One API for everything. Learn fit / predict / predict_proba once and every model, transformer and pipeline works the same way.

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

pipe = Pipeline([("pre", preprocessor()), ("clf", RandomForestClassifier(n_estim
grid = GridSearchCV(pipe, {"clf__max_depth": [2, 4, 8, None], "clf__min_samples_
                    cv=5, scoring="roc_auc", n_jobs=1)
grid.fit(X_tr, y_tr)
print("best params:", grid.best_params_)
print("best CV AUC:", round(grid.best_score_, 3))
print("test AUC   :", round(grid.score(X_te, y_te), 3))
```

```
# output
best params: {'clf__max_depth': 4, 'clf__min_samples_leaf': 1}
best CV AUC: 0.724
test AUC   : 0.757
```

```
import warnings; warnings.filterwarnings("ignore")
from prep import load_split, preprocessor
X_tr, X_te, y_tr, y_te = load_split()
import joblib
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

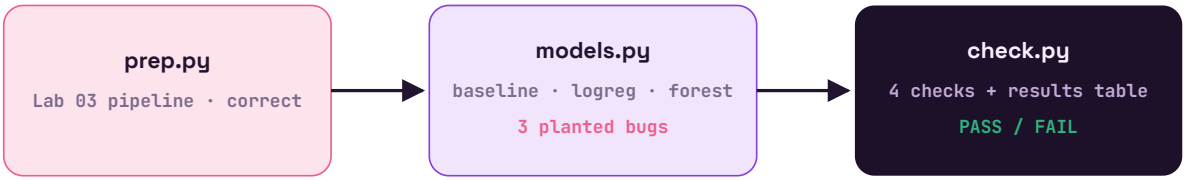
model = Pipeline([("pre", preprocessor()), ("clf", LogisticRegression(max_iter=1
joblib.dump(model, "churn_model.joblib")
loaded = joblib.load("churn_model.joblib")
new_customer = X_te.head(1).assign(region="central")    # unseen region is fine
print(new_customer.to_dict("records")[0])
print("P(churn) =", loaded.predict_proba(new_customer)[0, 1].round(3))
```

```
# output
{'tenure_months': 26.6, 'monthly_fee': 39.48, 'support_calls': 0, 'age': 34.9, '
P(churn) = 0.06
```

Hyperparameter names in a Pipeline

Inside a Pipeline, parameters are addressed as `stepname__param`: `clf__max_depth`, `pre__num__impute__strategy`. A typo raises "Invalid parameter".

The lab: an honest model comparison



Three bugs that make a comparison lie: a broken baseline, AUC from hard labels, and an overfitting forest.

Common mistakes

Calling `fit` on test data · using `predict` where a metric needs scores (AUC, log-loss) · pickling only the model and re-implementing preprocessing in serving · reporting the best CV score as the final test score.

INTERVIEW CORNER

Q. What's the difference between fit, transform and predict? **A.** fit learns from data; transform applies a learned transformation; predict applies a learned model to produce outputs.

Q. How would you tune hyperparameters without leaking? **A.** Cross-validated search on the training data only, with preprocessing inside the Pipeline; touch the test set once at the end.

Key takeaways

1. fit on train; predict / predict_proba on new data.
2. Search hyperparameters with CV on training data.
3. Save and serve the whole Pipeline.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.



70% of customers stay. What accuracy does the most-frequent baseline get?

0.70 — any model must beat it.

Is logistic regression a regression or a classification model?

Classification: it outputs a class probability.

A node has 50 churners out of 100. Its Gini impurity?

$1 - 0.5^2 - 0.5^2 = 0.5$.

Train AUC 1.00, test AUC 0.59. Diagnosis and fix?

Overfitting; limit depth / raise min_samples_leaf, or use an ensemble.

Forest or boosting: which trains trees in parallel?

Random forest (bagging); boosting is sequential.

Why compute AUC from predict_proba, not predict?

AUC ranks by score; 0/1 labels throw away the ranking.

What's the hyperparameter name for max_depth of step “clf” in a Pipeline?

clf__max_depth.

You saved only the model, not the scaler. What breaks in serving?

Inputs aren't preprocessed the same way; save the whole Pipeline.

REFERENCE

Interview questions

Asked in ML engineer and data scientist interviews. Try an answer out loud before opening.



Explain the bias–variance trade-off.

Simple models err systematically (bias); flexible models vary with the sample (variance). Tune complexity to minimise total test error.

When would you choose logistic regression over gradient boosting?

When explainability, speed or small data matter, or the relationship is near-linear; it's also the baseline to beat.

How does a random forest reduce overfitting compared with one tree?

Averaging many decorrelated trees (bootstrap rows + random features) cancels their individual noise.

What is the difference between bagging and boosting?

Bagging trains independent models on resamples in parallel and averages; boosting trains sequentially, each fixing previous errors.

How do you interpret feature importance safely?

Prefer permutation importance on held-out data; beware correlated features sharing importance.

What does C control in LogisticRegression?

Inverse regularization strength: small C = strong penalty, simpler model.

Why might a model with higher AUC have lower accuracy?

AUC measures ranking across all thresholds; accuracy uses one threshold (0.5) and is dominated by the majority class.

How do you make results reproducible?

Fix random_state everywhere, pin library versions, version data and save the fitted Pipeline.

REFERENCE

Glossary

Every term used on this page, A to Z.

A a

AUC
Area under the ROC curve; ranking quality (Area 05).

Bagging
Averaging models trained on bootstrap resamples.

Baseline
Simplest honest model to beat.

Bias
Systematic error from an overly simple model.

Boosting
Sequential models that correct predecessors' errors.

Classification
Predicting a category.

Coefficient
Learned weight of a linear model.

Decision tree
Model of nested yes/no splits.

Ensemble
Combination of models.

Estimator
scikit-learn object with fit.

Gini impurity
$1 - \Sigma p^2$; node impurity.

Gradient boosting
Boosting that fits loss gradients with trees.

GridSearchCV
Exhaustive CV hyperparameter search.

Hyperparameter
Setting chosen before training.

joblib
Library to save and load fitted models.

Leaf
Terminal node of a tree.

Linear regression
$\hat{y} = w \cdot x + b$ by least squares.

Logistic regression
Sigmoid of a linear score; a classifier.

max_depth
Maximum tree depth.

Overfitting
Fitting noise; poor generalization.

Permutation importance
Score drop when a feature is shuffled.

Random forest
Bagged trees with random feature subsets.

Regression
Predicting a number.

Regularization
Penalty on model complexity.

Underfitting
Model too simple to capture the pattern.

Variance
Sensitivity of a model to the training sample.

REFERENCE

FAQ & troubleshooting

The problems learners hit most in this area, and the fix for each.



You see	Why	Fix
Expected 2D array, got 1D array instead	X must be (n_samples, n_features)	`x.reshape(-1, 1)` or pass a DataFrame
ConvergenceWarning: lbfgs failed to converge	Unscaled features or too few iterations	Scale numbers; `max_iter=1000`
Invalid parameter 'max_depth' for estimator Pipeline	Missing the step prefix	Use `clf__max_depth`
Train score ≫ test score	Overfitting	Limit depth, raise min_samples_leaf, regularize, more data
Model predicts only one class	Imbalance + 0.5 threshold, or a broken baseline	Check predict_proba; tune threshold; class_weight="balanced"
NotFittedError	predict called before fit (or on a fresh clone)	Call `fit` first; keep the fitted object

REFERENCE

Further reading

Go deeper when you are ready.



Resource	Why read it
James, Witten, Hastie & Tibshirani, *An Introduction to Statistical Learning (with Python)* (free at statlearning.com)	Linear models, trees, forests and boosting, clearly explained
Géron, *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow* (O'Reilly)	Practical scikit-learn workflows, chapters 2–7
scikit-learn User Guide, “Supervised learning” and “Ensembles” (scikit-learn.org)	Official reference for every estimator used here
Breiman, “Random Forests” (Machine Learning, 2001); Friedman, “Greedy Function Approximation: A Gradient Boosting Machine” (Annals of Statistics, 2001)	The original papers

HANDS-ON LAB · ABOUT 2.5 HOURS

Lab 04 · Churn models: baseline, logistic regression, random forest

`models.py` trains three churn models on the Lab 03 data and compares them. The table it prints is misleading: the baseline looks worse than it is, AUC is computed from the wrong output, and the forest memorises its training rows. `prep.py` is correct.

1. `pip install -r requirements.txt`, then `python check.py`.
2. Check 1: what should a “most frequent class” baseline predict? Fix `make_models`.

3. Check 2: AUC needs a score for each row. Which method returns the churn probability?
4. Check 3: the forest's train AUC is 1.000. Which hyperparameter in the spec limits it?
5. Re-run until all four checks pass and compare accuracy with AUC in the table.

Done when you see `PASS: all 4 checks passed — best model logreg, test AUC = 0.766`.