The Python Data Stack

NumPy · pandas · Matplotlib · scikit-learn

A hands-on hour: from raw CSV to a trained, honestly evaluated model.

Gopichand · code and notebooks at github.com/GC-0719/pydata-intro

By the end of this hour you can…

Load and clean a messy CSV

1

Missing values, duplicates, inconsistent labels — in about ten lines of pandas.

Answer questions with groupby

2

“Which major studies most?” becomes one line of code instead of a pivot table.

Make a chart people can read

3

Five chart types cover most real work — plus three rules that make them legible.

Train and evaluate a model

4

Four lines to fit it, and the discipline to know whether the score is real.

One stack, four layers

scikit-learn

models: fit, predict, evaluate

Matplotlib

pictures of the numbers

pandas

labeled tables: rows, columns, groups

NumPy

fast numeric arrays — the foundation

Each layer is written in terms of the one below it

- A pandas column IS a NumPy array — .values hands it to you.
- Matplotlib plots NumPy arrays, so it plots pandas columns for free.
- scikit-learn takes a DataFrame or an array and returns arrays.
- So the four are really one toolchain. Learn the bottom, the rest follows.

One dataset, all hour: 400 students

400
students

9
columns

48%
actually pass

3
kinds of mess

student_id	major	study_hours	sleep_hours	attendance	prior	tutoring	final	passed
1000	Computer Science	7.3	6.8	0.75	52.0	yes	64.8	0
1001	Business	3.9	7.2	0.72	62.0	no	62.3	0
1002	Psychology	8.1	NaN	0.91	81.0	YES	84.6	1

The question we will answer: given study habits and a prior score, can we predict who passes?

Synthetic but honest: the score really is built from these inputs plus real noise — which is why our model lands at 77%, not 99%.

Follow along (or just watch)

```
git clone <repo-url> && cd pydata-intro
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt

python src/pipeline.py        # the whole project
jupyter notebook              # notebooks/01 .. 05
```

Nothing to install?

Open the notebooks in Google Colab or VS Code — every dependency is in requirements.txt, and the data ships with the repo.

`notebooks/01–04`	one library each, with exercises and solutions
`notebooks/05`	the whole project, narrated
`src/pipeline.py`	the same project as a script you can run

1

NumPy

Arrays: the layer everything else stands on

Why not just use a Python list?

Python list

```
squares = [x * x for x in data]
# 1,000,000 items -> ~60 ms
```

NumPy array

```
squares = data ** 2
# same million items -> ~1 ms
```

~50x

faster, and shorter to write

- A list holds pointers to boxed objects scattered across memory.
- An array is one contiguous block of same-typed numbers.
- The loop runs in compiled C, not in the Python interpreter.
- One dtype per array — that constraint is what buys the speed.

The ndarray, and its four questions

```
import numpy as np

b = np.array([[1, 2, 3],
              [4, 5, 6]])
```

`.shape`	`(2, 3)`	rows and columns
`.ndim`	`2`	how many dimensions
`.dtype`	`int64`	the one type all elements share
`.size`	`6`	total number of elements

Making arrays

```
np.array([1, 2, 3])     # from a list
np.arange(0, 10, 2)     # 0 2 4 6 8
np.linspace(0, 1, 5)    # 5 even points
np.zeros((3, 4))        # filled with 0
np.ones(5) / np.eye(3)  # 1s / identity

rng = np.random.default_rng(0)
rng.normal(size=4)      # reproducible

arr.reshape(2, 5)  arr.ravel()  arr.T
```

Reshaping never copies the data — it re-labels how the same block of memory is read.

Indexing: same rules as lists, one set per dimension

```
m = np.arange(1, 13).reshape(3, 4)

m[1, 2]        # 7      one element
m[0]           # row 0
m[:, 1]        # column 1
m[0:2, 1:3]    # a sub-block
m[::2]         # every other row
```

`m[0:2, 1:3]`

- Slices are views, not copies — writing to one changes the original.
- Use .copy() when you want a separate array.

Boolean masking — the idea to actually remember

```
scores = np.array([88, 42, 67, 95, 71, 55, 79])

mask = scores >= 70          # [True False False True True False True]
scores[mask]                 # [88 95 71 79]   keep the True ones
mask.sum()                   # 4               True counts as 1
mask.mean()                  # 0.57            the pass rate, free

scores[(scores >= 60) & (scores <= 90)]   # & and, | or, ~ not
```

No loop, no if

You describe the condition; NumPy applies it to every element and hands back the matches.

Parentheses are not optional

Python's operator precedence puts & above >=. Forget the parentheses and you get a confusing error, every time.

NUMPY

Vectorization: write the formula, not the loop

Before

```
fahrenheit = []
for c in celsius:
    fahrenheit.append(c * 9 / 5 + 32)
```

After

```
fahrenheit = celsius * 9 / 5 + 32
```

- `+ - * / **` work element-by-element on whole arrays.
- np.sqrt, np.log, np.exp, np.sin … apply to every element at once.
- `@` is matrix multiplication; `*` is element-wise. They are not the same.
- A scalar applies to everything: `arr * 3`, `arr + 10`.

Aggregations, and what axis means

```
data = np.array([[10, 20, 30],
                 [40, 50, 60]])

data.sum()          # 210   everything
data.sum(axis=0)    # [50 70 90]     one value per column
data.sum(axis=1)    # [60 150]       one value per row
```

The memory hook

axis=0 collapses the rows (you move down a column).

axis=1 collapses the columns (you move across a row).

`.mean()` `.std()` `.min()` `.max()` `.argmax()` `np.median()` `np.percentile()`

All of them take axis. argmax gives you the position of the largest value — useful when you need which, not just how much.

Broadcasting: small arrays stretch to fit big ones

```
prices = np.array([[100., 200., 300.],      # store A
                   [110., 190., 320.]])     # store B
tax    = np.array([0.05, 0.07, 0.09])   # per product

prices * (1 + tax)      # (2,3) with (3,)  ->  (2,3)

# the standardization every ML pipeline does:
(X - X.mean(axis=0)) / X.std(axis=0)
```

The rule

Compare shapes right to left. Dimensions match if they are equal, or if one of them is 1.

No data is copied — NumPy just walks the small array again.

Broadcasting is why you will almost never write a nested loop again.

NumPy in one slide

Task	Code
Create	np.array · np.arange · np.linspace · np.zeros · np.ones
Inspect	.shape · .dtype · .ndim · .size
Reshape	.reshape(r, c) · .ravel() · .T
Filter	arr[arr > 5] · np.where(cond, a, b)
Aggregate	.sum() · .mean() · .std() · .argmax() — all take axis=
Math	+ - * / ** · np.sqrt · np.log · @ for dot product
Random	rng = np.random.default_rng(42) · rng.normal · rng.integers

Notebook: notebooks/01_numpy.ipynb — six sections, three exercises, solutions at the bottom.

2

pandas

Spreadsheets you can script

PANDAS

What pandas adds on top of NumPy

Series — one labeled column

A NumPy array plus an index. Homogeneous type, like the array underneath it.

```
s = pd.Series([10, 20, 30],
              index=['a','b','c'])
```

DataFrame — a table

A dict of Series sharing one index. Each column can hold a different type: numbers, text, dates.

```
df = pd.DataFrame({'name': [...],
                   'score': [...]})
```

- Labels instead of positions: df["final_score"] beats arr[:, 7].
- Mixed types in one table, which NumPy alone cannot do.
- Missing data is a first-class citizen (NaN), not a crash.
- Reads and writes CSV, Excel, JSON, SQL and Parquet out of the box.

PANDAS

The first four commands on any new dataset

`df.shape`	How big is it? (rows, columns)
`df.info()`	Column names, types, and how many values are missing
`df.describe()`	Min, max, mean, quartiles for every numeric column
`df["col"].value_counts()`	What categories exist, and how often

Run all four before you write a single line of analysis. Two of them found the defects in our file.

Selecting rows and columns

```
df["final_score"]              # one column   -> Series
df[["major", "final_score"]]   # several      -> DataFrame

df.loc[0:2, ["major", "final_score"]]   # by LABEL
df.iloc[0:3, [1, 7]]                    # by POSITION
```

[ ]

Columns by name. Double brackets when you want more than one — the inner list is the list of columns.

.loc

Label-based, and the end of the range is INCLUDED. Takes a boolean mask too.

.iloc

Position-based, and the end is EXCLUDED, exactly like Python slicing.

Filtering: boolean masks, again

```
df[df["final_score"] >= 85]                          # one condition

df[(df["major"] == "Computer Science")               # combine with & | ~
   & (df["study_hours"] > 8)]                        # parentheses required

df[df["major"].isin(["Biology", "Psychology"])]   # any of these
df.query("final_score > 80 and tutoring == 'yes'") # reads like English
```

It's the same idea as NumPy

A condition produces True/False per row; the DataFrame keeps the True ones.

Chained assignment warning?

Filter, then .copy() before you modify. Editing a slice of a DataFrame is the classic pandas footgun.

Cleaning: three defects hiding in this file

3

duplicate rows

`df.drop_duplicates()`

19

missing values

`df[c].fillna(df[c].median())`

12

“YES” vs “yes”

`df[c].str.strip().str.lower()`

```
clean = (df.drop_duplicates()
           .assign(tutoring=df["tutoring"].str.strip().str.lower()))
for col in ["sleep_hours", "attendance_rate"]:
    clean[col] = clean[col].fillna(clean[col].median())
```

Cleaning is most of the job. Nobody's data arrives tidy.

groupby: split → apply → combine

SPLIT
by major

→

APPLY
mean of final_score

→

COMBINE
one row per major

```
df.groupby("major")["final_score"].mean()

df.groupby("major").agg(
    students=("student_id", "count"),
    avg_score=("final_score", "mean"),
    pass_rate=("passed", "mean"),
)
```

major	students	avg_score	pass_rate
Computer Science	151	70.38	0.52
Psychology	78	68.96	0.45
Business	88	68.77	0.47
Biology	83	67.62	0.46

Most questions you have about a table are a groupby.

pandas in one slide

Task	Code
Load / save	pd.read_csv(path) · df.to_csv(path, index=False)
Look	.head() · .info() · .describe() · .value_counts() · .shape
Select	df["col"] · df[["a","b"]] · .loc[rows, cols] · .iloc[i, j]
Filter	df[df.col > 5] · .isin([…]) · .query("col > 5")
Clean	.isna() · .fillna() · .dropna() · .drop_duplicates() · .astype()
Derive	df["new"] = … · np.where · pd.cut · .apply()
Summarize	.groupby("k")["v"].mean() · .agg({…}) · pd.pivot_table
Combine	.merge(other, on="key") · pd.concat([a, b])

Notebook: notebooks/02_pandas.ipynb

3

Matplotlib

Making the numbers visible

MATPLOTLIB

Always start with fig, ax = plt.subplots()

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(x, y, label="sin(x)")
ax.set_title("Anatomy of a figure")
ax.set_xlabel("x"); ax.set_ylabel("value")
ax.legend()
fig.savefig("plot.png", dpi=160, bbox_inches="tight")
```

fig vs ax

fig is the whole canvas.
ax is one set of axes drawn on it.

Everything you want to control is a method on ax — which is why this style scales to multi-panel figures and plt.plot() does not.

`ax.set_title / set_xlabel / set_ylabel`	words
`ax.set_xlim / set_ylim / set_xticks`	scale and ticks
`ax.legend() · ax.annotate()`	identity and callouts
`plt.subplots(2, 2) → axes[0, 1]`	a grid of panels

Five charts cover most real work

Line

change over an ordered axis

```
ax.plot(x, y)
```

Bar

compare categories

```
ax.bar / ax.barh
```

Histogram

the shape of one variable

```
ax.hist(v, bins=25)
```

Scatter

relationship between two

```
ax.scatter(x, y)
```

Box

compare whole distributions

```
ax.boxplot([...])
```

Grid

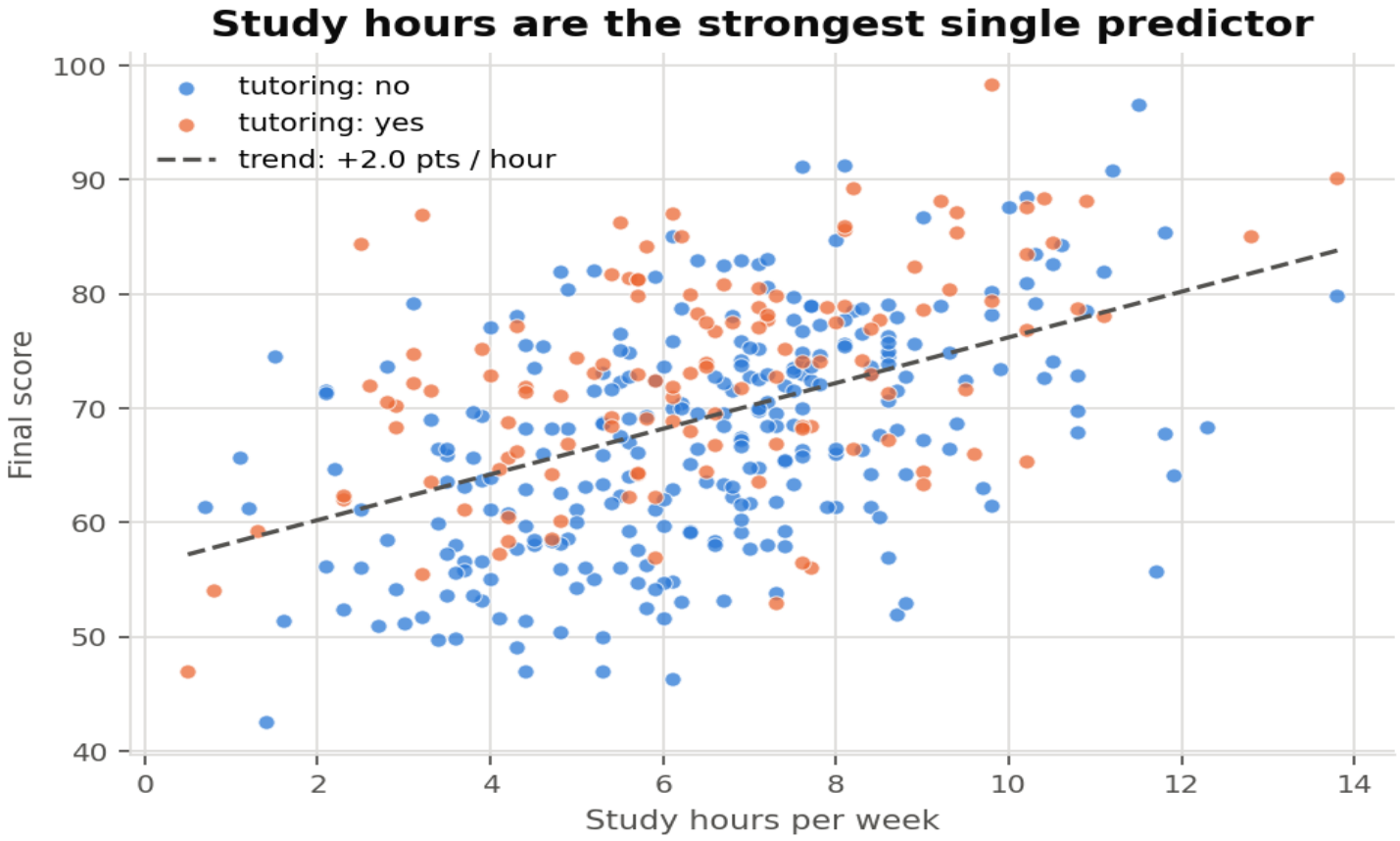
several views at once

```
plt.subplots(2, 2)
```

Pie charts are almost never the right answer — people compare angles badly. Use a sorted bar chart.

MATPLOTLIB

Same table, now you can see the answer



What the picture says

- The relationship is real but noisy — a cloud, not a line.
- About +2 points per extra study hour.
- Tutored students (orange) sit slightly higher at the same hours.
- Plenty of counter-examples. Any model here will be wrong sometimes — and that is the honest result.

Produced by src/plots.py — 12 lines.

Three rules for charts people can read

1

Label everything

Title, x-label, y-label, units. If a screenshot leaves the room, it must still explain itself.

2

Sort, and start at zero

Alphabetical order hides the story. A truncated bar axis exaggerates differences — that's not a style choice, it's an error.

3

One idea per chart

Never two y-axes. If you need two scales, make two charts. Add a legend when there is more than one series.

Bonus: pick colors that survive colorblindness and greyscale printing — and never let color be the only cue.

4

scikit-learn

Letting the model find the pattern

The vocabulary, in one picture

study	sleep	attend	prior	passed

X — features (what the model sees)

y — target

Classification

the target is a category — passed / failed, spam / not spam

Regression

the target is a number — the exact final score, a price

Fit

learn the pattern from training rows

Predict

apply it to rows the model has never seen

SCIKIT-LEARN

One API for 200+ models

```
model = SomeModel(hyperparameters)     # 1. choose
model.fit(X_train, y_train)            # 2. learn
predictions = model.predict(X_test)    # 3. apply
score = model.score(X_test, y_test)    # 4. evaluate
```

Learn it once

Swapping a decision tree for a random forest, or logistic regression for a neural network, is a one-line change. The surrounding code never moves.

Hyperparameters vs parameters

Hyperparameters are what YOU choose (n_neighbors, max_depth). Parameters are what the model learns during fit. Only the first kind goes in the constructor.

SCIKIT-LEARN

Hold out a test set. Always.


```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.25,      # how much to hold back
    random_state=42,     # reproducible split
    stratify=y,          # keep the class balance
)
```

Why it is non-negotiable

Score a model on rows it memorised and you measure memory, not learning. The test set is the only honest estimate of performance on tomorrow's data.

Split first. Everything else — scaling, imputing, feature selection — happens after, and only on the training half.

SCIKIT-LEARN

Pipelines stop data leakage

Leaky

```
scaler = StandardScaler()
X_all = scaler.fit_transform(X)  # sees the test rows
X_tr, X_te = train_test_split(X_all, ...)
```

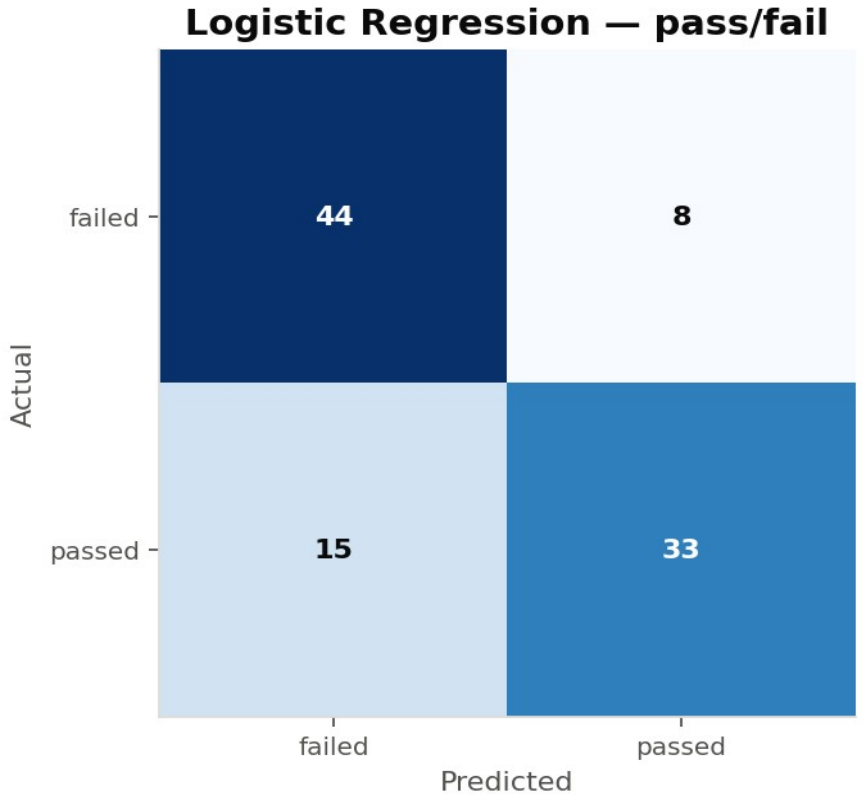
Correct

```
model = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=1000))
model.fit(X_train, y_train)  # scaler fits on train
```

- A Pipeline bundles preprocessing with the model, so every re-fit — including each cross-validation fold — re-fits the preprocessing correctly.
- ColumnTransformer applies different treatment per column: scale the numbers, one-hot encode the text.
- Scaling matters for distance and linear models (kNN, logistic regression, SVM); trees don't care.
- The whole pipeline saves as one object — no “remember to scale it the same way” note in production.

SCIKIT-LEARN

Accuracy alone will mislead you



Precision

Of everything I labelled “pass”, how much really passed? — the cost of false alarms.

Recall

Of everyone who really passed, how many did I catch? — the cost of misses.

F1

One number balancing the two, when you need a single score.

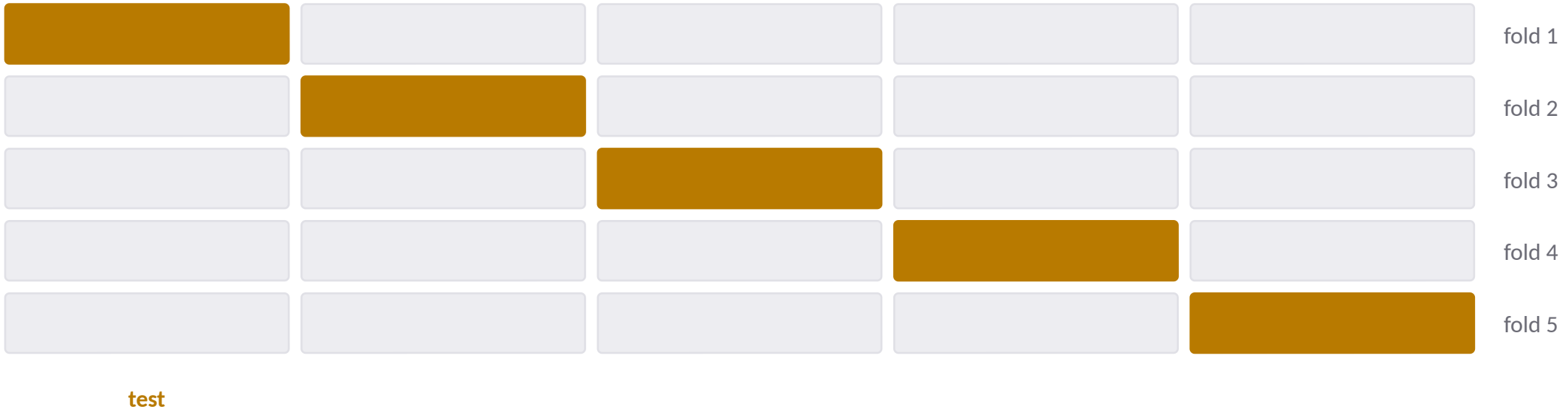
Support

How many test rows each class had. Small support = noisy metric.

A fraud detector that always says “not fraud” scores 99.9% accuracy and catches nothing.

SCIKIT-LEARN

One split can lie: cross-validate

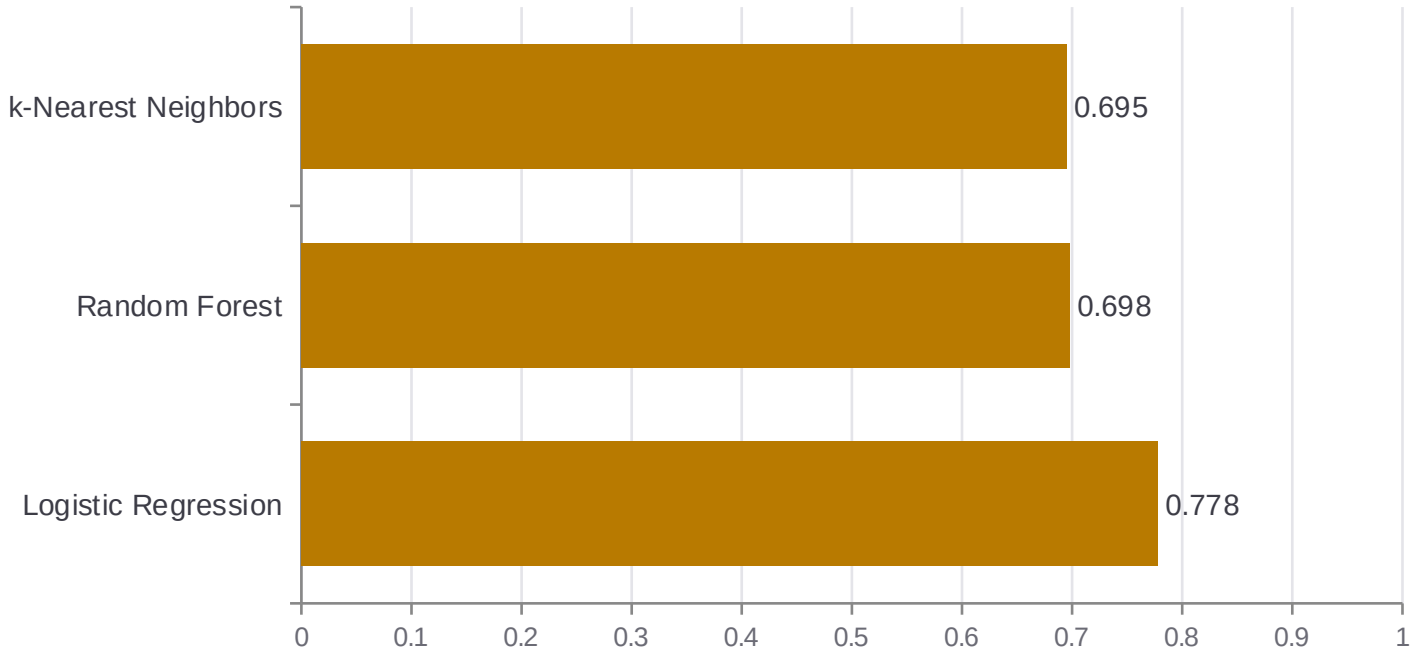



```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=cv, scoring="accuracy")
print(f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

Report the mean and the spread. A model at 0.78 ± 0.01 is a different animal from 0.78 ± 0.12.

SCIKIT-LEARN

Which model wins on our data?



Read it carefully

Guessing the majority class scores 0.52 here — that is the bar to beat.

The simplest model wins. On a small, mostly-linear dataset that is the normal outcome, not a disappointment.

Start simple. Reach for a bigger model only when a simple one has demonstrably run out of room.

Always compare against a baseline: majority class for classification, the mean for regression.

Five mistakes to not make

1. Evaluating on the data you trained on — that measures memory, not learning.
2. Scaling before splitting: the test set leaks into training. Use a Pipeline.
3. Judging on accuracy alone, especially with imbalanced classes.
4. Trusting one train/test split instead of cross-validating.
5. Tuning hyperparameters against the test set — then it isn't a test set any more.

All five have the same shape: the model got to see something it shouldn't have.

All four, one command

```
$ python src/pipeline.py

STEP 1 - load and clean            [pandas]
  raw rows: 403   duplicates: 3   missing cells: 19
  clean rows: 400  missing cells: 0
STEP 2 - explore                   [pandas + numpy]
  study_hours 0.474 | prior_score 0.436 | attendance 0.100
STEP 3 - visualize                 [matplotlib]     6 figures -> outputs/
STEP 4 - compare models            [scikit-learn]   5-fold CV
  Logistic Regression 0.778 | Random Forest 0.698 | kNN 0.695
STEP 5 - evaluate the winner       test accuracy 0.770
STEP 6 - regression baseline       MAE 6.63 pts, R2 0.470
STEP 7 - save artifacts            model + metrics.json
```

Every step labelled with the library doing the work.

What the project found

77%

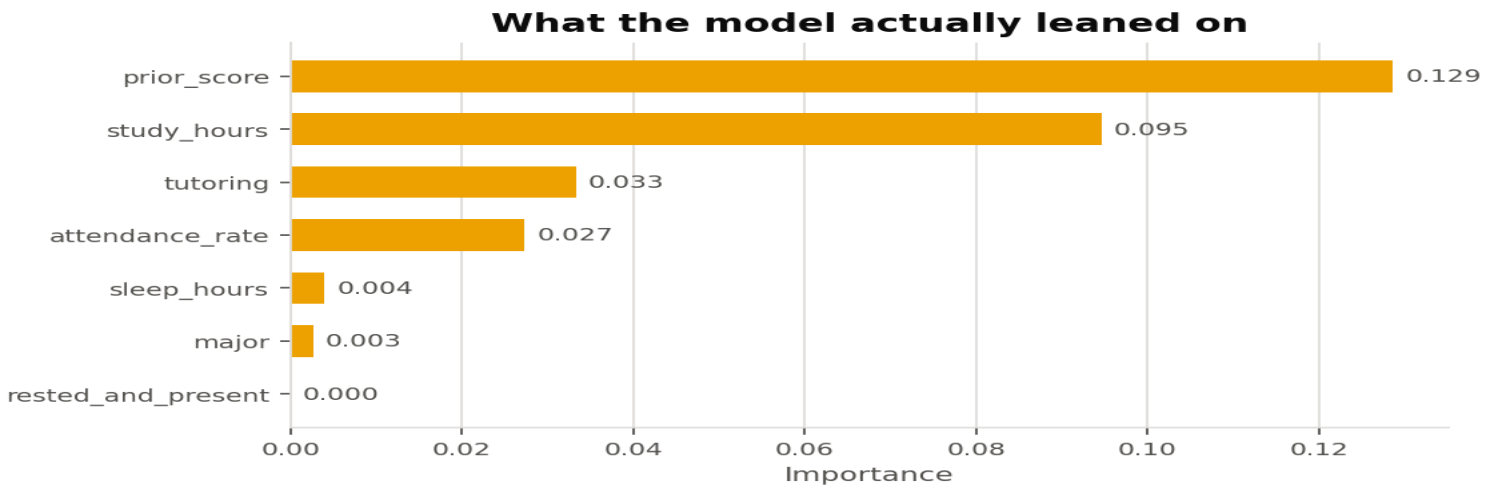
held-out accuracy
(baseline: 52%)

6.6

points average error
when predicting the score

0.47

R^2 — half the variance
is just noise



The honest read

Prior score and study hours carry the signal; major barely matters.

77% is a real result on noisy human behaviour. A 99% here would mean we leaked the answer into the features.

Where to go next

This week

Finish the exercises in notebooks 01–05, then re-run the project with your own CSV. Changing the dataset is where it clicks.

Next

seaborn for statistical plots in one line · GridSearchCV for tuning · scikit-learn's own user guide, which is unusually good.

Then

Kaggle Learn's micro-courses, or a small end-to-end project you actually care about. Depth beats breadth.

The four cheat-sheet slides and every notebook are in the repo — nothing here needs to be copied down.

```
pip install numpy pandas matplotlib scikit-learn jupyter
```

Questions

Notebooks, slides, data and the full project:

`github.com/GC-0719/pydata-intro`