scikit-learn

Models that learn from data: fit, predict, penalise, cross-validate, score

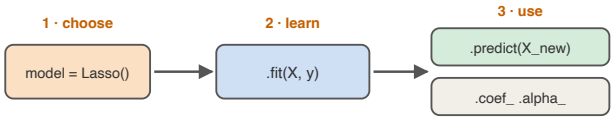
Python cheatsheet · MATH60033A · Quantitative Methods in IB · used in sessions 5 to 8, 10 and 13

```
from sklearn.linear_model import LassoCV
```

import each tool from its module: linear_model, preprocessing, …

1 · One pattern for every model

Every estimator works the same way: create it, fit it, use it.



X: a table of predictors (rows × columns) · y: the outcome, one value per row

`model.fit(X, y)`
learn from the predictors X (a table) and the outcome y (one column).

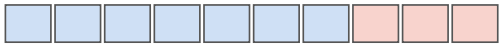
`model.predict(X_new)`
the model's prediction for new rows.

`model.coef_ · model.intercept_`
the fitted coefficients — scikit-learn's spelling of statsmodels' `.params`.

`model.score(X, y)`
R^2 for a regression, accuracy for a classifier.

> scikit-learn gives coefficients, not p-values or standard errors. For inference, use statsmodels.

2 · Hold something back



X_tr, y_tr — 70%: the model learns here X_te, y_te — 30%

scored once, at the very end

`train_test_split(X, y, test_size=0.3, random_state=7)`
cut the rows once: learn on 70%, judge on 30% the model never saw.

```
Xtr, Xte, ytr, yte = train_test_split(X, y,
test_size=0.3, random_state=7)
```

`random_state=7`
the same cut every run, so your numbers reproduce.

3 · Same scale for every predictor

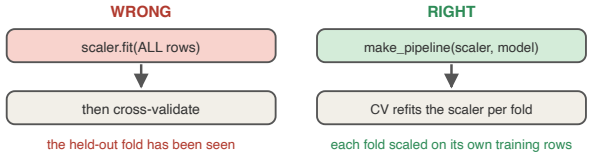
A penalty compares coefficient sizes, so units must not decide them.

`StandardScaler()`
rescales each column to mean 0, standard deviation 1.

`make_pipeline(StandardScaler(), model)`
chains the steps: fit scales then fits; predict scales then predicts.

```
pipe = make_pipeline(StandardScaler(),
LassoCV(cv=5)).fit(Xtr, ytr)
```

`pipe[-1]`
the last step — the model itself: `pipe[-1].coef_`, `pipe[-1].alpha_`.



> Standardise **inside** the pipeline. Scaling all rows first lets the test rows leak into training.

4 · Ridge, lasso, elastic net

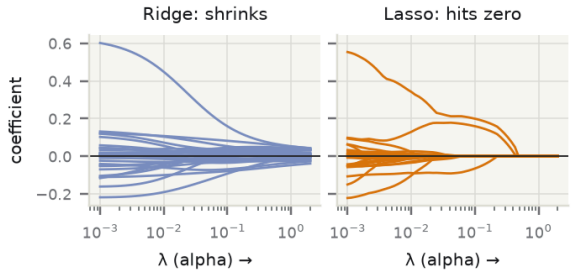
*OLS plus a price on big coefficients. In scikit-learn, λ is called **alpha**.*

`LinearRegression()`
plain OLS: the benchmark, no penalty.

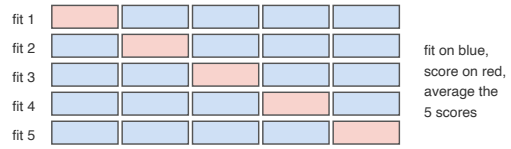
`Ridge(alpha=10)`
shrinks every coefficient, keeps them all.

`Lasso(alpha=0.8)`
shrinks, and sets the weak ones to exactly 0: it selects.

`ElasticNet(alpha=1, l1_ratio=0.5)`
a mix: `l1_ratio` 1 is the lasso, 0 is ridge. Keeps correlated predictors together.



5 · Let cross-validation choose λ

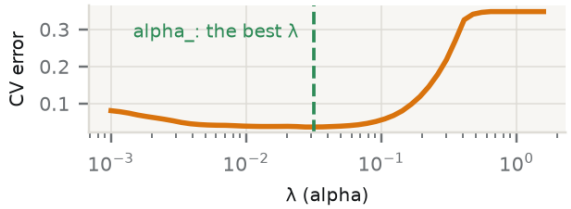


`RidgeCV(alphas=np.logspace(-3, 3, 60))`
tries 60 values of λ, keeps the best.

`LassoCV(cv=5, random_state=0, max_iter=50_000)`
5-fold CV over its own grid of λ.

`ElasticNetCV(l1_ratio=[.1, .5, .9, 1], cv=5)`
chooses λ **and** the mix.

`.alpha_` · `.l1_ratio_` · `.mse_path_`
the chosen λ, the chosen mix, the CV error at every λ.



6 · Score on rows the model never saw

`mean_squared_error(yte, pipe.predict(Xte)) ** 0.5`
RMSE: the typical miss, in the units of y.

`cross_val_score(model, X, y, cv=5, scoring=...)`
five fits, five scores: their mean is the CV estimate.

```
-cross_val_score(m, X, y, cv=5,
scoring="neg_root_mean_squared_error").mean()
```

`KFold(n_splits=5, shuffle=True, random_state=7)`
the folds themselves, to pass as `cv=`.

> Choose λ (or k) by cross-validation on the training rows. Touch the test set **once**.

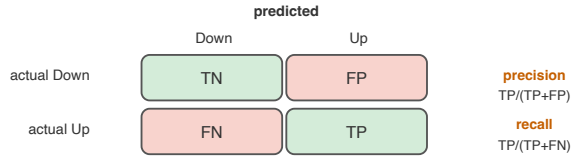
7 · Classify: neighbours and a grid (session 8)

`KNeighborsClassifier(n_neighbors=5)`
the class most common among the k closest training rows. `KNeighborsRegressor` averages them.

`GridSearchCV(pipe, {"kneighborsclassifier__n_neighbors": range(1, 26)}, cv=5)`
tries every k by CV; step name, two underscores, parameter.

`.best_params_` · `.best_score_`
the chosen k and its CV accuracy.

`confusion_matrix(y_te, y_hat)` · `classification_report(...)`
right and wrong, by class; precision and recall.



8 · Also in the course

`NearestNeighbors(n_neighbors=1).fit(X)`
session 10: find each treated unit's closest control, for matching.

`PCA().fit(Z)` · `.explained_variance_ratio_`
session 13: components of standardised columns, and the variance each explains.

9 · Read what the penalty kept

`pipe[-1].coef_`
one coefficient per column of X, on the **standardised** scale: compare sizes, not units.

`kept = [c for c, b in zip(XCOLS, coef) if b != 0]`
the predictors the lasso or elastic net kept.

```
print(f"lasso kept {len(kept)} of {len(XCOLS)}")
```

> A zero means "not needed for prediction here", never "no effect".

10 · The whole workflow, in one cell

As in sessions 5 and 6: four models, scored on held-out rows.

`def test_rmse(model): …`
fit on 70% inside a pipeline, return (kept, RMSE on 30%).

```
pipe = make_pipeline(StandardScaler(),
model).fit(Xtr, ytr)
kept = int(np.sum(np.abs(pipe[-1].coef_) > 1e-8))
rmse = mean_squared_error(yte, pipe.predict(Xte)) **
0.5
```

`results = {"ridge": test_rmse(RidgeCV(...)), …}`
one entry per model; then a table.

```
pd.DataFrame(results, index=["kept", "test RMSE"]).T
```

11 · When scikit-learn complains

`ConvergenceWarning`
the lasso stopped before settling: raise `max_iter=50_000`.

`ValueError: Input X contains NaN`
drop or fill missing values first; scikit-learn will not.

`Expected 2D array, got 1D`
X must be a table: `d[["x"]]` (double brackets), not `d["x"]`.