COMP2200/COMP6200 — Week 9
Supervised Learning Pipelines

Greg Baker

4th May 2026

This week's Python is the hairiest we'll get

- If you've not programmed in Python much, this will be a tough lecture – there's a lot here.
- The point is to get to the inference code at the end (which is very simple).

Competency checklist (you should be able to. . .)

1. Load a dataset with `pandas` and identify target vs features.
2. Split data properly (`train_test_split` with `stratify` for classification).
3. Distinguish numeric vs categorical columns.
4. Build a `ColumnTransformer` for preprocessing (impute + scale + one-hot encode).
5. Wrap preprocessing + model in a single `Pipeline`.
6. Evaluate with cross-validation (`cross_val_score`/`cross_validate`) and correct metrics.
7. Tune simple hyperparameters with `GridSearchCV` or `RandomizedSearchCV`.
8. Avoid common leakage traps.
9. Save and reload a fitted pipeline with `joblib`.

Learning to drive safely

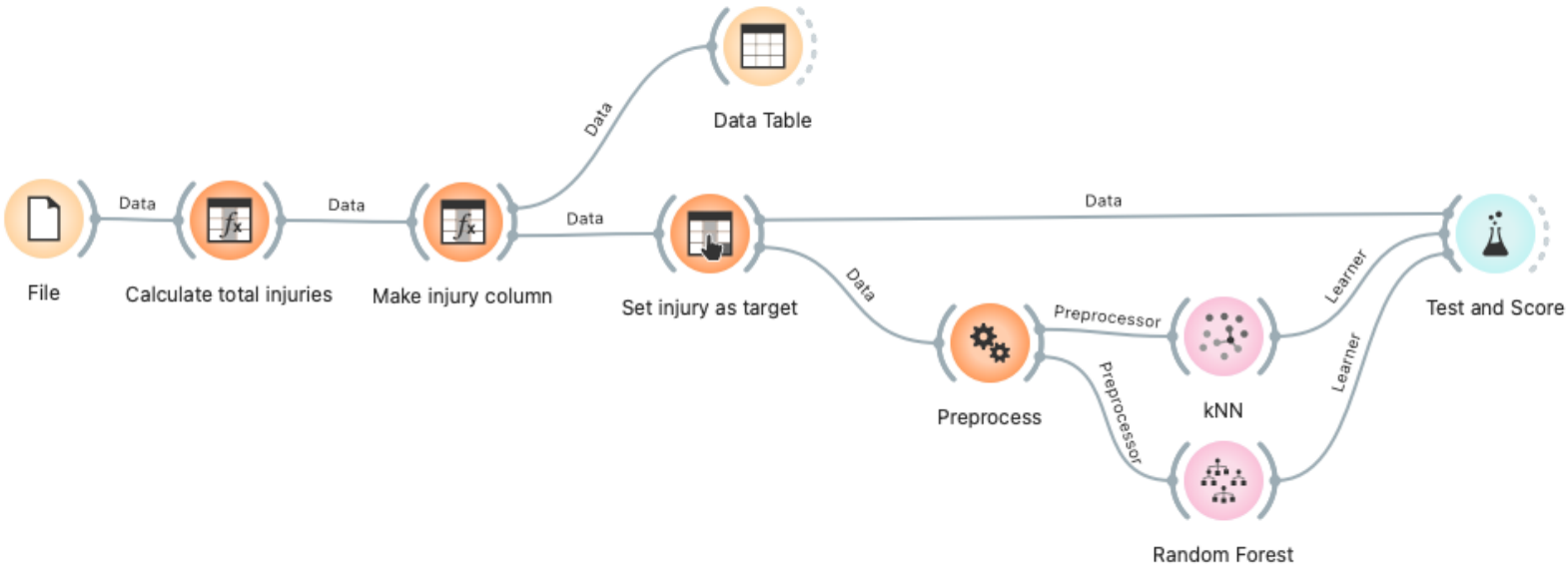
NSW recorded crashes from 2019–2023. Each row describes a crash and whether anyone was injured.
Our goal: predict whether a crash leads to any injury so we can help people drive more safely.

Load the data

```
import pandas as pd
raw = pd.read_excel("nsw_road_crash_data_2019-2023_crash.xlsx")
```

- Read Excel file into DataFrame `raw`.

Orange version



Look at column names

```
>>> raw.columns
Index(['Crash ID', 'Degree of crash', 'Degree of crash -
    detailed',
       'Reporting year', 'Year of crash', 'Month of crash',
       'Day of week of crash', 'Two-hour intervals', 'Street of
    crash',
       'Street type', 'Distance', 'Direction', 'Identifying
    feature',
       'Identifying feature type', 'Town', 'Route no.', 'School
    zone location',
       'School zone active', 'Type of location', 'Latitude',
    'Longitude',
       'LGA', 'Urbanisation', 'Conurbation 1', 'Alignment',
       'Primary permanent feature', 'Primary temporary feature',
       'Primary hazardous feature', 'Street lighting', 'Road
    surface',
       'Surface condition', 'Weather', 'Natural lighting',
    'Signals operation',
       'Other traffic control', 'Speed limit',
       'Road classification (admin)', 'RUM - code', 'RUM -
    description',
       'DCA - code', 'DCA - description', 'DCA supplement',
    'First impact type',
       'Key TU type', 'Other TU type', 'No. of traffic units
    involved',
       'No. killed', 'No. seriously injured', 'No. moderately
    injured',
       'No. minor-other injured'],
      dtype='object')
```

Summary statistics

```
>>> raw.describe()
           Crash ID  Reporting year  Year of crash
    Distance    Route no.
count  1.000000e+03          1000.0         1000.0
    1000.000000   643.000000
mean   1.191868e+06          2019.0         2019.0
    2552.032000  1641.996890
std    9.120605e+02             0.0            0.0
    44802.570032  2706.033652
min    1.189190e+06          2019.0         2019.0
    0.000000     1.000000
25%    1.191198e+06          2019.0         2019.0
    0.000000    14.000000
50%    1.191979e+06          2019.0         2019.0
    0.000000   200.000000
75%    1.192652e+06          2019.0         2019.0
    150.000000   676.500000
max    1.193206e+06          2019.0         2019.0
    999999.000000  7770.000000
```

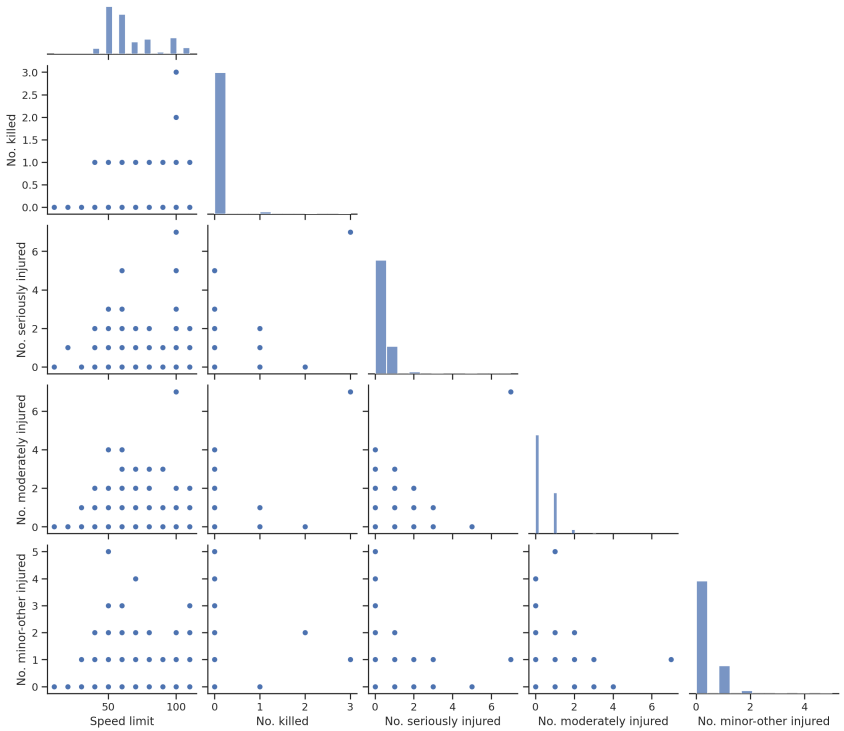
- Quick numeric overview.

Data shape and axes

```
>>> raw.shape
(1000, 49)
```

- `(rows, columns)` — axis 0 is rows, axis 1 is columns.

Pairplot


No. killed
No. seriously injured
No. moderately injured
No. minor-other injured
Speed limit
No. killed
No. seriously injured
No. moderately injured
No. minor-other injured

Think–pair–share: feature selection

- Crash ID
- Degree of crash
- Degree of crash - detailed
- Reporting year
- Year of crash
- Month of crash
- Day of week of crash
- Two-hour intervals
- Street of crash
- Street type
- Distance
- Direction
- Identifying feature
- Identifying feature type
- Town
- Route no.
- School zone location
- School zone active
- Type of location
- Latitude
- Longitude
- LGA
- Urbanisation
- Conurbation 1
- Alignment
- Primary permanent feature
- Primary temporary feature
- Primary hazardous feature
- Street lighting
- Road surface
- Surface condition
- Weather
- Natural lighting
- Signals operation
- Other traffic control
- Speed limit
- Road classification (admin)
- RUM - code
- RUM - description
- DCA - code
- DCA - description
- DCA supplement
- First impact type
- Key TU type
- Other TU type
- No. of traffic units involved
- No. killed
- No. seriously injured
- No. moderately injured
- No. minor-other injured

Targets and features

- We call the thing we want to predict `y`.
- `X` holds the columns we use as inputs.

What is `raw[injury_cols]`?

```
injury_cols = ["No. killed", "No. seriously injured",
               "No. moderately injured", "No. minor-other
    injured"]
raw[injury_cols]
```

- Subset of `raw` containing only injury counts.

What is `raw[injury_cols].sum(axis=1)`?

```
raw[injury_cols].sum(axis=1)
```

- Row-wise injury totals (axis 1 = across columns).

What is `raw[injury_cols].sum(axis=0)`?

```
raw[injury_cols].sum(axis=0)
```

- Column-wise sums (axis 0 = down rows).

Create target y

```
injury_total = raw[injury_cols].sum(axis=1)
y = (injury_total > 0).astype(int)
```

- y is 1 if any injuries occurred, else 0.

Checkpoint

- What does `(raw[injury_cols].sum(axis=1) > 0).astype(int)` compute?

Create feature matrix `X`

```
X = raw.drop(columns=injury_cols)
```

- Remove injury columns; everything else becomes features.

Inspect the data

```
print("X shape:", X.shape)
print("y shape:", y.shape)
print(X.head())
```

- DataFrame shape is `(rows, columns)`; Series shape is `(rows,)`.
- Inspect first rows to spot issues.

Common scikit-learn imports

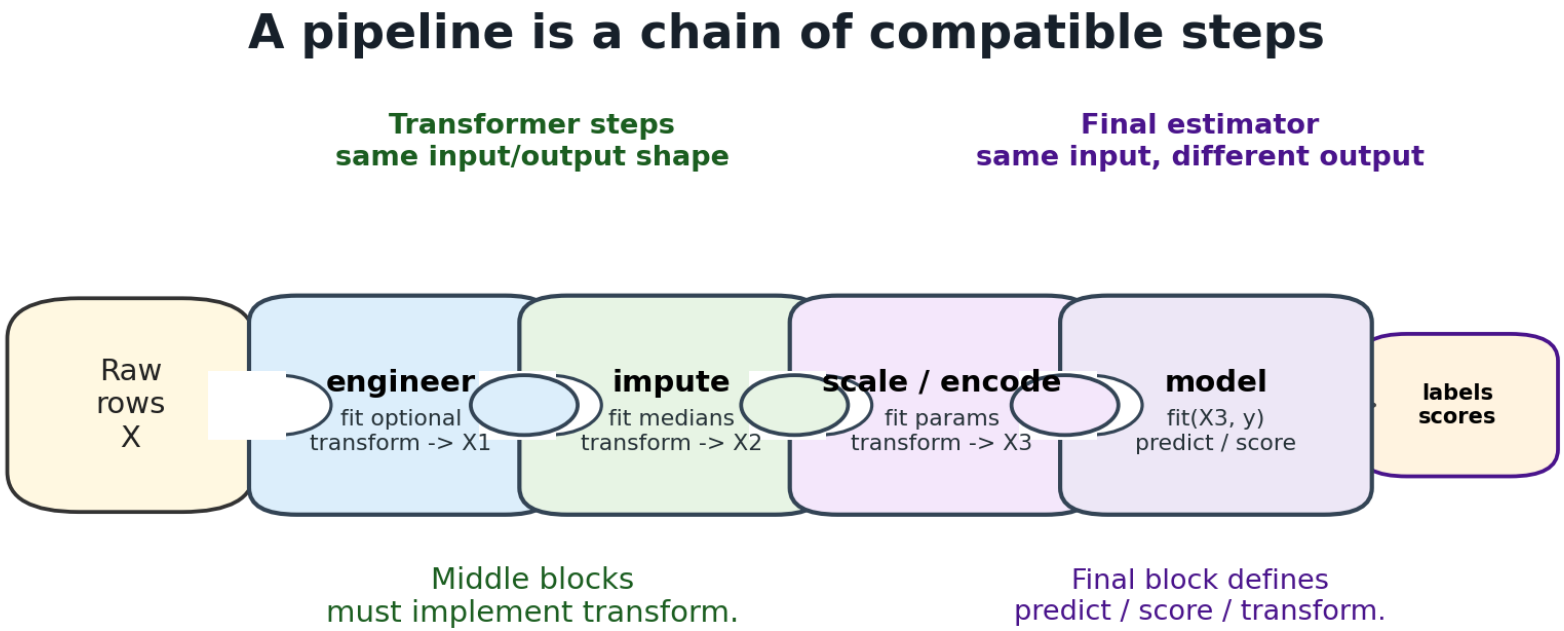
```
from sklearn.compose import ColumnTransformer,
    make_column_selector as selector
from sklearn.preprocessing import OneHotEncoder, StandardScaler,
    FunctionTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
```

- These classes handle preprocessing and modelling steps.

Pipelines, not manual steps

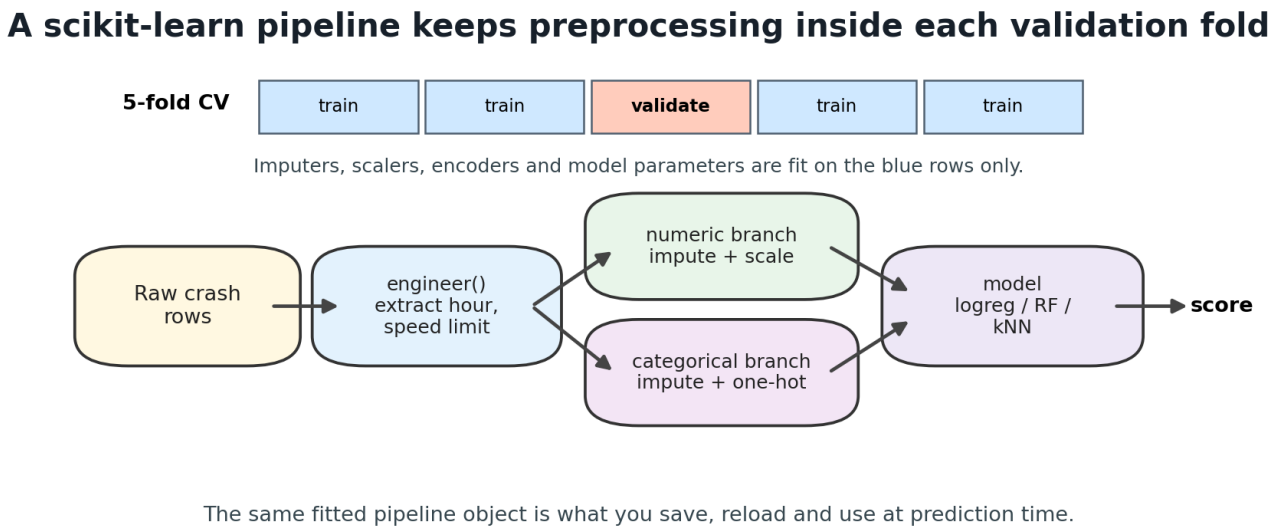
- Chain preprocessing and modelling together.
- Avoid forgetting steps on new data.

What is a pipeline?



This is why the blocks can be swapped, searched and cross-validated as one object.

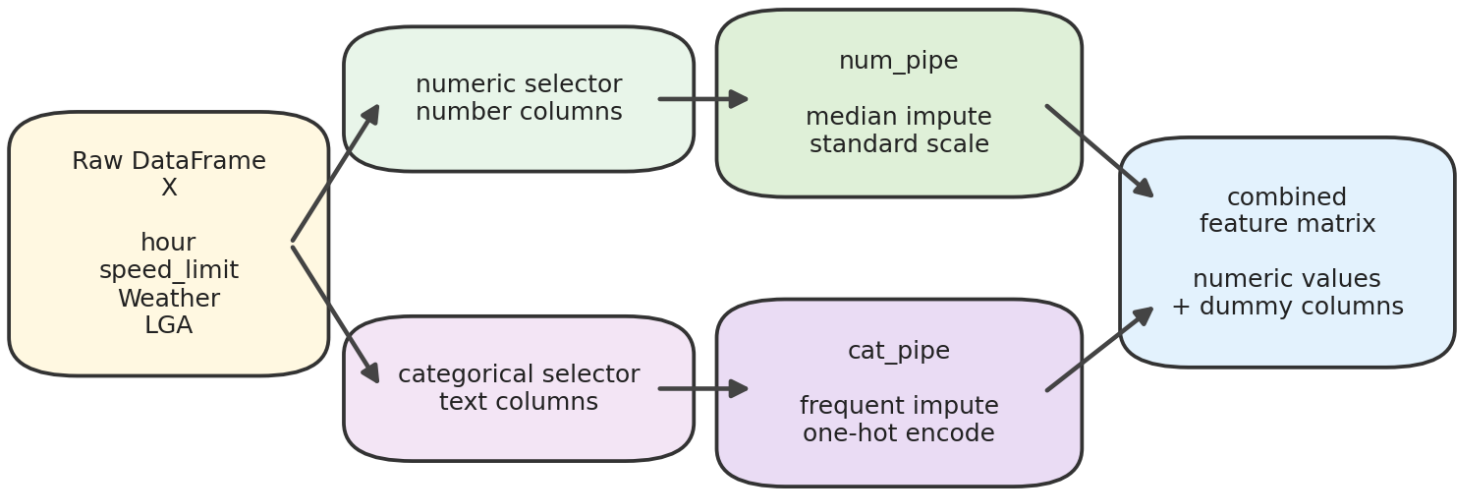
Pipeline and validation shape



Leakage trap

Target columns and future-only fields must be removed before the model sees the feature matrix `X`.

ColumnTransformer splits and recombines



The model sees one matrix, but each column type was cleaned in the right way.

Planned manipulations

- Extract numbers from strings.
- Remove missing values.
- Scale features to equalise mean and variance.

Regular expression basics

- `\d` matches a digit; `\d{1,2}` matches 1 or 2 digits.
- Parentheses capture a group.
- Prefix patterns with `r"..."` to keep backslashes literal.

Using `str.extract`

```
s = pd.Series(["60 km/h", "80 km/h"])
s.str.extract(r"(\d+)")
```

- Pulls out digits from strings.
- Apply `astype(float)` to turn the result into numbers.

Safe feature engineering

```
def engineer(df):
    df = df.copy()
    df["hour"] = (
        df["Two-hour intervals"]
          .str.extract(r"^(\d{1,2})", expand=False)
          .astype(float)
          .fillna(-1)
    )
    df["speed_limit"] = (
        df["Speed limit"]
          .str.extract(r"(\d+)", expand=False)
          .astype(float)
          .fillna(-1)
    )
    return df.drop(columns=["Two-hour intervals", "Speed limit",
                            "Degree of crash", "Degree of crash
    - detailed"])
```

- `engineer` creates numeric `hour` and `speed_limit` features

Reminder: information leakage

- Training data must not contain information unavailable at prediction time.
- Leaking test data or future features gives overly optimistic results.

Live coding: spot the bug

```
def engineer(df):
    df = df.copy()
    df["hour"] = df["Two-hour
    intervals"].str.extract(r"^(\d{1,2})").astype(float)
    df["speed_limit"] = df["Speed
    limit"].str.extract(r"(\d+)").astype(float)
    return df  # BUG: target columns leak through
```

- What risk does this bug introduce?

Why pipelines, not manual steps?

- **One object** encapsulates all preprocessing and the model.
- **No leakage**: fit transformers inside cross-validation.
- **Swap models** without touching preprocessing.
- **Deployable**: same code at training and prediction.

Numeric pipeline

```
num_pipe = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("scale", StandardScaler())
])
```

- Fill missing numeric values then scale.

Categorical pipeline

```
cat_pipe = Pipeline([
    ("impute", SimpleImputer(strategy="most_frequent")),
    ("encode", OneHotEncoder(handle_unknown="ignore"))
])
```

- Replace missing categories and one-hot encode them.

Combine with ColumnTransformer

```
preprocess = ColumnTransformer([
    ("num", num_pipe, selector(dtype_include="number")),
    ("cat", cat_pipe, selector(dtype_exclude="number"))
])
```

- Applies the right pipeline to each column.

Baseline logistic regression

```
log_clf = Pipeline([
    ("engineer", FunctionTransformer(engineer, validate=False)),
    ("preprocess", preprocess),
    ("model", LogisticRegression(max_iter=1000))
])
```

- `FunctionTransformer` runs our `engineer` function.
- `preprocess` handles numeric and categorical columns.
- `LogisticRegression` learns to predict injury risk.

Checkpoint

- Why wrap feature engineering and preprocessing in a pipeline?

Answer

- Ensures the exact same steps are applied during training and testing.
- Transformers fit only on training data, preventing leakage.

Cross-validation review

- Split data into several folds.
- Train on some folds and validate on the remaining fold.
- Average scores across folds for a robust estimate.

Cross-validation

```
from sklearn.model_selection import cross_validate

scoring = {"accuracy": "accuracy", "roc_auc": "roc_auc"}

scores = cross_validate(log_clf, X, y, scoring=scoring,
    n_jobs=-1)
print("LogReg accuracy:", scores["test_accuracy"].mean())
print("LogReg ROC AUC:", scores["test_roc_auc"].mean())
```

- Default 5-fold CV reports accuracy and ROC AUC.

Poll: choosing a metric

- Which metric better handles class imbalance?
  1. Accuracy
  2. F1 score

Poll: expected winner

- Which model do you expect to score higher ROC AUC?
  1. Logistic regression
  2. k-nearest neighbours
  3. Random forest

Random forest alternative

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = Pipeline([
    ("engineer", FunctionTransformer(engineer, validate=False)),
    ("preprocess", preprocess),
    ("model", RandomForestClassifier(random_state=42))
])

rf_scores = cross_validate(rf_clf, X, y,
                           scoring=scoring, n_jobs=-1)
print("RF accuracy:", rf_scores["test_accuracy"].mean())
print("RF ROC AUC:", rf_scores["test_roc_auc"].mean())
```

- Swap logistic regression for a random forest.
- Compare cross-validated metrics to choose a model.
- Examine `rf_clf["model"].feature_importances_` for insights.

kNN alternative

```
from sklearn.neighbors import KNeighborsClassifier

knn_clf = Pipeline([
    ("engineer", FunctionTransformer(engineer, validate=False)),
    ("preprocess", preprocess),
    ("model", KNeighborsClassifier())
])

knn_scores = cross_validate(knn_clf, X, y,
                            scoring=scoring, n_jobs=-1)
print("kNN accuracy:", knn_scores["test_accuracy"].mean())
print("kNN ROC AUC:", knn_scores["test_roc_auc"].mean())
```

- Try a non-parametric alternative with k-nearest neighbours.

Fit logistic regression on all data

```
log_clf.fit(X, y)
feature_names = (
    log_clf.named_steps["preprocess"]
           .get_feature_names_out()
)
coefs = pd.Series(
    log_clf.named_steps["model"].coef_[0],
    index=feature_names
).sort_values(key=abs, ascending=False)
coefs.head()
```

- Inspect coefficients after preprocessing has created the actual model columns.

Top coefficients

```
coefs.head()
```

- Larger magnitude implies stronger influence on injury risk.

Fit random forest on all data

```
rf_clf.fit(X, y)
feature_names = (
    rf_clf.named_steps["preprocess"]
          .get_feature_names_out()
)
importances = pd.Series(
    rf_clf.named_steps["model"].feature_importances_,
    index=feature_names
).sort_values(ascending=False)
importances.head()
```

- Feature importances must line up with the encoded feature names.

Top importances

```
importances.head()
```

Present your recommendation

- Which model and features would you present to stakeholders?

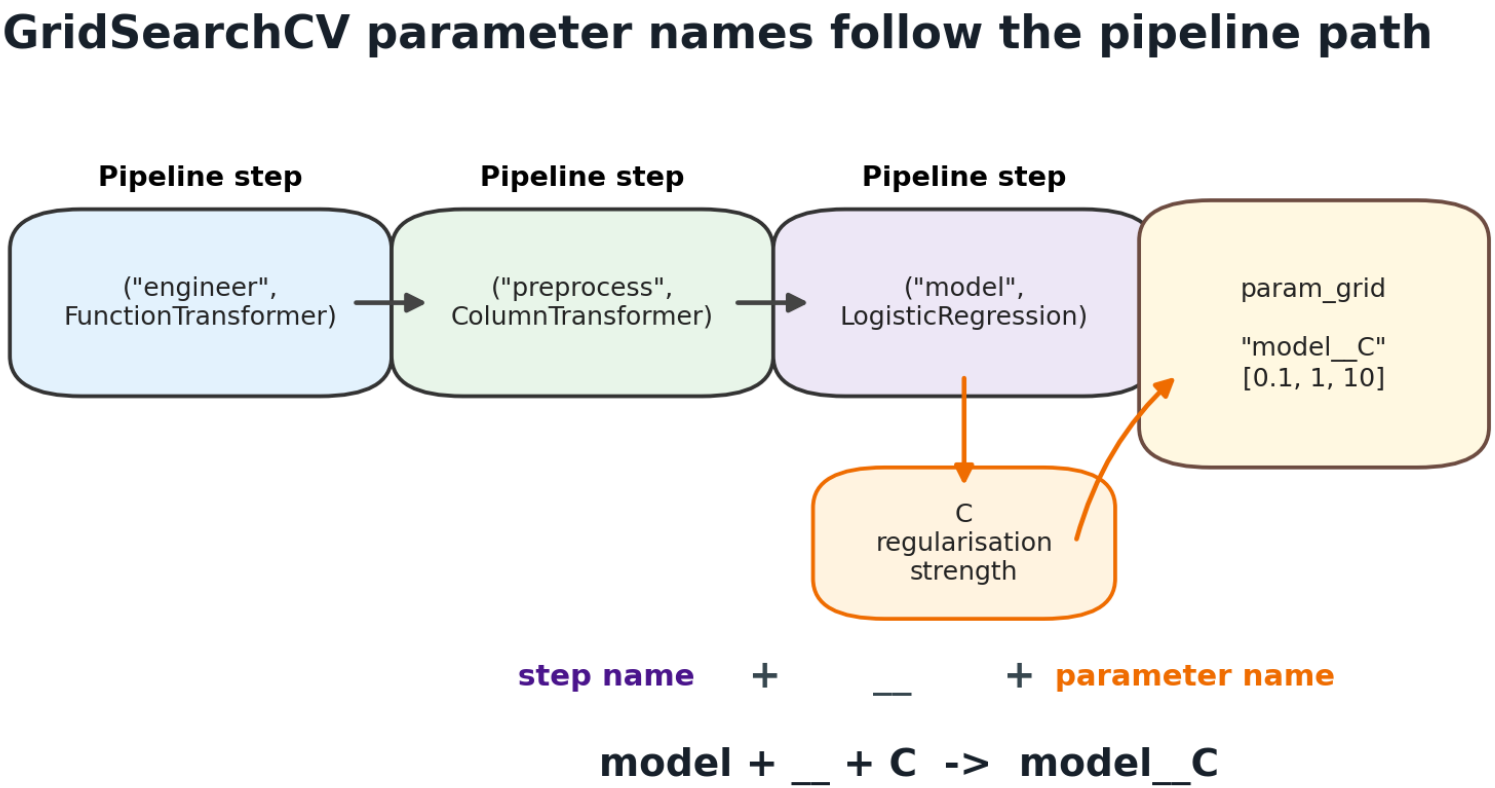
Why tune hyperparameters?

- Models have settings that influence performance.
- Search over these hyperparameters to find better scores.

You have seen this already

- Week 5: choose k for k-nearest neighbours using validation or cross-validation.
- Week 6: Orange Parameter Fitter tried different random-forest settings.
- Week 9: the same idea, but inside a Python `Pipeline`.
- Pipeline parameter names use `step__parameter`, e.g. `model__C`.

Where `model__C` comes from



GridSearchCV: simple, explicit

```
from sklearn.model_selection import GridSearchCV
param_grid = {
  'model__C': [0.1, 1, 10],
  'model__penalty': ['l2'],              # keep it safe for lbfgs
  'model__solver': ['lbfgs'],
}
gs = GridSearchCV(log_clf, param_grid=param_grid, cv=5,
    scoring='roc_auc', n_jobs=-1)
gs.fit(X, y)
print(gs.best_params_, gs.best_score_)
best = gs.best_estimator_
```

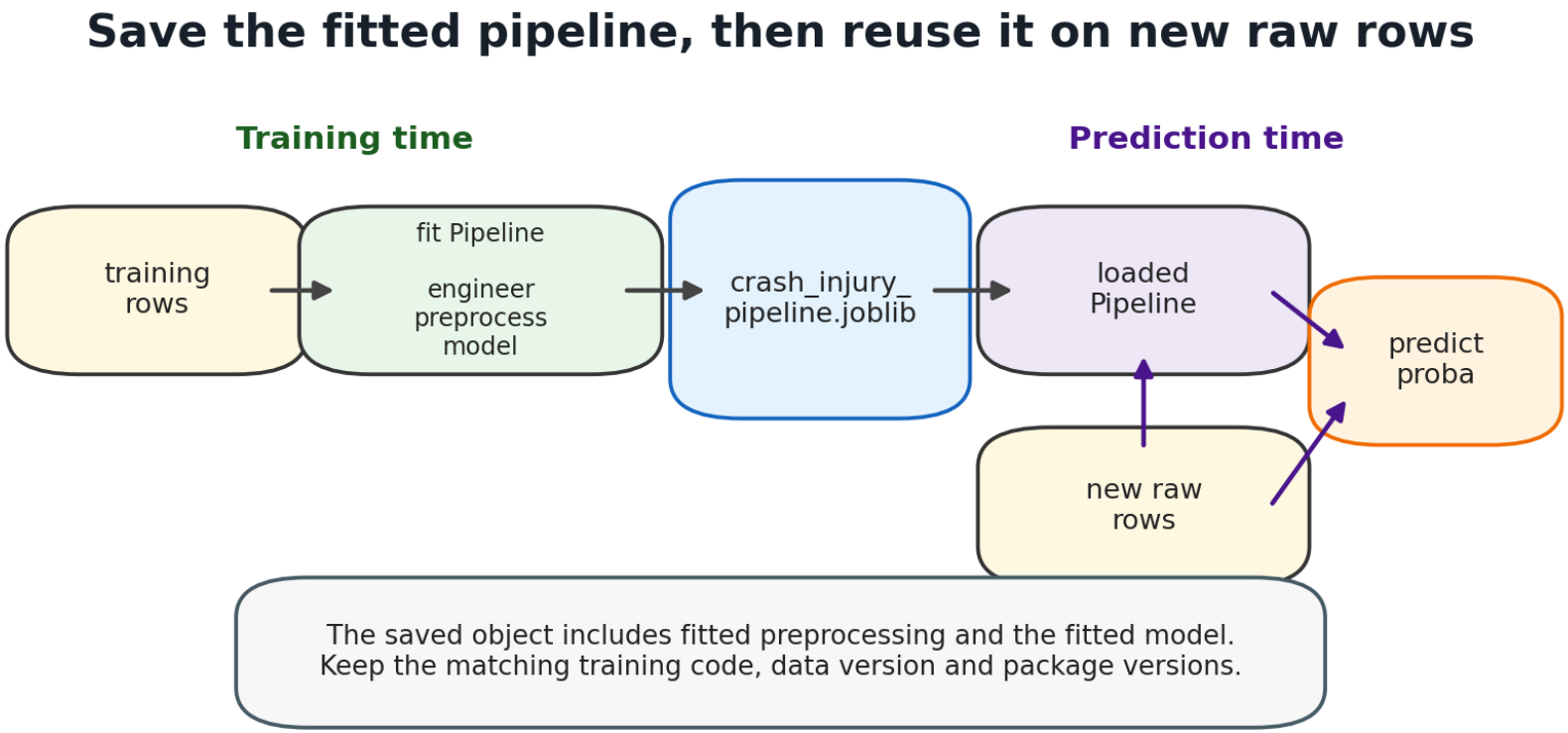
RandomizedSearchCV: quick scan

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

param_dist = {'model__C': loguniform(1e-2, 1e2)}
rs = RandomizedSearchCV(log_clf, param_distributions=param_dist,
                        n_iter=25, cv=5, scoring='roc_auc',
                        n_jobs=-1, random_state=42)
rs.fit(X, y)
rs.best_params_, rs.best_score_
```

From training to inference

- Training is where we fit feature engineering, preprocessing, and the model.
- Inference is where we use that fitted object to make predictions later.
- The inference program must apply the same parsing, imputation, scaling, encoding, and model.
- A fitted `Pipeline` keeps those steps together.

Save the fitted workflow



Save the fitted pipeline

```
import joblib

final_model = best
final_model.fit(X, y)

joblib.dump(final_model, "crash_injury_pipeline.joblib")
```

- If you skipped `GridSearchCV`, use `final_model = log_clf`.
- Save the whole fitted pipeline, not just the classifier at the end.
- Keep the training code and data version with the saved model.

Joblib and custom transformers

- `joblib.dump` stores the fitted pipeline state.
- It stores references to custom Python functions; it does not bundle the source code from a notebook cell.
- Our pipeline uses `FunctionTransformer(engineer, validate=False)`.
- Put `engineer` in an importable `.py` file, such as `crash_pipeline.py`, before using the saved model for inference.

Load it and predict later

```
import joblib
from crash_pipeline import engineer

loaded = joblib.load("crash_injury_pipeline.joblib")

new_rows = raw.sample(5,
    random_state=1).drop(columns=injury_cols)
loaded.predict(new_rows)
loaded.predict_proba(new_rows)[:, 1]
```

- `joblib` is convenient for your own scikit-learn models.
- The module containing `engineer` must be available when loading.
- Do not load model files from untrusted sources.

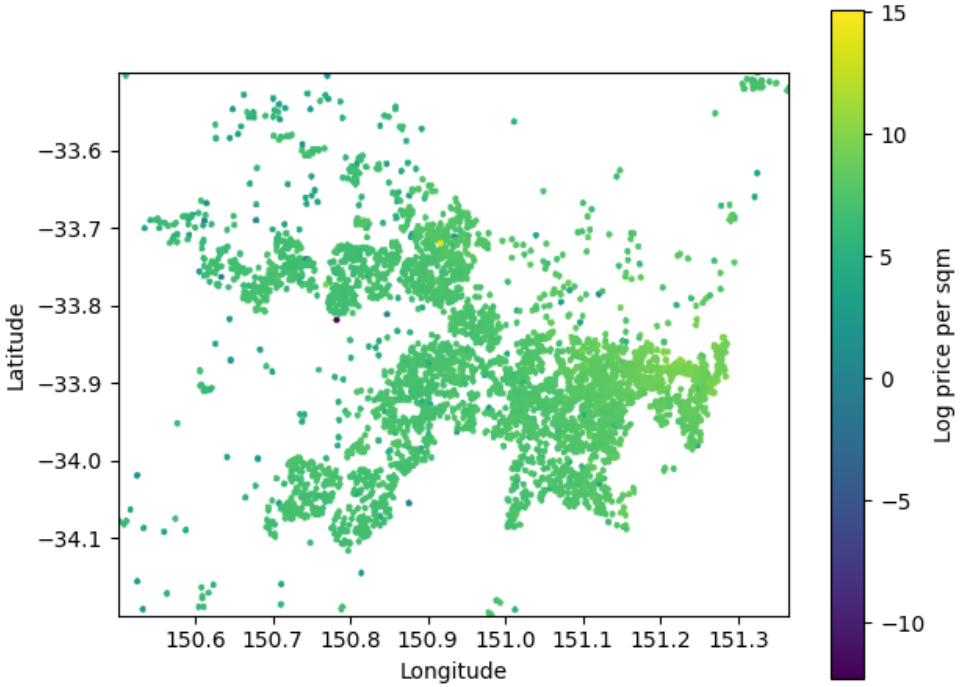
Production checklist for this course

- Can another person re-run the training script and get a valid model?
- Does the saved object include preprocessing as well as the model?
- Do you know which data, code, and package versions created it?
- Have you tested prediction on new rows that were not used for training?

Same pipeline idea, but for regression

- We do not need a totally different workflow when the target is a number.
- Keep the preprocessing pattern, swap in a regressor, and change the evaluation metrics.
- A simple example is predicting Sydney land value per square metre from latitude and longitude.

Sydney land values



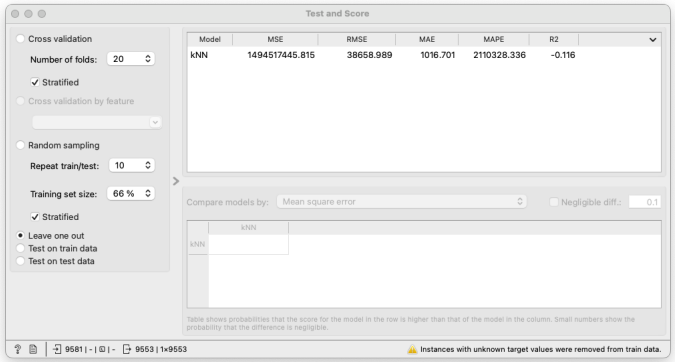
- Target: `price_per_sqm`
- Features: `latitude`, `longitude`
- Nearby places often have similar values, so k-NN regression is a natural baseline
- Tree and random-forest regressors are good follow-up comparisons

kNN regressor on land values

```
from sklearn.neighbors import KNeighborsRegressor

land = pd.read_csv("../Week6/sydney-landvalues.csv")
X_land = land[["latitude", "longitude"]]
y_land = land["price_per_sqm"]

land_knn = Pipeline([
    ("scale", StandardScaler()),
    ("model", KNeighborsRegressor(n_neighbors=5))
])

land_scores = cross_validate(
    land_knn, X_land, y_land,
    scoring=("neg_mean_absolute_error",
             "neg_root_mean_squared_error", "r2"),
    n_jobs=-1
)
```

Reading regression metrics

- **MAE**: average absolute miss in dollars
- **RMSE**: like MAE, but punishes large misses more
- R^2: how much of the signal the model is capturing
- There is no "accuracy" for predicting a number
- Compare models by asking which one misses by less



Pitfalls

- Not stratifying classification splits.
- Fitting preprocessors on full data (leakage).
- Forgetting `handle_unknown='ignore'` with one-hot encoding.
- Using accuracy on imbalanced data; prefer F1/AUC.
- Not setting `random_state` when comparing models.
- Tuning on the test set; keep a hold-out or use nested CV for research-grade claims.

Takeaways

- Pipelines give you correctness by default and easier deployment.
- Start with a robust preprocessing template; swap models as needed.
- Validate properly, then tune a couple of high-impact knobs.
- The same pipeline shape extends to simple regression tasks such as land-value prediction.
- Save your best pipeline; version your data and code.