COMP2200/COMP6200 — Week 8
Orange → Python → `Pipeline` (Supervised)

Greg Baker

15th September 2025

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

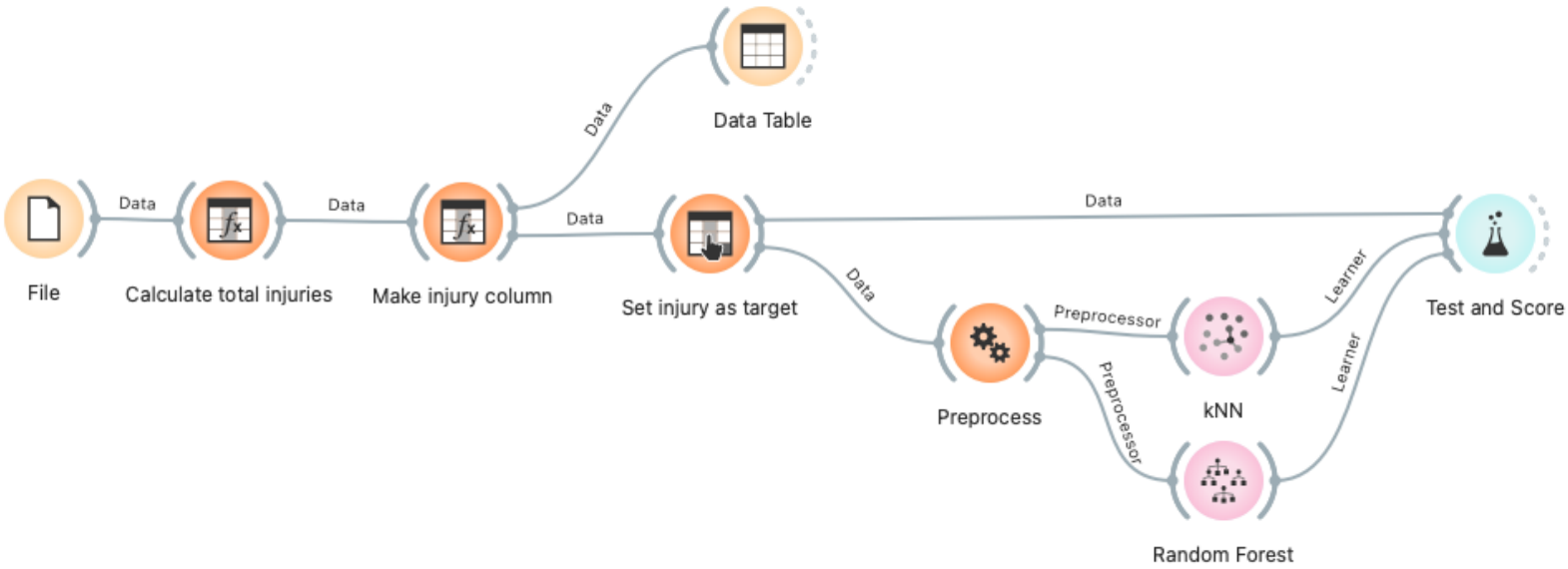
NSW recorded crashes from 2019–2023. Each row describes a crash and whether anyone was injured.
Our goal: predict whether a crash leads to any injury so we can help people drive more safely.

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
```

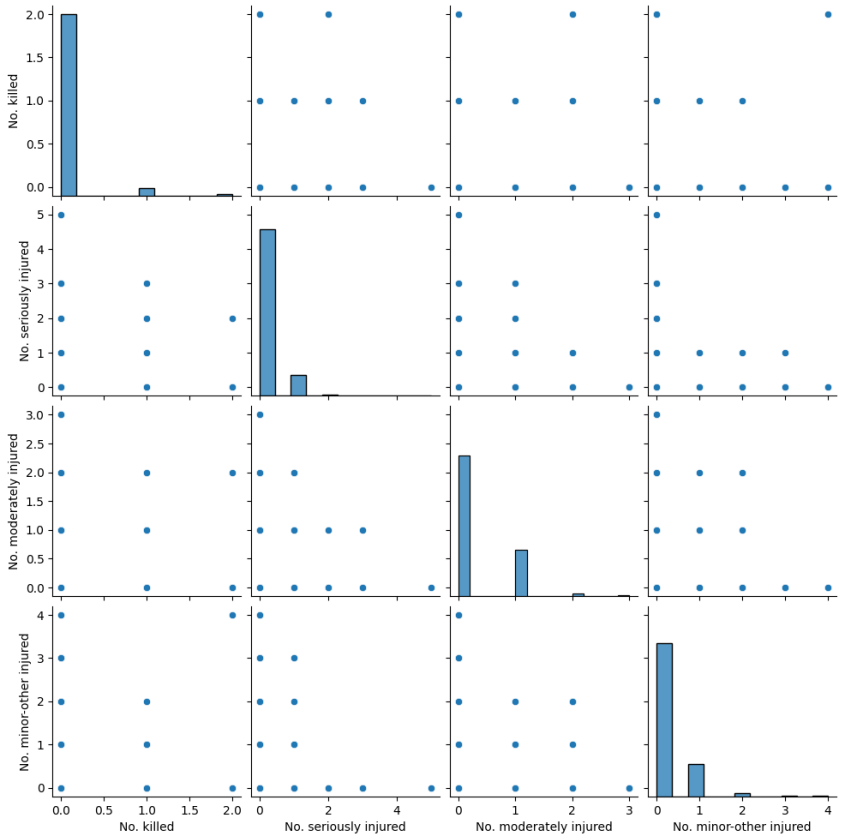
- Quick numeric overview.

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

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?

- All steps except the last must implement `transform`.
- `fit` calls each step in sequence, passing transformed data on.
- The pipeline exposes the methods of the final estimator.

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

- Train on full data and inspect coefficients.

Top coefficients

```
coefs.head()
```

- Larger magnitude implies stronger influence on injury risk.

Fit random forest on all data

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

- Feature importances highlight predictive columns.

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.

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_
```

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.
- Save your best pipeline; version your data and code.