COMP2200/COMP6200 Week 9
Python Pipeline Code Spells

Supervised Learning Pipelines

How to use this

This week is syntactically heavy. You are not expected to memorise every line. The useful skill is recognising the pattern:

1. choose `X` and `y`,
2. build preprocessing,
3. wrap preprocessing and the model in a `Pipeline`,
4. evaluate with cross-validation,
5. fit the final pipeline and use it for prediction.

When you feel lost, start from one of the complete code spells below and change only the dataset columns, model, or metric.

Imports introduced or reused this week

```
import pandas as pd

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

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.dummy import DummyClassifier

from sklearn.model_selection import cross_validate, cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.inspection import permutation_importance

import joblib
```

You do not need all of these in every script. Import the ones used by the spell you are running.

Pandas setup spells

Spell	Meaning
`pd.read_excel("file.xlsx")`	Read an Excel file into a DataFrame.
`pd.read_csv("file.csv")`	Read a CSV file into a DataFrame.
`raw.columns`	Show column names.
`raw.describe()`	Numeric summary: count, mean, standard deviation, quartiles.
`raw.shape`	Tuple `(rows, columns)`.
`raw.head()`	First few rows.
`df["column"]`	Select one column as a Series.
`df[["a", "b"]]`	Select several columns as a DataFrame.
`df.result.value_counts()`	Count class labels or categories.

Target and feature spells

Crash-data target:

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

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

- `axis=1`: across columns, one result per row.
- `axis=0`: down rows, one result per column.
- `(injury_total > 0)` creates True/False values.
- `.astype(int)` turns True/False into 1/0.
- `drop(columns=...)` removes columns from the input features.

Tic-tac-toe target and features:

```
df = pd.read_csv("all_tictactoe_games.csv")
y = df["result"]
X = df[["move1", "move2", "move3"]]

all_moves = [f"move{i}" for i in range(1, 10)]
X_all = df[all_moves]
```

`[f"move{i}" for i in range(1, 10)]` is a list comprehension. It builds `["move1", ..., "move9"]`. The upper end of `range` is not included.

Feature engineering spells

Make a safe copy before editing:

```
def engineer(df):
    df = df.copy()
    ...
    return df
```

`df.copy()` avoids accidentally changing the original DataFrame.

Extract numbers from strings:

```
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)
)
```

Regex spell	Meaning
`r"..."`	Raw string. Backslashes mean regex things, not Python escape codes.
`\d`	A digit.
`\d+`	One or more digits.
`\d{1,2}`	One or two digits.
`^`	Start of the string.
`(...)`	Capture this part and return it.
`expand=False`	Return a Series instead of a one-column DataFrame.

Drop columns after engineering:

```
return df.drop(columns=[
    "Two-hour intervals",
    "Speed limit",
    "Degree of crash",
    "Degree of crash - detailed",
])
```

This removes columns we have replaced or columns that would leak the answer.

Preprocessing spells

Numeric preprocessing:

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

Categorical preprocessing:

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

Combine numeric and categorical preprocessing:

```
preprocess = ColumnTransformer([
    ("num", num_pipe, selector(dtype_include="number")),
```

```
    ("cat", cat_pipe, selector(dtype_exclude="number")),
])
```

- `SimpleImputer`: fills missing values.
- `StandardScaler`: puts numeric columns onto a comparable scale.
- `OneHotEncoder`: turns categories into numeric indicator columns.
- `handle_unknown="ignore"`: do not crash if prediction-time data has a new category.
- `ColumnTransformer`: sends different columns through different preprocessing steps.
- `selector(dtype_include="number")`: choose numeric columns automatically.

Full pipeline spells

Crash-data supervised pipeline:

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

`FunctionTransformer(engineer, validate=False)` lets our own DataFrame function behave like a scikit-learn transformer. Use `validate=False` when the function expects a pandas DataFrame with column names.

Tic-tac-toe categorical pipeline:

```
preprocess = ColumnTransformer([
    ("ohe", OneHotEncoder(handle_unknown="ignore"), X.columns)
])

pipe = Pipeline([
    ("prep", preprocess),
    ("clf", LogisticRegression(max_iter=500)),
])
```

Pipeline step names are your handles for later. In the lecture we use `"model"`; in the practical solution we use `"clf"`. Either is fine, but you must use the same name consistently.

Evaluation spells

Several metrics with `cross_validate`:

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

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

One metric with `cross_val_score`:

```
cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=0)

scores = cross_val_score(pipe, X, y, cv=cv, scoring="f1_macro")
print(scores.mean())
```

Dummy baseline:

```
dummy = DummyClassifier()
print(cross_val_score(dummy, X, y, cv=cv).mean())
```

- `cross_validate`: use when you want multiple metrics.
- `cross_val_score`: use when you want one metric.
- `StratifiedKFold`: keeps class proportions similar in each fold.
- `n_jobs=-1`: use all available CPU cores.
- `f1_macro`: average F1 across classes, useful when classes are imbalanced.

Model comparison spells

To try another model, keep the preprocessing and change the final step.

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

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

Then evaluate each pipeline with the same `X`, `y`, folds, and scoring. That keeps the comparison fair.

Inspecting fitted pipelines

You can inspect fitted pipeline internals after calling `fit`.

Names of post-preprocessing features:

```
log_clf.fit(X, y)
feature_names = (
    log_clf.named_steps["preprocess"]
           .get_feature_names_out()
)
```

Logistic regression coefficients:

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

print(coefs.head())
```

Random forest importances:

```
importances = pd.Series(
    rf_clf.named_steps["model"].feature_importances_,
    index=feature_names
).sort_values(ascending=False)
```

Per-class coefficients in a multi-class model:

```
classifier = pipe.named_steps["clf"]
for class_name, coefficients in zip(classifier.classes_, classifier.coef_):
    strengths = pd.Series(index=feature_names, data=coefficients)
    print(class_name)
    print(strengths.nlargest(10))
    print(strengths.nsmallest(10))
```

Permutation importance:

```
importance = permutation_importance(
    pipe,
    importance_data[["move1", "move2", "move3"]],
    importance_data["result"],
    scoring="f1_macro",
    n_repeats=5,
    random_state=0,
)

move_importance = pd.Series(
    index=X.columns,
    data=importance.importances_mean,
)
```

Hyperparameter search spells

Pipeline hyperparameter names use:

`step_name__parameter_name`

So `model__C` means: go to the `"model"` step, then set the `C` parameter on that model.

Grid search:

```
param_grid = {
    "model__C": [0.1, 1, 10],
    "model__penalty": ["l2"],
    "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_
```

Randomized search:

```
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)
```

Saving, loading, and inference spells

Save the whole fitted pipeline:

```
final_model = best
final_model.fit(X, y)

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

Load it later and predict:

```
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]
```

- Save the fitted pipeline, not just the classifier.
- Keep the training script and data version with the saved model.
- If the pipeline uses `FunctionTransformer(engineer, validate=False)`, keep `engineer` in an importable `.py` file. `joblib` stores a reference to the function, not a copy of the notebook cell.
- Do not load `joblib` files from untrusted sources.
- `predict_proba(...)[:, 1]` means: all rows, probability column for class 1.

Common mistakes and fixes

- **The model sees the answer in `X`.** Drop target columns and future-only columns before fitting.
- **Coefficient counts do not match original columns.** One-hot encoding creates new columns. Use `get_feature_names_out()`.
- **`GridSearchCV` says `C` is not a valid parameter.** Use the pipeline name: `model__C` or `clf__C`.
- **Prediction data has a category not seen during training.** Set `handle_unknown="ignore"` in `OneHotEncoder`.
- **The custom engineering function loses column names.** Use `FunctionTransformer(engineer, validate=False)`.
- **Cross-validation scores look too good.** Check for leakage and make sure preprocessing is inside the pipeline.
- **`axis=0` and `axis=1` are mixed up.** `axis=0` goes down rows; `axis=1` goes across columns.

- **Inspecting `named_steps` gives unfitted objects.** Call `fit` first, or use `best_estimator_` from a fitted search.

Minimal skeleton

```
# 1. Load data
df = pd.read_csv("data.csv")
y = df["target"]
X = df.drop(columns=["target"])

# 2. Preprocess
num_pipe = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("scale", StandardScaler()),
])
cat_pipe = Pipeline([
    ("impute", SimpleImputer(strategy="most_frequent")),
    ("encode", OneHotEncoder(handle_unknown="ignore")),
])
preprocess = ColumnTransformer([
    ("num", num_pipe, selector(dtype_include="number")),
    ("cat", cat_pipe, selector(dtype_exclude="number")),
])

# 3. Model pipeline
pipe = Pipeline([
    ("preprocess", preprocess),
    ("model", LogisticRegression(max_iter=1000)),
])

# 4. Evaluate
scores = cross_validate(
    pipe,
    X,
    y,
    scoring={"accuracy": "accuracy"},
    n_jobs=-1,
)
print(scores["test_accuracy"].mean())

# 5. Fit final model
pipe.fit(X, y)
```