05_hyperparams

June 21, 2025

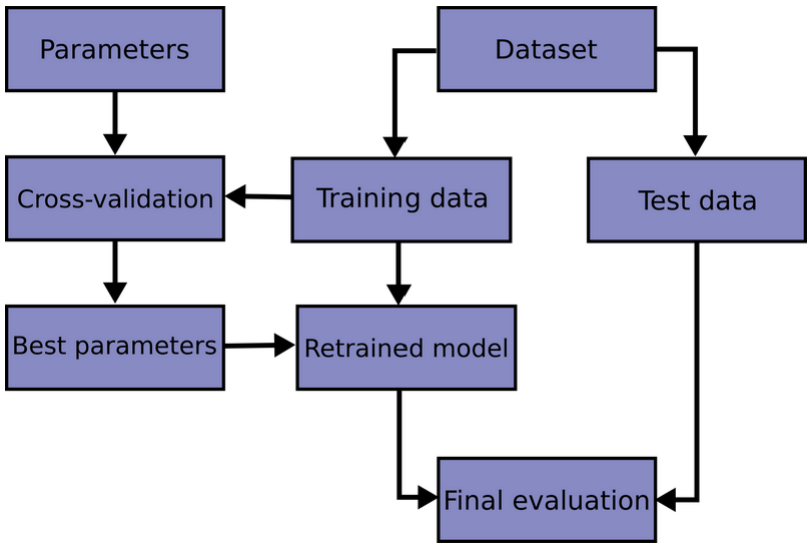
1 What are we doing?

1.1 Objectives

- Construct a cross-validation pipeline.
- Use cross-validation to evaluate different hyperparameter performance.
- Perform grid search for systemic evaluation.
- Store and manage results.

1.2 Procedure

The diagram below, taken from Scikit Learn's documentation, shows the procedure that we will follow:



- System requriements:
  - Automation: the system should operate automatically with the least amount of supervision.
  - Replicability: changes to code and (arguably) data should be logged and controled. Randomness should also be controlled (random seeds, etc.)
  - Persistence: persist results for later analysis.

1.3 What is a Hyperparameter?

- Generally speaking, hyperparameters are parameters that control the learning process: regularization weights, learning rate, entropy/gini metrics, etc.
- Hyperparameters will drive the behaviour and performance of a model. Model selection is intimately related with hyperparameter tuning.
- Selection critieria are based on performance evaluation and, to get better performance estimates, we use cross-validation.

1.4 Searching the Hyperparameter Grid

- To address the automation requirement, we could use `GridSearchCV()`, which is a self-contained function for performing a Grid Search over a hyperparameter space.
- To "Search the Hyperparameter Grid" exhaustively means that we will consider all possible combination of hyperparameter values in the search space and evaluate the model using those hyperparams. For example, if we have two parameters that we are exploring, kernel (takes values "rbf" and "poly") and C (takes values 1.0 and 0.5), then this grid would be the combinations:
  - (rbf, 1.0)
  - (rbf, 0.5)
  - (poly, 1.0)
  - (poly, 0.5)
- Under each combination, we perform CV and evaluate the model's performance.

2 Setup

We start with Give me some credit data that we used in the previous session.

[ ]:
```
%load_ext dotenv
%dotenv
import os
import sys
sys.path.append(os.getenv('SRC_DIR'))
import pandas as pd
import numpy as np
import os
ft_path = os.getenv("CREDIT_DATA")
df_raw = pd.read_csv(ft_path)
```

[ ]:
```
df = df_raw.drop(columns = ["Unnamed: 0"]).rename(
    columns = {
        'SeriousDlqin2yrs': 'delinquency',
        'RevolvingUtilizationOfUnsecuredLines':␣
 ↪'revolving_unsecured_line_utilization',
        'age': 'age',
        'NumberOfTime30-59DaysPastDueNotWorse': 'num_30_59_days_late',
```

```
        'DebtRatio': 'debt_ratio',
        'MonthlyIncome': 'monthly_income',
        'NumberOfOpenCreditLinesAndLoans': 'num_open_credit_loans',
        'NumberOfTimes90DaysLate':  'num_90_days_late',
        'NumberRealEstateLoansOrLines': 'num_real_estate_loans',
        'NumberOfTime60-89DaysPastDueNotWorse': 'num_60_89_days_late',
        'NumberOfDependents': 'num_dependents'
    }
).assign(
    high_debt_ratio = lambda x: (x['debt_ratio'] > 1)*1,
    missing_monthly_income = lambda x: x['monthly_income'].isna()*1,
    missing_num_dependents = lambda x: x['num_dependents'].isna()*1,
)
```

Use a simple pipeline composed of:

- Preprocessing steps.
- Logistic Regression classifier.

We will explore the hyperparameter sapce by evaluating different regularization strategies and parameters.

[ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
```

[ ]:
```
num_cols = ['revolving_unsecured_line_utilization', 'age',
       'num_30_59_days_late', 'debt_ratio', 'monthly_income',
       'num_open_credit_loans', 'num_90_days_late', 'num_real_estate_loans',
       'num_60_89_days_late', 'num_dependents',
       # Although expressed as numbers, these columns are boolean:
       # 'high_debt_ratio',
       # 'missing_monthly_income',
       # 'missing_num_dependents'
       ]

pipe_num_simple = Pipeline([
    ('imputer', SimpleImputer(strategy = 'median')),
    ('standardizer', StandardScaler())
])

ctransform_simple= ColumnTransformer([
    ('numeric_simple', pipe_num_simple, num_cols),
], remainder='passthrough')
```

```
pipe_lr = Pipeline([
    ('preprocess', ctransform_simple),
    ('clf', LogisticRegression())
])
pipe_lr
```

Obtain the parameters of the pipeline with `.get_params()`.

[ ]:
```
pipe_lr.get_params()
```

2.1 Setup the Splitting Strategy

[ ]:
```
X = df.drop(columns = 'delinquency')
Y = df['delinquency']

scoring = ['neg_log_loss', 'roc_auc', 'f1', 'accuracy', 'precision', 'recall']

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,␣
 ↪random_state = 42)
```

To perform the Grid Search we need to define a parameter grid:

- A parameter grid defines all of the combinations of parameters that we need to explore.
- The function `GridSearchCV()` performs an exhaustive search of parameter combinations.
- The parameter grid is defined as a dictionary of lists:
  - Each entry's key is the name of the parameter.
  - Each entry's value is the list of values that we would like to explore.

[ ]:
```
param_grid = {
    'clf__C': [0.01, 0.5, 1.0],
    'clf__penalty': ['l1', 'l2'],
    'clf__solver': ['liblinear'],
    }
```

Some key inputs to `GridSearchCV` are:

- `estimator`: the pipeline or classifier that we are tuning.
- `param_grid`: the parameter grid defined as a dictionary of lists described above.
- `n_jobs`: settings for parallel computation.
- `refit`: options for refitting the model using the best-performing configuration.

[ ]:
```
grid_cv = GridSearchCV(
    estimator=pipe_lr,
    param_grid=param_grid,
    scoring = scoring,
    cv = 5,
    refit = "neg_log_loss")
```

```
grid_cv.fit(X_train, Y_train)
```

Access the cross-validation results using the property `.cv_results_`:

[ ]:
```
res = grid_cv.cv_results_
res = pd.DataFrame(res)
res.columns

res[['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_clf__C', 'param_clf__penalty', 'param_clf__solver', 'params',
       'mean_test_neg_log_loss',
       'std_test_neg_log_loss', 'rank_test_neg_log_loss']].
 ↪sort_values('rank_test_neg_log_loss')
```

Access the best-performing configuration:

[ ]:
```
grid_cv.best_params_
```

[ ]:
```
grid_cv.best_estimator_
```

The best-performing classifier (pipeline) trained on the complete training set is:

3 Tracking GridSearchCV Experiments

- We can expand our infrastructure for hyperparameter tuning across various models.
- The plan:
  - Create a model ingredient to obtain the classifier object.
  - Create experiment param grids to organize our parameter grids.
  - Schedule the experiments.

Explore the code in `./05_src/exp__logistic_simple.py` and `./05_src/exp__logistic_grid_search.py`:

- `exp__logistic_simple.py` implements a single experiment run in MLFlow, i.e., a single set of parameters will be trained and evaluated by the code.
- `exp__logistic_grid_search.py` runs through a series of tests (one test given by a parametrization of the model pipeline). Each run is recorded independently as a parent run.
- Also notice that we have pulled the data component of the experiment to a module of its own.

3.1 Running Experiments from the Command Line

Access the experiment through the Command Line Interface.

```
cd src  # if required
python -m credit.exp__logistic_grid_search.py
```