Hyperparameter Tuning

A *hyperparameter* is a parameter that is not learned during the training of an estimator. *Tuning* a hyperparameter is the process of searching for hyperparameter values which optimize the performance of the estimator. The primary danger we need to guard against is over-fitting, and our primary defense will be *nested cross-validation*.

We have already seen the regularization hyperparameters of the Ridge and Lasso linear models, as well as the number of components in Principle Component Analysis.

The documentation for any estimator should include information about its hyperparameters. You can use the `.get_params()` method of an instantiated estimator returns a dictionary of its hyperparameters.

In [5]:
```
from sklearn.ensemble import RandomForestRegressor

# The .get_params() method of an instantiated estimator returns a dict
RandomForestRegressor().get_params()
```

Out[5]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': None,
 'max_features': 1.0,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

Nested Cross-validation

Nested cross-validation works in two layers:

- Outer loop (evaluation): The data is split into train/test folds, just like in standard cross-validation. Each outer test fold is held out to estimate performance.
- Inner loop (tuning): Within each outer training set, another cross-validation is run to select the best hyperparameters.

For each outer split, hyperparameters are tuned only on the training portion (inner CV), and the resulting model is then tested on the outer holdout. Averaging across all outer folds provides an unbiased estimate of how the tuned model will perform on unseen data.

Without nested cross-validation, there's a risk of overfitting to a particular split: among the many hyperparameter combinations tried, some may appear to perform well on that split purely by chance, but won't generalize as well to new data.

We demonstrate that now by fitting `RandomForestClassifier` to data where the target values are draws from a Bernoulli random variable and are completely independent from the target. We will see that using regular cross-validation to select the best hyperparameters results in overfitting, while nested cross-validation does not.

In [22]:
```
import numpy as np
from sklearn.model_selection import train_test_split, RandomizedSearch
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.base import clone
from scipy.stats import randint, uniform

rng = np.random.default_rng(231234)

# Generate random features and random binary labels (pure noise)
X = rng.normal(size=(100, 10))  # 200 samples, 10 features
y = rng.binomial(1, 0.5, size=100)  # labels 0 or 1, 50/50

# Single train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.

# Hyperparameter distributions for RandomizedSearchCV
param_distributions = {
    "n_estimators": randint(50, 500),
    "max_depth": randint(2, 50),
    "max_features": uniform(0.1, 0.9),
    "min_samples_split": randint(2, 10),
    "min_samples_leaf": randint(1, 10),
    "bootstrap": [True, False]
```

```python
}

# 1) Hyperparameter tuning with single CV (overfitting example)
search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_distributions,
    n_iter=200,
    cv=5,
    scoring='accuracy',
    random_state=42,
    n_jobs=-1
)
search.fit(X_train, y_train)

best_model = search.best_estimator_
single_cv_val = search.best_score_
single_cv_test = accuracy_score(y_test, best_model.predict(X_test))

print("Single CV tuning (likely overfit):")
print("Best hyperparameters:", search.best_params_)
print("Validation Accuracy:", single_cv_val)
print("Test Accuracy:", single_cv_test)


# 2) Nested CV
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
nested_scores = []
inner_cv_scores = []


for train_idx, test_idx in outer_cv.split(X):
    X_outer_train, X_outer_test = X[train_idx], X[test_idx]
    y_outer_train, y_outer_test = y[train_idx], y[test_idx]

    # Inner CV for hyperparameter tuning
    inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)
    inner_search = RandomizedSearchCV(
        RandomForestClassifier(random_state=42),
        param_distributions,
        n_iter=200,
        cv=inner_cv,
        scoring='accuracy',
        random_state=42,
        n_jobs=-1
    )
    inner_search.fit(X_outer_train, y_outer_train)

    inner_cv_scores.append(float(inner_search.best_score_))


    # Evaluate best model on outer test fold
    best_inner_model = clone(inner_search.best_estimator_)
    best_inner_model.fit(X_outer_train, y_outer_train)
    acc = accuracy_score(y_outer_test, best_inner_model.predict(X_oute
    nested_scores.append(acc)

print("\nNested CV (unbiased estimate):")
print("Inner fold accuracies:", inner_cv_scores)
print("Outer fold accuracies:", nested_scores)
print("Mean Accuracy:", np.mean(nested_scores))
```

```
Single CV tuning (likely overfit):
Best hyperparameters: {'bootstrap': True, 'max_depth': 34, 'max_feature
s': np.float64(0.13017919126220145), 'min_samples_leaf': 6, 'min_sample
s_split': 3, 'n_estimators': 59}
Validation Accuracy: 0.6428571428571429
Test Accuracy: 0.43333333333333335

Nested CV (unbiased estimate):
Inner fold accuracies: [0.6001899335232669, 0.600664767331434, 0.622981
9563152897, 0.6372269705603039, 0.7378917378917379]
Outer fold accuracies: [0.5, 0.65, 0.45, 0.65, 0.5]
Mean Accuracy: 0.55
```