بسم الله الرحمن الرحيم

Hyperparameter Tuning

مقدمة سريعة :

Hyperparameter Tuning

هو عملية **اختيار أفضل القيم للـ Hyperparameters** (المعاملات اللي بنحددها قبل التدريب ومش بتتغير أثناءه) علشان نحسّن أداء النموذج.

ليه بنعمله؟

- لأن الـ Hyperparameters بتأثر بشكل مباشر على **دقة النموذج**.
- اختيار قيم غلط ممكن يخلي الموديل underfitting أو overfitting.
- الهدف: نلاقي **أفضل combination** من القيم يدي **أعلى أداء** على بيانات الاختبار.

أمثلة على Hyperparameters

- **Learning rate** في Gradient Descent.
- **Number of trees / depth** في Decision Tree أو Random Forest.
- **C , gamma** في SVM.
- **k** في KNN.

طرق Hyperparameter Tuning

1. **Grid Search**: نجرب كل combinations الممكنة.
2. **Random Search**: نجرّب قيم عشوائية من الـ space.

3. **Bayesian Optimization / AutoML**: طرق أذكى للبحث بتتعلم من النتائج السابقة.
   Hyperparameter Tuning = عملية تجربة قيم مختلفة للـ Hyperparameters علشان نختار الأفضل، وده مهم لتحسين أداء الموديل ومنع الـ underfitting أو overfitting.

Cross Validation => لتقييم الموديل بدقة.
GridSearchCV => لتجربة كل البارامترات بدقة عالية لكن بطيء.
RandomizedSearchCV => أسرع من GridSearchCV، مناسب لو مساحة البحث كبيرة.

ليه تتعلمهم؟!

1. **Cross Validation**
   - لازم تعرفه لأنه أساس تقييم الموديل بشكل صح.(مهم في interview)
   - أي شركة أو Interview هيسألوك إزاي بتقيّم الموديل =>CV هو الإجابة الأساسية.
2. **GridSearchCV**
   - مهم جدًا لو بتتعامل مع موديل عنده Hyperparameters (زي SVM, Random Forest, XGBoost).
   - بيعلمك إزاي تختار أفضل بارامترات بدل ما تجرب عشوائي بنفسك.
3. **RandomizedSearchCV**
   - لو الداتا كبيرة أو البارامترات كتيرة => بيكون الحل العملي لأنه أسرع من GridSearchCV.

Cross Validation (CV)

الـ Cross Validation دوره الرئيسي إنك تقيّم وتقارن بين الموديلات، مش إنه يديك موديل جاهز.

هو باختصار طريقة لحل مشكلة الـ over fitting و اختيار ال model المناسب وله 6 انواع.

1.Hold Out 2.K-Fold 3.stratified k-fold 4.leave one out

5.leave P out 6.Repeated random subsampling

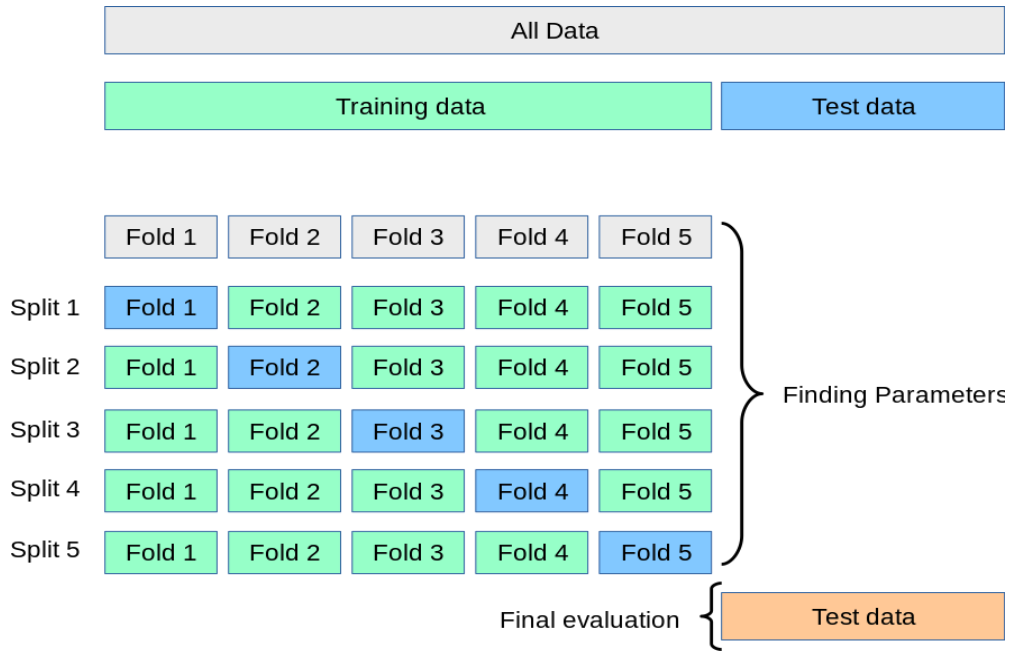
1.Hold Out

هي باختصار انك بتقسم الداتا ل train , test عادي جدا و دي اللي انت بتعملها دايما.

مشكلتها انك بتمرن مرة واحدة فقط .

2.K-Fold(cv)

بتقسم الداتا كلها ل train , test.. بتاخد train تقسمها لمجموعات مثلا 5 مجموعات . كل مجموعة فيها x,y بعدها بتمسك 4 تعمل لهم train و 1 validation و تغير validation لحد م كله يدخل validation و train و كل مرة بتحسب الدقة و في الاخر تجيب المتوسط . و لما تخلص اختبر تاني علي داتا ال test الي انت سبتها .



مشكلتها ان الداتا لازم تكون متوازنة يعني مثلا لو التارجت 0و1 بنسبة 80% 0 و 20% 1

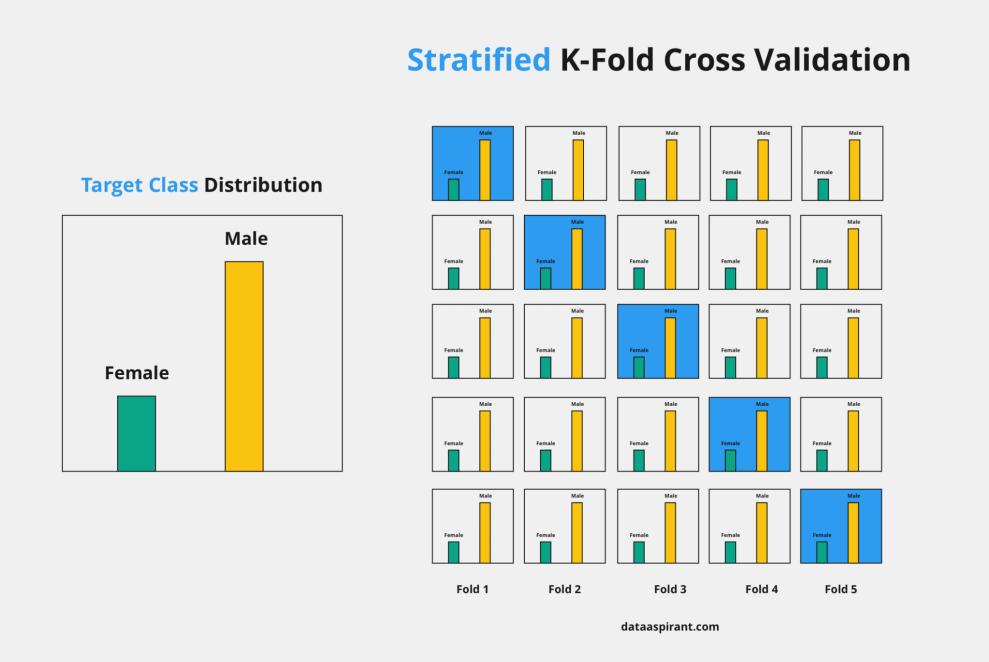
لازم برضو كل مجموعة تكون نفس النسبة ... ال k-fold مش بتقدر تعمل كده.

code</>

```
from sklearn.model_selection import KFold , cross_val_score
k = KFold(n_splits = 5 )
cvs = cross_val_score(model,x,y,cv=k)
cvs.mean()
```

3.stratified k-fold

نفس k-fold و لكن بتقسم بنفس النسب



code</>

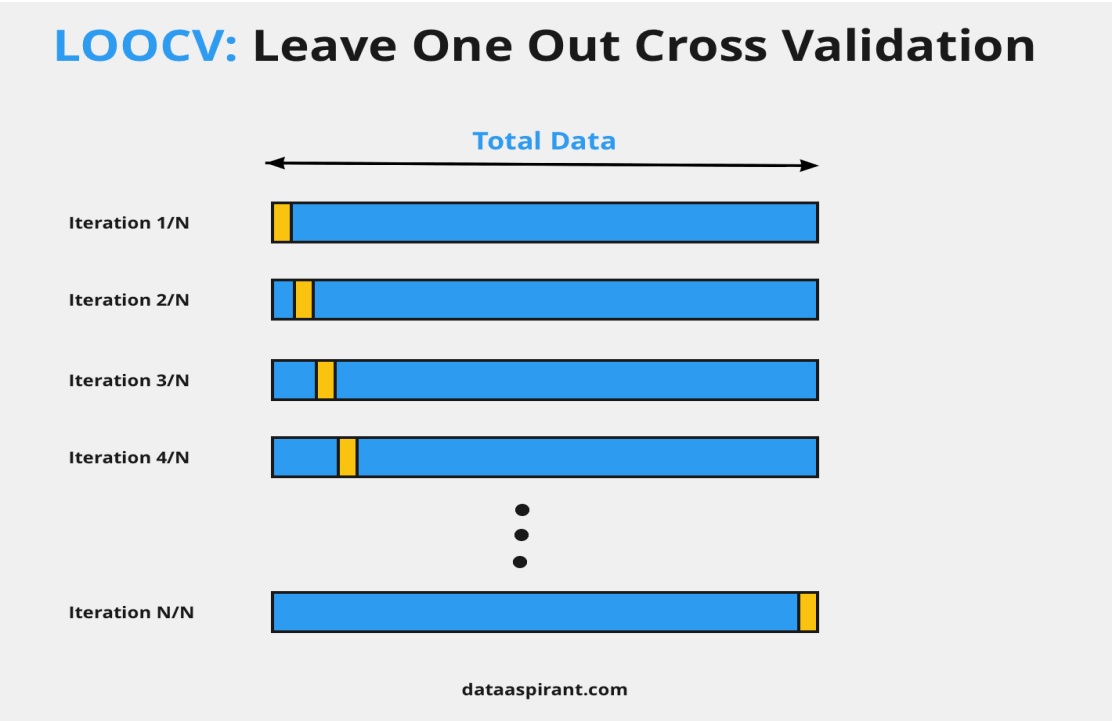
```
from sklearn.model_selection import StratifiedKFold , cross_val_score
stf = StratifiedKFold(n_splits = 5 )
cvs = cross_val_score(model,x,y,cv=stf)
cvs.mean()
```

---

4. Leave one out (cv)

1. بتقسم الداتا الكلية الي train , test.

2.لو عندك 1000 row تاخد منهم 999 تعمل بيهم train و 1 تعمل بيه validation و هكذا لحد ما كل الصفوف تدخل train و كل الصفوف تدخل validation .



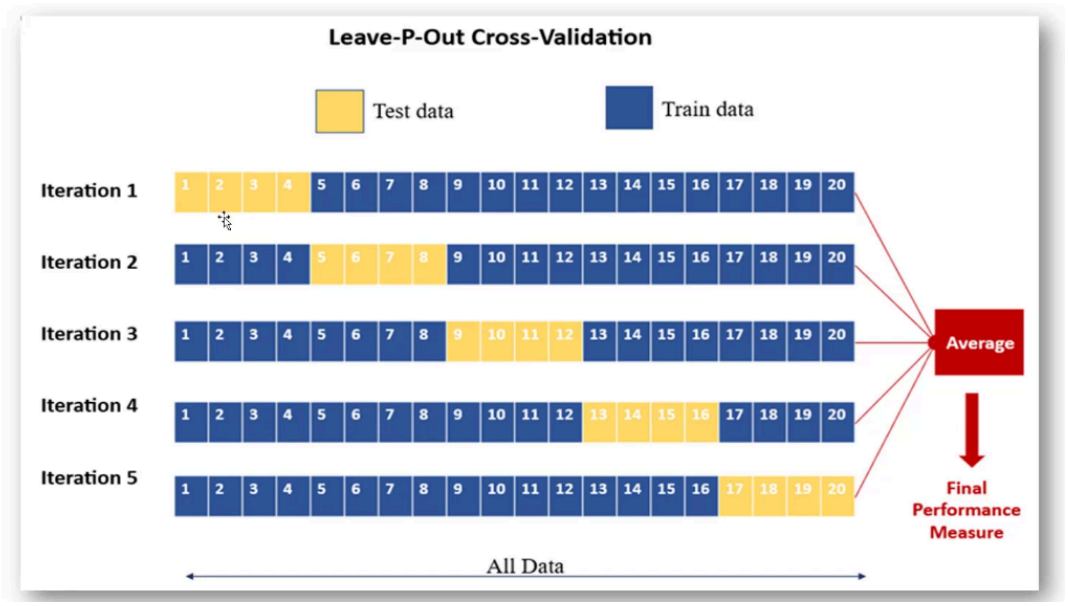
code</>

```
from sklearn.model_selection import LeaveOneOut , cross_val_score
loo = LeaveOneOut()
cvs = cross_val_score(model,x,y,cv=loo)
cvs.mean()
```

---

5. Leave P out (cv)

نفس leave one out بس بدل متاخد validation 1 هتاخد validation 3



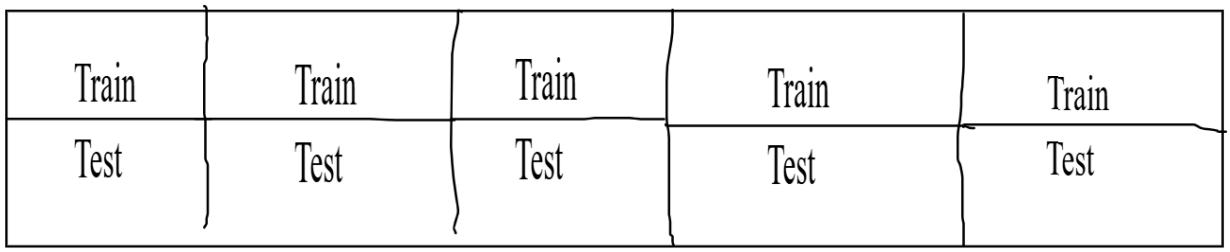
code</>

```
from sklearn.model_selection import LeavePOut , cross_val_score
# 3. إنشاء Leave-P-Out (هنا P=2 يعني هنسيب عينتين كل مرة)
lpo = LeavePOut(p=2)
cvs = cross_val_score(model,x,y,cv=lpo)
cvs.mean()
```

6.Repeated Random Subsampling

1.بتقسم الداتا الكلية ل train , test

2. تاخد ال train تقسمة ل مجموعات مثلا 5 كل مجموعة بتقسمها train , test و تمسك كل مجموعة لوحدها تمرن عليها المودل و تحسب الدقة تعمل كده بعدد المجموعات الي هو 5 و في الاخر تحسب المتوسط .




```
from sklearn.model_selection import ShuffleSplit, cross_val_score
train / 30% %70 مرة وكل مرة 30 هنا) Repeated Random Subsampling إنشاء .3 #
(test
(rs = ShuffleSplit(n_splits=30, test_size=0.3, random_state=42

Cross Validation تطبيق .4 #
('scores = cross_val_score(model, X, y, cv=rs, scoring='r2
```

---

finish cross validation

GridSearchCV

GridSearchCV is a powerful tool in machine learning, provided by the `scikit-learn` library, used for hyperparameter tuning. It systematically searches through a specified parameter grid to find the optimal combination of hyperparameters for a given model. Here's a concise overview:

Key Features :

Hyperparameter Tuning: It evaluates all possible combinations of hyperparameters to identify the best-performing set.

Cross-Validation: It uses cross-validation to ensure the model generalizes well to unseen data.

Scoring: You can specify a scoring metric (e.g., accuracy, precision, etc.) to evaluate model performance.

How It Works :

Define a model and a parameter grid (dictionary of hyperparameters and their possible values).

1=>Pass the model and parameter grid to `GridSearchCV`.

2=>Fit the `GridSearchCV` object to your training data.

3=>Retrieve the best parameters and the best model.

---

code</>

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
# Define the model
model = RandomForestClassifier()
# Define the parameter grid
param_grid = {
#هتحط هنا كل البرامتر الي عايز تظبطها و حط مجموعة قيم علشان
 نجرب و نشوف مين احسن قيمة

    'n_estimators': [50, 100, 200],

    'max_depth': [None, 10, 20],

    'min_samples_split': [2, 5, 10]

}

# نعمل GridSearchCV
grid_search = GridSearchCV(
    estimator=model,           # الموديل بتاعك
    param_grid=param_grid,     # البرامترات والقيم اللي هنجربها
    cv=5,                      # عدد الـ folds اللي هتقسم عليها الداتا (Cross
Validation)
    scoring='accuracy',        # المقياس اللي هتقيس عليه (ممكن تغيرها ل f1,
precision, recall...)
    n_jobs=-1                  # يستخدم كل أنوية المعالج لتسريع العملية
)
# نعمل fit على الداتا
grid_search.fit(X_train, y_train)
# نعرض أفضل بارامترات
print("Best Parameters:", grid_search.best_params_)
# نعرض أفضل نتيجة
print("Best Score:", grid_search.best_score_)
```

هيطبع لك افضل برامتر تستخدمة في المودل بتاعك .

Advantages :

Automates hyperparameter tuning.

Ensures robust evaluation with cross-validation.

Works with any scikit-learn estimator.

Limitations :

Computationally expensive for large parameter grids.

May not be ideal for very large datasets or complex models.

For more efficient tuning, consider **RandomizedSearchCV** or advanced techniques like **Bayesian Optimization**.

RandomizedSearchCV

RandomizedSearchCV is a method in Scikit-learn used for hyperparameter optimization. It performs a randomized search over a specified parameter distribution for an estimator, using cross-validation to evaluate the performance of different parameter combinations. This approach is particularly efficient when the search space is large, as it doesn't exhaustively test all combinations like GridSearchCV.

Key Features :

- ***Random Sampling***: Instead of testing all combinations, it samples a fixed number of parameter settings from the specified distributions.
- ***Cross-Validation***: Evaluates each parameter combination using cross-validation to ensure robust performance.
- ***Efficiency***: Reduces computational cost compared to exhaustive methods like GridSearchCV.

Important Parameters :

- `estimator`: The machine learning model to optimize (e.g., RandomForestClassifier).
- `param_distributions`: Dictionary specifying parameter distributions to sample from.
- `n_iter`: Number of parameter settings to sample.
- `scoring`: Metric to evaluate model performance (e.g., accuracy, F1-score).
- `cv`: Number of cross-validation folds.
- `random_state`: Ensures reproducibility of results.
- `n_jobs`: Number of parallel jobs to run.

code</>

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Load dataset
X, y = load_iris(return_X_y=True)

# Define model and parameter distributions
model = RandomForestClassifier()
param_distributions = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_distributions,
    n_iter=10,
    scoring='accuracy',
    cv=5,
    random_state=42,
    n_jobs=-1
)

# Fit the model
random_search.fit(X, y)

# Best parameters and score
print("Best Parameters:", random_search.best_params_)
print("Best Score:", random_search.best_score_)
```

Advantages :

- Faster than GridSearchCV for large parameter spaces.
- Allows flexibility in specifying parameter distributions (e.g., uniform, log-uniform).
- Can be parallelized for faster computation.

When to Use:

- When the parameter space is large and exhaustive search is computationally expensive.
- When you want a balance between exploration and computational efficiency.