Hyperparameter Tuning

> 💡 ***VIMP- Used everyday***

Parameter Vs Hyperparameter

Parameter

- Parameters are the internal variables of a model that are learned from the data during the training process.

🔷 **Parameters are learned from data during training.**

They define how the model makes predictions.

- In **Linear Regression (y = mx + b)**:
  - `m` (slope) and `b` (intercept) are **parameters**.
- In **Neural Networks**:
  - Weights and biases of neurons are **parameters**.
- **Key Points**:
  - **Learned during training.**
  - **Not set by the user.**
  - **Directly influence the model's predictions.**

Hyperparameter

Definition: External variables that are set **before training the model**.

- Hyperparameters are set before training and control the learning process.

- They cannot be learned from the data and need to be tuned manually or using search techniques.
- **Examples**:
  - Learning rate in Gradient Descent.
  - Number of trees in a Random Forest.
  - Regularization strength in Logistic Regression.
- **Key Points**:
  - Set by the user before training.
  - Control the learning process.
  - Do not directly influence the model's predictions but affect how the model learns.

Aspect	Parameter	Hyperparameter
Definition	Internal variables learned from data.	External variables set before training.
Set By	Model (learned during training).	User (set before training).
Examples	Weights, coefficients, node splits.	Learning rate, number of trees, regularization.
Influence	Directly influence predictions.	Control the learning process.

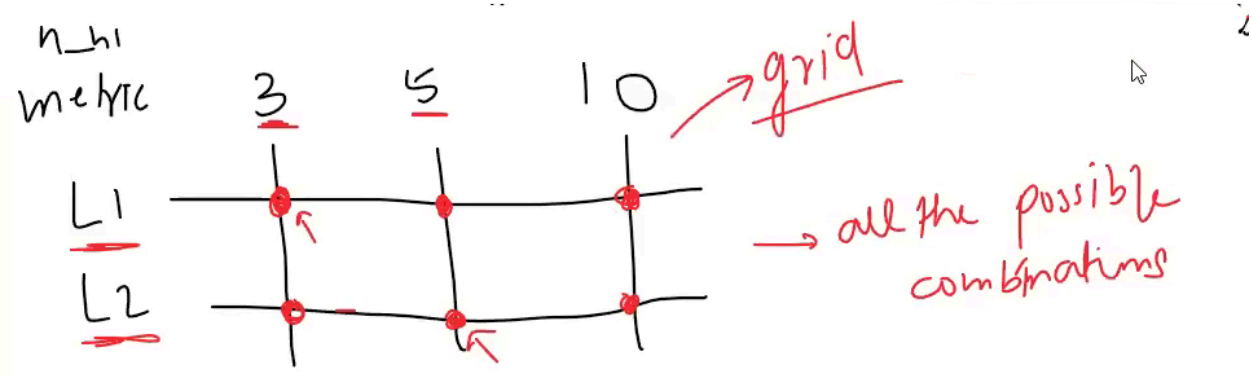
GridSearchCV

`GridSearchCV` is used to **find the optimal hyperparameter** (e.g., `alpha` for Ridge regression) by testing all combinations in the specified parameter grid (`alpha: [1, 2, 5, ..., 90]`) and selecting the best one using cross-validation (CV).

GridSearchCV refers to an algorithm that performs an exhaustive search over a specified grid
of hyperparameters, using cross-validation to determine which hyperparameter

combination
gives the best model performance.

We form a grid and test every combination:



CV = Cross validation

- You divide each model in certain parts (5, 10, etc.)
- Train the model and calculate the average.
- Ex. If CV=5, for each combination, model will be trained 5 times.

- `GridSearchCV` combines CV with **hyperparameter search**:
  1. Tests all `alpha` values.
  2. Uses 5-fold CV to evaluate each `alpha`.
  3. Selects the `alpha` with the best average validation score.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

# Step 1: Define model and parameter grid
ridge_regressor = Ridge()
parameters = {'alpha': [1, 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90]}
```

```
# Step 2: GridSearchCV tests all alphas with 5-fold CV
ridgecv = GridSearchCV(
    ridge_regressor,
    parameters,
    scoring='r2',
    cv=5  # 5-fold cross-validation
)
ridgecv.fit(X_train, y_train)

# Step 3: Best alpha and model
print("Best alpha:", ridgecv.best_params_['alpha'])  # e.g., alpha=10
print("Best MSE:", -ridgecv.best_score_)  # Convert back to positive MSE
```

Output:

{'alpha': 20}
best ridge score: 0.6917447889048314

Predict y:

```
ridge_pred=ridgecv.predict(X_test)
```

Test all values from 1 to 50

```
alphas = np.arange(1, 51)

# Initialize the Ridge regressor
ridge_regressor = Ridge()

# Set up the parameter grid
parameters = {'alpha': alphas}
```

```
ridgecv = GridSearchCV(ridge_regressor, parameters, scoring='r2', cv=5)

# Fit the model to the training data
ridgecv.fit(X_train, y_train)

# Retrieve the best alpha value
best_alpha = ridgecv.best_params_['alpha']
print(f"The best alpha value is: {best_alpha}")
```

Output:

The best alpha value is: 20

Another Example:

```
df = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/master/BostonHousing.csv')
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

```
X = df.iloc[:,:-1]
y = df.iloc[:,-1]
```

```
from sklearn.model_selection import cross_val_score,KFold
from sklearn.neighbors import KNeighborsRegressor

knn = KNeighborsRegressor()
```

```
kfold = KFold(n_splits=5, shuffle=True, random_state=1)

scores = cross_val_score(knn, X, y, cv=kfold, scoring='r2')

scores.mean()

Output:
0.47
```

Now, use GridSearchCV:

```
from sklearn.model_selection import GridSearchCV

knn = KNeighborsRegressor()

param_grid = {
    'n_neighbors':[1,3,5,7,10,12,15,17,20],
    'weights':['uniform','distance'],
    'algorithm':['ball_tree', 'kd_tree', 'brute'],
    'p':[1,2]
    }
```

```
gcv = GridSearchCV(knn, param_grid, scoring='r2', refit=True, cv=kfold, verb
ose=2)
gcv.fit(X,y)
```

`refit=True`: This means that once the best hyperparameters are found based on the cross-validation score, the model will be **refitted** on the entire training dataset using those best hyperparameters.

- `verbose=0`:
    - No output is printed. This is the default setting and means no logging of the process.
- `verbose=1`:

- Some output is printed, typically just a brief progress summary (e.g., starting and ending grid search and the best score). You'll see information about the overall progress of the search.

- `verbose=2`:
  - More detailed output is printed, including information about the current parameter combination being tested and the progress for each fold in cross-validation. It provides step-by-step updates for each hyperparameter combination.

- `verbose=3`:
  - This is the highest level of verbosity. It shows detailed information about each iteration, including the parameters being tested and the score for each fold in cross-validation. It's useful when you want to track everything happening during the grid search.

Best Parameters:

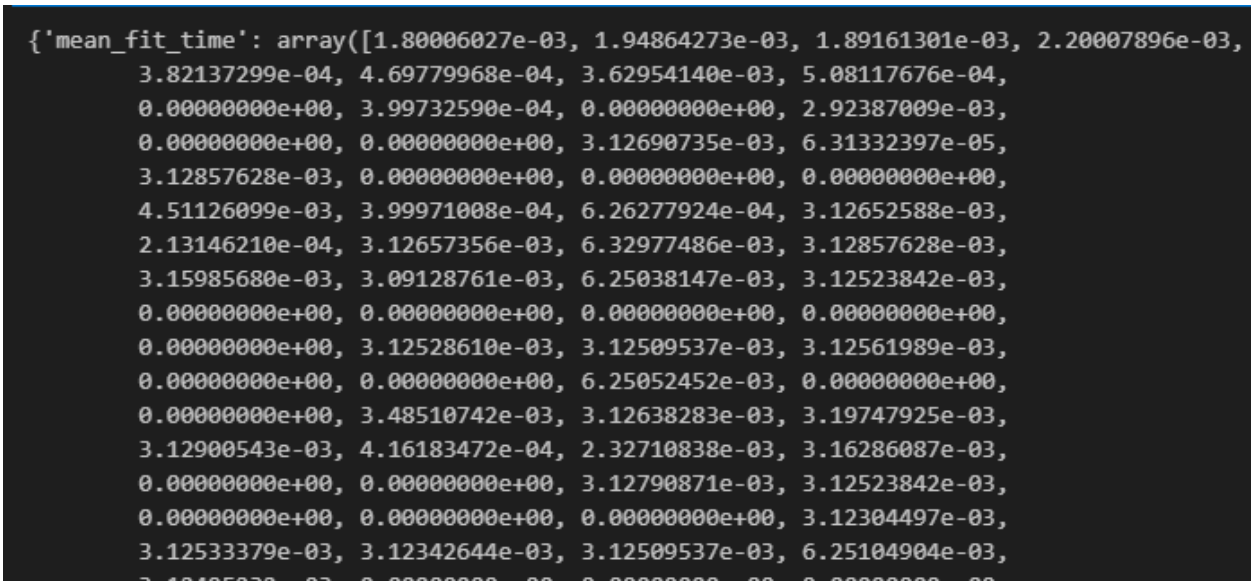
```
gcv.best_params_

Output:
{'algorithm': 'ball_tree', 'n_neighbors': 5, 'p': 1, 'weights': 'distance'}
```

```
gcv.best_score_

Output: 0.6117139367845081
```

```
gcv.cv_results_
```



```
{'mean_fit_time': array([1.80006027e-03, 1.94864273e-03, 1.89161301e-03, 2.20007896e-03,
        3.82137299e-04, 4.69779968e-04, 3.62954140e-03, 5.08117676e-04,
        0.00000000e+00, 3.99732590e-04, 0.00000000e+00, 2.92387009e-03,
        0.00000000e+00, 0.00000000e+00, 3.12690735e-03, 6.31332397e-05,
        3.12857628e-03, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        4.51126099e-03, 3.99971008e-04, 6.26277924e-04, 3.12652588e-03,
        2.13146210e-04, 3.12657356e-03, 6.32977486e-03, 3.12857628e-03,
        3.15985680e-03, 3.09128761e-03, 6.25038147e-03, 3.12523842e-03,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 3.12528610e-03, 3.12509537e-03, 3.12561989e-03,
        0.00000000e+00, 0.00000000e+00, 6.25052452e-03, 0.00000000e+00,
        0.00000000e+00, 3.48510742e-03, 3.12638283e-03, 3.19747925e-03,
        3.12900543e-03, 4.16183472e-04, 2.32710838e-03, 3.16286087e-03,
        0.00000000e+00, 0.00000000e+00, 3.12790871e-03, 3.12523842e-03,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.12304497e-03,
        3.12533379e-03, 3.12342644e-03, 3.12509537e-03, 6.25104904e-03,
```

pd.DataFrame(gcv.cv_results_)[['param_algorithm', 'param_n_neighbors', 'param_p', 'param_weights', 'mean_test_score']].sort_values('mean_test_score',ascending=False)

	param_algorithm	param_n_neighbors	param_p	param_weights	mean_test_score
81	brute	5	1	distance	0.611714
45	kd_tree	5	1	distance	0.611714
9	ball_tree	5	1	distance	0.611714
49	kd_tree	7	1	distance	0.605716
85	brute	7	1	distance	0.605716
...	...	...	...	...	...
38	kd_tree	1	2	uniform	0.331522
2	ball_tree	1	2	uniform	0.331522
75	brute	1	2	distance	0.331522
39	kd_tree	1	2	distance	0.331522
3	ball_tree	1	2	distance	0.331522

Use these parameters in new model:

```
# Train best model
best_gb = gcv.best_estimator_
y_pred_best = best_gb.predict(X_test)
```

`best_estimator_` **is a model itself which can be directly used.**

`best_gb` is the trained model with the best hyperparameters, ready for predictions.

RandomizedSearchCV

`from sklearn.model_selection import RandomizedSearchCV`

- Randomly tries out X possibilities.
  - You provide the value of X.
- **Purpose**: A technique for hyperparameter tuning that randomly samples a fixed number of hyperparameter combinations from specified distributions.

📌 Why Use It?

✅**Faster** than GridSearchCV (especially for large search spaces).

✅ Works well when we **don't know** the best hyperparameter values.

✅ Can find a good model **without testing all combinations**.

Feature	GridSearchCV	RandomizedSearchCV
Search Method	Exhaustive (tests all combinations)	Random Sampling
Speed	Slow (checks every combination)	Faster (tests only a subset)
Best for	Small parameter space	Large parameter space
Computational Cost	High	Lower
Chances of Finding Optimal	Higher	Slightly lower but close

Python Code

- Everything is same as **GridSearchCV**

```
from sklearn.model_selection import RandomizedSearchCV

rcv = RandomizedSearchCV(knn, param_grid, scoring='r2', refit=True, cv=kfold, verbose=2)

rcv.fit(X,y)
```

```
rcv.best_score_

Output: 0.6057158068725681
```

- With **GridSearchCV**, the score was **0.61**

```
rcv.best_params_

Output:{'weights': 'distance', 'p': 1, 'n_neighbors': 7, 'algorithm': 'brute'}
```

Key Parameters in RandomizedSearchCV

- 🔷 `param_distributions`: Dictionary of hyperparameters to sample from.
- 🔷 `n_iter`: Number of random combinations to test.
- 🔷 `scoring`: Metric to optimize (e.g., accuracy, RMSE).
- 🔷 `cv`: Number of cross-validation folds.
- 🔷 `random_state`: Ensures reproducibility.
- 🔷 `n_jobs`: Number of CPU cores to use (`-1` = all cores).

When to Use?

✔️ **Use GridSearchCV** when the hyperparameter space is **small**.

✔️ **Use RandomizedSearchCV** when the hyperparameter space is **large**, and **speed is a concern**.