

# Introducing Grid Search

HYPERPARAMETER TUNING IN PYTHON



**Alex Scriven**  
Data Scientist

# Automating 2 Hyperparameters

Your previous work:

```
neighbors_list = [3, 5, 10, 20, 50, 75]
accuracy_list = []
for test_number in neighbors_list:
    model = KNeighborsClassifier(n_neighbors=test_number)
    predictions = model.fit(X_train, y_train).predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    accuracy_list.append(accuracy)
```

Which we then collated in a dataframe to analyse.

# Automating 2 Hyperparameters

What about testing values of 2 hyperparameters?

Using a GBM algorithm:

- `learn_rate` `[0.001, 0.01, 0.05]`
- `max_depth` `[4, 6, 8, 10]`

We could use a (*nested*) for loop!

# Automating 2 Hyperparameters

Firstly a model creation function:

```
def gbm_grid_search(learn_rate, max_depth):  
    model = GradientBoostingClassifier(  
        learning_rate=learn_rate,  
        max_depth=max_depth)  
    predictions = model.fit(X_train, y_train).predict(X_test)  
    return([learn_rate, max_depth, accuracy_score(y_test, predictions)])
```

# Automating 2 Hyperparameters

Now we can loop through our lists of hyperparameters and call our function:

```
results_list = []

for learn_rate in learn_rate_list:
    for max_depth in max_depth_list:
        results_list.append(gbm_grid_search(learn_rate, max_depth))
```

# Automating 2 Hyperparameters

We can put these results into a DataFrame as well and print out:

```
results_df = pd.DataFrame(results_list, columns=['learning_rate', 'max_depth', 'accuracy'])  
print(results_df)
```

learning_rate	max_depth	accuracy
0.001	4	0.75
0.001	6	0.75
0.01	4	0.77
0.01	6	0.76

# How many models?

There were many more models built by adding more hyperparameters and values.

- The relationship is not linear, it is exponential
- One more value of a hyperparameter is not *just* one model
- 5 for Hyperparameter 1 and 10 for Hyperparameter 2 is 50 models!

What about cross-validation?

- 10-fold cross-validation would make  $50 \times 10 = 500$  models!

# From 2 to N hyperparameters

What about adding more hyperparameters?

We could nest our loop!

```
# Adjust the list of values to test
learn_rate_list = [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5]
max_depth_list = [4, 6, 8, 10, 12, 15, 20, 25, 30]
subsample_list = [0.4, 0.6, 0.7, 0.8, 0.9]
max_features_list = ['auto', 'sqrt']
```



# From 2 to N hyperparameters

Adjust our function:

```
def gbm_grid_search(learn_rate, max_depth, subsample, max_features):  
    model = GradientBoostingClassifier(  
        learning_rate=learn_rate,  
        max_depth=max_depth,  
        subsample=subsample,  
        max_features=max_features)  
    predictions = model.fit(X_train, y_train).predict(X_test)  
    return([learn_rate, max_depth, accuracy_score(y_test, predictions)])
```

# From 2 to N hyperparameters

Adjusting our for loop (nesting):

```
for learn_rate in learn_rate_list:
    for max_depth in max_depth_list:
        for subsample in subsample_list:
            for max_features in max_features_list:
                results_list.append(gbm_grid_search(learn_rate,max_depth,
                                                    subsample,max_features))
results_df = pd.DataFrame(results_list, columns=['learning_rate',
                                                'max_depth', 'subsample', 'max_features', 'accuracy'])
print(results_df)
```

# From 2 to N hyperparameters

How many models now?

- $7 \times 9 \times 5 \times 2 = 630$  (6,300 if cross-validated!)

We can't keep nesting forever!

Plus, what if we wanted:

- Details on training times & scores
- Details on cross-validation scores

# Introducing Grid Search

Let's create a grid:

- Down the left all values of `max_depth`
- Across the top all values of `learning_rate`

learn_rate				
max_depth		0.001	0.01	0.05
	4	(4 , 0.001)	(4 , 0.01)	(4 , 0.05)
	6	(6 , 0.001)	(6 , 0.01)	(6 , 0.05)
	8	(8 , 0.001)	(8 , 0.01)	(8 , 0.05)

# Introducing Grid Search

Working through each cell on the grid:

learn_rate				
max_depth		0.001	0.01	0.05
	4	(4 , 0.001)	(4 , 0.01)	(4 , 0.05)
	6	(6 , 0.001)	(6 , 0.01)	(6 , 0.05)
	8	(8 , 0.001)	(8 , 0.01)	(8 , 0.05)

(4,0.001) is equivalent to making an estimator like so:

```
GradientBoostingClassifier(max_depth=4, learning_rate=0.001)
```

# Grid Search Pros & Cons

Some advantages of this approach:

Advantages:

- You don't have to write thousands of lines of code
- Finds the best model within the grid (\*special note here!)
- Easy to explain

# Grid Search Pros & Cons

Some disadvantages of this approach:

- Computationally expensive! Remember how quickly we made 6,000+ models?
- It is 'uninformed'. Results of one model don't help creating the next model.

We will cover 'informed' methods later!

# Let's practice!

**HYPERPARAMETER TUNING IN PYTHON**



# Grid Search with Scikit Learn

HYPERPARAMETER TUNING IN PYTHON



**Alex Scriven**  
Data Scientist

# GridSearchCV Object

Introducing a `GridSearchCV` object:

```
sklearn.model_selection.GridSearchCV(  
    estimator,  
    param_grid, scoring=None, fit_params=None,  
    n_jobs=None, refit=True, cv='warn',  
    verbose=0, pre_dispatch='2*n_jobs',  
    error_score='raise-deprecating',  
    return_train_score='warn')
```

# Steps in a Grid Search

Steps in a Grid Search:

1. An algorithm to tune the hyperparameters. (Sometimes called an 'estimator')
2. Defining which hyperparameters we will tune
3. Defining a range of values for each hyperparameter
4. Setting a cross-validation scheme; and
5. Define a score function so we can decide which square on our grid was 'the best'.
6. Include extra useful information or functions

# GridSearchCV Object Inputs

The important inputs are:

- `estimator`
- `param_grid`
- `cv`
- `scoring`
- `refit`
- `n_jobs`
- `return_train_score`

# GridSearchCV 'estimator'

The `estimator` input:

- Essentially our algorithm
- You have already worked with KNN, Random Forest, GBM, Logistic Regression

Remember:

- Only one estimator per GridSearchCV object

# GridSearchCV 'param\_grid'

The `param_grid` input:

- Setting which hyperparameters and values to test

Rather than a list:

```
max_depth_list = [2, 4, 6, 8]
min_samples_leaf_list = [1, 2, 4, 6]
```

This would be:

```
param_grid = {'max_depth': [2, 4, 6, 8],
              'min_samples_leaf': [1, 2, 4, 6]}
```

# GridSearchCV 'param\_grid'

The `param_grid` input:

Remember: The keys in your `param_grid` dictionary must be valid hyperparameters.

For example, for a Logistic regression estimator:

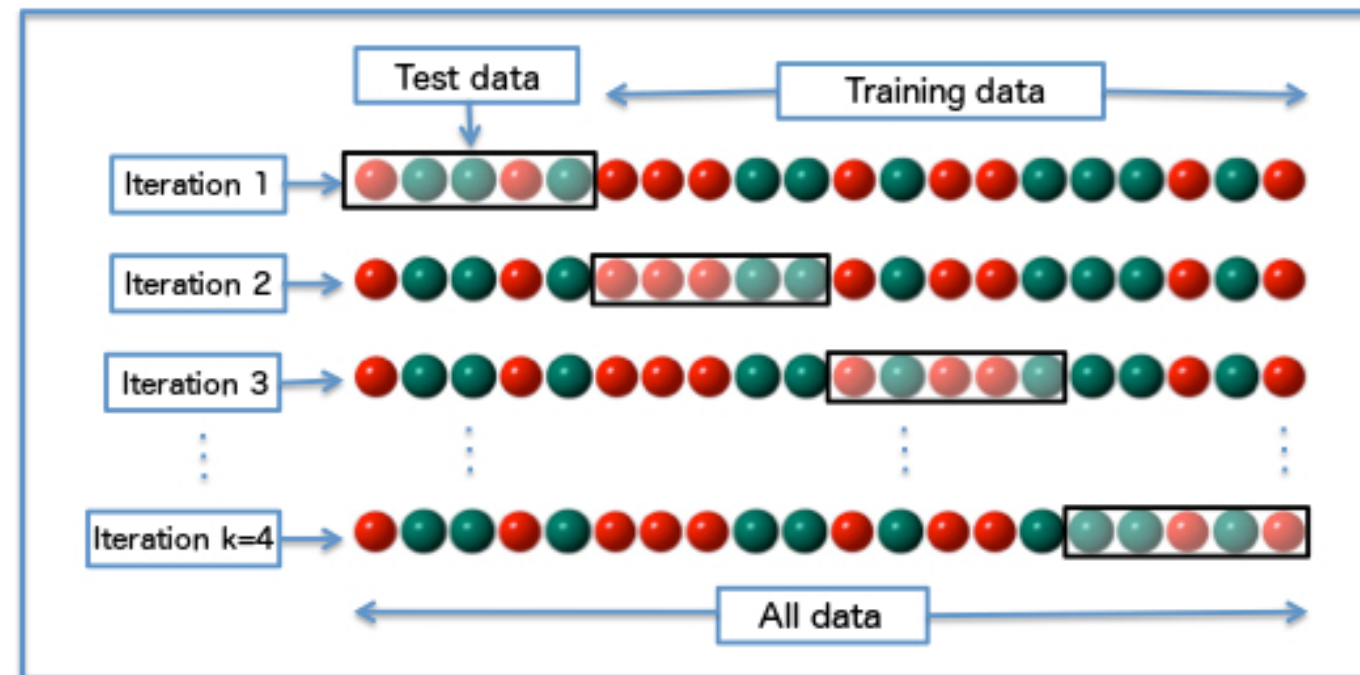
```
# Incorrect
param_grid = {'C': [0.1, 0.2, 0.5],
              'best_choice': [10, 20, 50]}
```

```
ValueError: Invalid parameter best_choice for estimator LogisticRegression
```

# GridSearchCV 'cv'

The `cv` input:

- Choice of how to undertake cross-validation
- Using an integer undertakes k-fold cross validation where 5 or 10 is usually standard





# GridSearchCV 'scoring'

The `scoring` input:

- Which score to use to choose the best grid square (model)
- Use your own or Scikit Learn's `metrics` module

You can check all the built in scoring functions this way:

```
from sklearn import metrics  
sorted(metrics.SCORERS.keys())
```

# GridSearchCV 'refit'

The `refit` input:

- Fits the best hyperparameters to the training data
- Allows the `GridSearchCV` object to be used as an estimator (for prediction)
- A very handy option!

# GridSearchCV 'n\_jobs'

The `n_jobs` input:

- Assists with parallel execution
- Allows multiple models to be created at the same time, rather than one after the other

Some handy code:

```
import os  
print(os.cpu_count())
```

Careful using all your cores for modelling if you want to do other work!

# GridSearchCV 'return\_train\_score'

The `return_train_score` input:

- Logs statistics about the training runs that were undertaken
- Useful for analyzing bias-variance trade-off but adds computational expense.
- Does not assist in picking the best model, only for analysis purposes

# Building a GridSearchCV object

Building our own GridSearchCV Object:

```
# Create the grid
param_grid = {'max_depth': [2, 4, 6, 8], 'min_samples_leaf': [1, 2, 4, 6]}

#Get a base classifier with some set parameters.
rf_class = RandomForestClassifier(criterion='entropy', max_features='auto')
```

# Building a GridSearchCv Object

Putting the pieces together:

```
grid_rf_class = GridSearchCV(  
    estimator = rf_class,  
    param_grid = parameter_grid,  
    scoring='accuracy',  
    n_jobs=4,  
    cv = 10,  
    refit=True,  
    return_train_score=True)
```

# Using a GridSearchCV Object

Because we set `refit` to `True` we can directly use the object:

```
#Fit the object to our data
grid_rf_class.fit(X_train, y_train)

# Make predictions
grid_rf_class.predict(X_test)
```

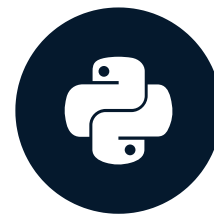
# Let's practice!

**HYPERPARAMETER TUNING IN PYTHON**



# Understanding a grid search output

HYPERPARAMETER TUNING IN PYTHON



**Alex Scriven**  
Data Scientist

# Analyzing the output

Let's analyze the GridSearchCV outputs.

Three different groups for the GridSearchCV properties;

- A results log
  - `cv_results_`
- The best results
  - `best_index_` , `best_params_` & `best_score_`
- 'Extra information'
  - `scorer_` , `n_splits_` & `refit_time_`

# Accessing object properties

Properties are accessed using the dot notation.

For example:

```
grid_search_object.property
```

Where `property` is the actual property you want to retrieve

# The `.cv_results_` property

The `cv_results_` property:

Read this into a DataFrame to print and analyze:

```
cv_results_df = pd.DataFrame(grid_rf_class.cv_results_)

print(cv_results_df.shape)
```

`(12, 23)`

- The 12 rows for the 12 squares in our grid or 12 models we ran

# The .cv\_results\_ 'time' columns

The `time` columns refer to the time it took to fit (and score) the model.

Remember how we did a 5-fold cross-validation? This ran 5 times and stored the average and standard deviation of the times it took in seconds.

	mean_fit_time	std_fit_time	mean_score_time	std_score_time
0	0.321069	0.007236	0.015008	0.000871
1	0.678216	0.066385	0.034155	0.003767
2	0.939865	0.009502	0.055868	0.004148
3	0.296547	0.006261	0.017990	0.002803
4	0.686065	0.016163	0.040048	0.001304
5	1.097201	0.006327	0.057136	0.004468
6	0.416973	0.085533	0.021157	0.003901
7	0.788864	0.021954	0.042638	0.004802
8	1.198466	0.054694	0.049674	0.006884
9	0.398824	0.027500	0.025307	0.009473
10	0.719588	0.019231	0.035629	0.005712
11	0.847477	0.036584	0.029104	0.005220

# The .cv\_results\_ 'param\_' columns

The `param_` columns store the parameters it tested on that row, one column per parameter

param_max_depth	param_min_samples_leaf	param_n_estimators
10	1	100
10	1	200
10	2	100
10	2	200
10	2	300

# The .cv\_results\_ 'param' column

The `params` column contains dictionary of all the parameters:

```
pd.set_option("display.max_colwidth", -1)
print(cv_results_df.loc[:, "params"])
```

params
<code>{'max_depth': 10, 'min_samples_leaf': 1, 'n_estimators': 100}</code>
<code>{'max_depth': 10, 'min_samples_leaf': 1, 'n_estimators': 200}</code>
<code>{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 100}</code>
<code>{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 200}</code>
<code>{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 300}</code>

# The .cv\_results\_ 'test\_score' columns

The `test_score` columns contain the scores on our test set for each of our cross-folds as well as some summary statistics:

split0_test_score	split1_test_score	...	mean_test_score	std_test_score
0.72820401	0.7859811	...	0.76010401	0.02995142
0.73539669	0.7963085	...	0.76590708	0.02721413
0.72929381	0.78686003	...	0.7718143	0.02775648
0.72820401	0.78554164	...	0.77044862	0.02794597
0.72885789	0.78795869	...	0.77122424	0.03288053



# The .cv\_results\_ 'rank\_test\_score' column

The rank column, ordering the `mean_test_score` from best to worst:

rank_test_score
9
4
1
3
2

# Extracting the best row

We can select the best grid square easily from `cv_results_` using the `rank_test_score` column

```
best_row = cv_results_df[cv_results_df["rank_test_score"] == 1]
print(best_row)
```

mean_fit_time	...	params	...	mean_test_score	rank_test_score
0.97765441	...	<code>{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 200}</code>	...	0.7718143	1

# The `.cv_results_` 'train\_score' columns

The `test_score` columns are then repeated for the `training_scores` .

Some important notes to keep in mind:

- `return_train_score` must be `True` to include training scores columns.
- There is no ranking column for the training scores, as we only care about test set performance

# The best grid square

Information on the best grid square is neatly summarized in the following three properties:

- `best_params_` , the dictionary of parameters that gave the best score.
- `best_score_` , the actual best score.
- `best_index_` , the row in our `cv_results_.rank_test_score` that was the best.

# The `best_estimator_` property

The `best_estimator_` property is an estimator built using the best parameters from the grid search.

For us this is a Random Forest estimator:

```
type(grid_rf_class.best_estimator_)
```

```
sklearn.ensemble.forest.RandomForestClassifier
```

We could also directly use this object as an estimator if we want!

# The `best_estimator_` property

```
print(grid_rf_class.best_estimator_)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',  
                        max_depth=10, max_features='auto', max_leaf_nodes=None,  
                        min_impurity_decrease=0.0, min_impurity_split=None,  
                        min_samples_leaf=1, min_samples_split=2,  
                        min_weight_fraction_leaf=0.0, n_estimators=300, n_jobs=None,  
                        oob_score=False, random_state=None, verbose=0,  
                        warm_start=False)
```

# Extra information

Some extra information is available in the following properties:

- `scorer_`

What scorer function was used on the held out data. (we set it to AUC)

- `n_splits_`

How many cross-validation splits. (We set to 5)

- `refit_time_`

The number of seconds used for refitting the best model on the whole dataset.

# Let's practice!

**HYPERPARAMETER TUNING IN PYTHON**