https://www.youtube.com/watch?v=HdlDYng8g9s&list=PLeo1K3hjS3uvCeTYTeyfe0-rN5r8zn9rw&index=17

In this tutorial for beginners we will look into,

1) how to hyper tune machine learning model paramers

2) choose best model for given machine learning problem

We will start by comparing traditional train\_test\_split approach with k fold cross validation.

Then we will see how GridSearchCV helps run K Fold cross validation with its convenient api.

GridSearchCV helps find best parameters that gives maximum performance.

RandomizedSearchCV is another class in sklearn library that does same thing as GridSearchCV but without running exhaustive search, this helps with computation time and resources.

We will also see how to find best model among all the classification algorithm using GridSearchCV. In the end we have interesting exercise for you to solve.

**Use RandomizedSearchCV to reduce number of iterations and with random combination of parameters.**

**This is useful when you have too many parameters to try and your training time is longer.**

**It helps reduce the cost of computation**

**from** sklearn **import** svm, datasets

iris **=** datasets**.**load\_iris()

**import** pandas **as** pd

df **=** pd**.**DataFrame(iris**.**data,columns**=**iris**.**feature\_names)

df['flower'] **=** iris**.**target

df['flower'] **=** df['flower']**.**apply(**lambda** x: iris**.**target\_names[x])



**Approach 1: Use train\_test\_split and manually tune parameters by trial and error**

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(iris**.**data, iris**.**target, test\_size**=**0.3)

model **=** svm**.**SVC(kernel**=**'rbf',C**=**30,gamma**=**'auto')

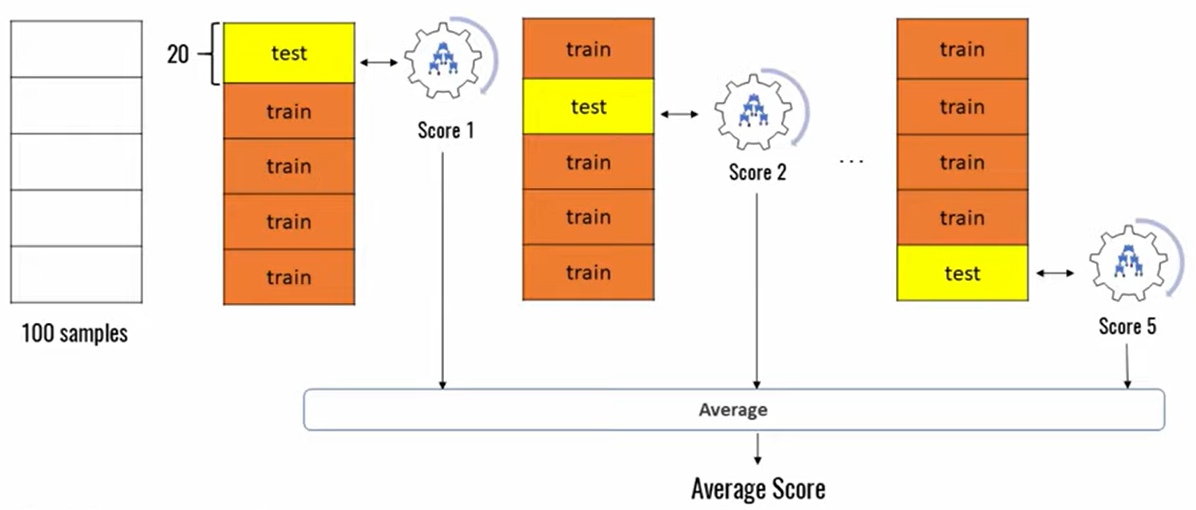
model**.**fit(X\_train,y\_train)

model**.**score(X\_test, y\_test)

0.9111

**Approach 2: Use K Fold Cross validation**

**Manually try suppling models with different parameters to cross\_val\_score function with 5 fold cross validation**



cross\_val\_score(svm**.**SVC(kernel**=**'linear',C**=**10,gamma**=**'auto'),iris**.**data, iris**.**target, cv**=**5)

Output:

array([1. , 1. , 0.9 , 0.96666667, 1. ])

cross\_val\_score(svm**.**SVC(kernel**=**'rbf',C**=**10,gamma**=**'auto'),iris**.**data, iris**.**target, cv**=**5)

Output

array([0.96666667, 1. , 0.96666667, 0.96666667, 1. ])

cross\_val\_score(svm**.**SVC(kernel**=**'rbf',C**=**20,gamma**=**'auto'),iris**.**data, iris**.**target, cv**=**5)

Output:

array([0.96666667, 1. , 0.9 , 0.96666667, 1. ])

**Above approach is tiresome and very manual. We can use for loop as an alternative**

kernels **=** ['rbf', 'linear']

C **=** [1,10,20]

avg\_scores **=** {}

**for** kval **in** kernels:

**for** cval **in** C:

cv\_scores **=** cross\_val\_score(svm**.**SVC(kernel**=**kval,C**=**cval,gamma**=**'auto'),

iris**.**data, iris**.**target, cv**=**5)

avg\_scores[kval **+** '\_' **+** str(cval)] **=** np**.**average(cv\_scores)

avg\_scores

Output

{'rbf\_1': 0.9800000000000001,

'rbf\_10': 0.9800000000000001,

'rbf\_20': 0.9666666666666668,

'linear\_1': 0.9800000000000001,

'linear\_10': 0.9733333333333334,

'linear\_20': 0.9666666666666666}

**From above results we can say that rbf with C=1 or 10 or linear with C=1 will give best performance**

**Approach 3: Use GridSearchCV**

**GridSearchCV does exactly same thing as for loop above but in a single line of code**

**from** sklearn.model\_selection **import** GridSearchCV

clf **=** GridSearchCV(svm**.**SVC(gamma**=**'auto'), {

'C': [1,10,20],

'kernel': ['rbf','linear']

}, cv**=**5, return\_train\_score**=False**)

clf**.**fit(iris**.**data, iris**.**target)

clf**.**cv\_results\_

df **=** pd**.**DataFrame(clf**.**cv\_results\_)

df[['param\_C','param\_kernel','mean\_test\_score']]

Output:

|  | **param\_C** | **param\_kernel** | **mean\_test\_score** |
| --- | --- | --- | --- |
| **0** | 1 | rbf | 0.980000 |
| **1** | 1 | linear | 0.980000 |
| **2** | 10 | rbf | 0.980000 |
| **3** | 10 | linear | 0.973333 |
| **4** | 20 | rbf | 0.966667 |
| **5** | 20 | linear | 0.966667 |

:

clf**.**best\_params\_

Output

{'C': 1, 'kernel': 'rbf'}

clf**.**best\_score\_

Output

0.98

dir(clf)

Out[78]:

['\_\_abstractmethods\_\_',

'\_\_class\_\_',

'\_\_delattr\_\_',

'\_\_dict\_\_',

'\_\_dir\_\_',

'\_\_doc\_\_',

'\_\_eq\_\_',

'\_\_format\_\_',

'\_\_ge\_\_',

'\_\_getattribute\_\_',

'\_\_getstate\_\_',

'\_\_gt\_\_',

'\_\_hash\_\_',

'\_\_init\_\_',

'\_\_init\_subclass\_\_',

'\_\_le\_\_',

'\_\_lt\_\_',

'\_\_module\_\_',

'\_\_ne\_\_',

'\_\_new\_\_',

'\_\_reduce\_\_',

'\_\_reduce\_ex\_\_',

'\_\_repr\_\_',

'\_\_setattr\_\_',

'\_\_setstate\_\_',

'\_\_sizeof\_\_',

'\_\_str\_\_',

'\_\_subclasshook\_\_',

'\_\_weakref\_\_',

'\_abc\_impl',

'\_check\_is\_fitted',

'\_estimator\_type',

'\_format\_results',

'\_get\_param\_names',

'\_run\_search',

'best\_estimator\_',

'best\_index\_',

'best\_params\_',

'best\_score\_',

'classes\_',

'cv',

'cv\_results\_',

'decision\_function',

'error\_score',

'estimator',

'fit',

'fit\_params',

'get\_params',

'iid',

'inverse\_transform',

'multimetric\_',

'n\_jobs',

'n\_splits\_',

'param\_grid',

'pre\_dispatch',

'predict',

'predict\_log\_proba',

'predict\_proba',

'refit',

'refit\_time\_',

'return\_train\_score',

'score',

'scorer\_',

'scoring',

'set\_params',

'transform',

'verbose']

**Use RandomizedSearchCV to reduce number of iterations and with random combination of parameters. This is useful when you have too many parameters to try and your training time is longer. It helps reduce the cost of computation**

**from** sklearn.model\_selection **import** RandomizedSearchCV

rs **=** **RandomizedSearchCV**(svm**.**SVC(gamma**=**'auto'), {

'C': [1,10,20],

'kernel': ['rbf','linear']

},

cv**=**5,

return\_train\_score**=False**,

n\_iter**=**2

)

rs**.**fit(iris**.**data, iris**.**target)

pd**.**DataFrame(rs**.**cv\_results\_)[['param\_C','param\_kernel','mean\_test\_score']]

Out[102]:

|  | **param\_C** | **param\_kernel** | **mean\_test\_score** |
| --- | --- | --- | --- |
| **0** | 10 | rbf | 0.98 |
| **1** | 1 | linear | 0.98 |

**How about different models with different hyperparameters?**

**from** sklearn **import** svm

**from** sklearn.ensemble **import** RandomForestClassifier

**from** sklearn.linear\_model **import** LogisticRegression

model\_params **=** {

'svm': {

'model': svm**.**SVC(gamma**=**'auto'),

'params' : {

'C': [1,10,20],

'kernel': ['rbf','linear']

}

},

'random\_forest': {

'model': RandomForestClassifier(),

'params' : {

'n\_estimators': [1,5,10]

}

},

'logistic\_regression' : {

'model': LogisticRegression(solver**=**'liblinear',multi\_class**=**'auto'),

'params': {

'C': [1,5,10]

}

}

}

scores **=** []

**for** model\_name, mp **in** model\_params**.**items():

clf **=** GridSearchCV(mp['model'], mp['params'], cv**=**5, return\_train\_score**=False**)

clf**.**fit(iris**.**data, iris**.**target)

scores**.**append({

'model': model\_name,

'best\_score': clf**.**best\_score\_,

'best\_params': clf**.**best\_params\_

})

df **=** pd**.**DataFrame(scores,columns**=**['model','best\_score','best\_params'])

df

Out[93]:

|  | **model** | **best\_score** | **best\_params** |
| --- | --- | --- | --- |
| **0** | svm | 0.980000 | {'C': 1, 'kernel': 'rbf'} |
| **1** | random\_forest | 0.953333 | {'n\_estimators': 5} |
| **2** | logistic\_regression | 0.966667 | {'C': 5} |

**Based on above, I can conclude that SVM with C=1 and kernel='rbf' is the best model for solving my problem of iris flower classification**