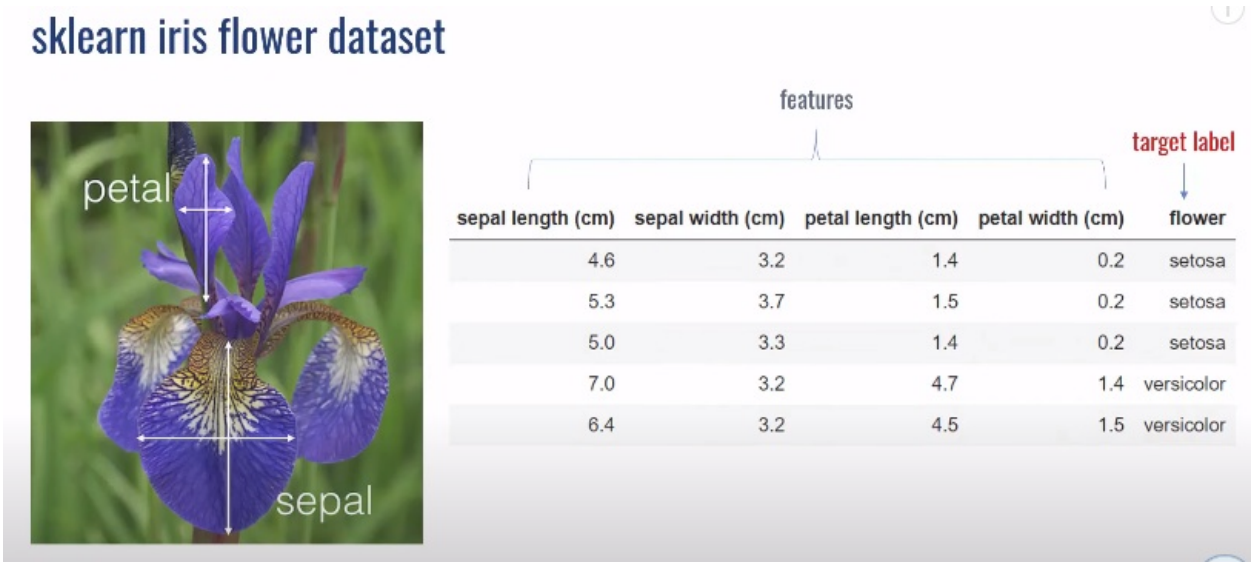
Hyper Parameter Tuning

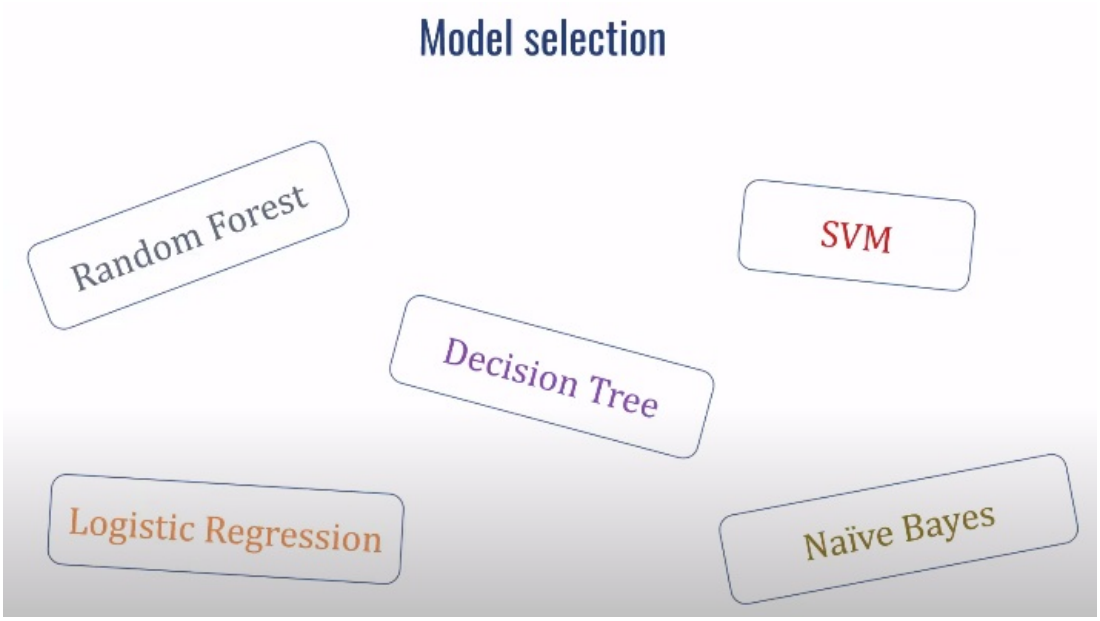
Here we are going to talk about how to chose the best model for ur given ML problem and how to do **Hyper Parameter Tuning**.

Lets say u are trying to classify Iris flower dataset where on the basis of petal, sepal length and width we are trying to predict what type of flower is it



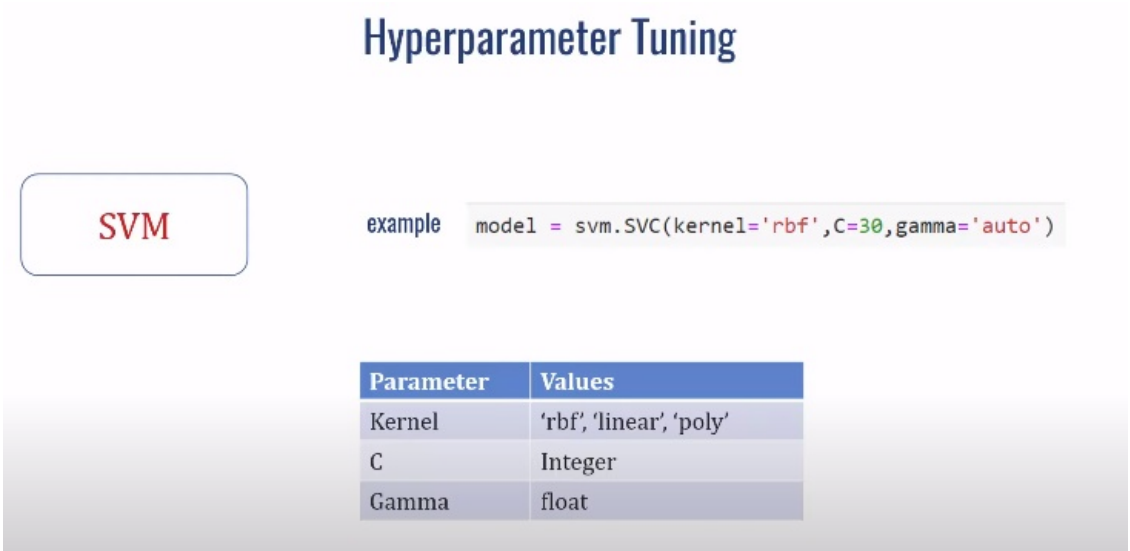
sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	flower
4.6	3.2	1.4	0.2	setosa
5.3	3.7	1.5	0.2	setosa
5.0	3.3	1.4	0.2	setosa
7.0	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor

Now, the first question that arises is which model should i use?



Now, lets say that **SVM** is the model that u want to use, the problem doesnt end there, now u have to have **Hyper Parameter**, like what kind of kernel, C or Gamma should we use? there are so many values to chose from.

The process of chhosing the optimal parameter is called **Hyper Parameter Tuning**



Parameter	Values
Kernel	'rbf', 'linear', 'poly'
C	Integer
Gamma	float

In [2]:
```
from sklearn import svm, datasets
import pandas as pd
```

In [4]:
```
iris = datasets.load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
```

In [5]:
```
df['flower'] = iris.target
df['flower'] = df['flower'].apply(lambda x: iris.target_names[x])
df.head()
```

Out[5]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	flower
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Approach 1: Use train_test_split and manually tune parameters by trial and error

So we have loaded Iris flower dataset, Now the traditional approach that we can take to solve this problem is we use Train Test Splitting

In [6]:
```
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)
```

And then lets say we first try the SVM model, so first we will see how to do HyperParameter tuning and then will look into how to choose the model so just assume that u are going to use SVM model

In [7]:
```
model = svm.SVC(kernel='rbf',C=30,gamma='auto')
model.fit(X_train,y_train)
model.score(X_test, y_test)
```

Out[7]:
```
0.9777777777777777
```

Above we randomly initialize with some parameters, since we dont know what is the best Parameters.

The issue here is that base on your Train and Test set the score might vary, right now our score is 97% but if we exevute it again it will change so we cant rely on this mthod as the score keeps changing base on our sample, so for that reason we use **K FOLD Cross Validation**

Approach 2: Use K Fold Cross validation

Below what we will do is try cross_val_score for 5 folds and try this method on different values of "Kernel" and "C"

In [14]:
```
from sklearn.model_selection import cross_val_score
```

In [15]:
```
cross_val_score(svm.SVC(kernel='linear',C=10,gamma='auto'),iris.data, iris.target, cv=5)
```

Out[15]:
```
array([1.        , 1.        , 0.9       , 0.96666667, 1.        ])
```

In [16]:
```
cross_val_score(svm.SVC(kernel='rbf',C=10,gamma='auto'),iris.data, iris.target, cv=5)
```

Out[16]:
```
array([0.96666667, 1.        , 0.96666667, 0.96666667, 1.        ])
```

In [17]:
```
cross_val_score(svm.SVC(kernel='rbf',C=20,gamma='auto'),iris.data, iris.target, cv=5)
```

Out[17]:
```
array([0.96666667, 1.        , 0.9       , 0.96666667, 1.        ])
```

You can see above we got 5 values for different parameters, u can find the average of these values and based on that you can determine the optimal value of these Parameters.

But u can see that this method is very manual and repitative cuz there are so many values you can supply as different combinations so u will have to make alot of cross validation to try out different combinations.

So the other approach we can take is we can just run a for loop

In [19]:
```
import numpy as np
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
```

Out[19]:
```
{'rbf_1': 0.9800000000000001,
 'rbf_10': 0.9800000000000001,
 'rbf_20': 0.9666666666666668,
 'linear_1': 0.9800000000000001,
 'linear_10': 0.9733333333333334,
 'linear_20': 0.9666666666666666}
```

Approach 3: Use GridSearchCV

As above we can see using this way we can also find the best optimal score but u can see that this approach also has some issues like if we have 4 parameters then will have to run like 4 loops then it will be too many itertions and its just not convenient.

Luckily SKLearn provides an API called **GridSearchCV** which will do the exact same thing, i will do the exact same thing as shown in the for loop above but we will be able to do that in a single line of code

In [22]:
```
from sklearn.model_selection import GridSearchCV
```

Now the first parameter we pass in GridSearchCV is the model so for our example we use **SVM** and we apply gamma value to be auto, then the Second parameter is very important, this is your parameter grid, in parameter grid, u will say i want the value of "C" to be 1, 10 and 20, these are the different value that we are going to try, and then "kernel" and in kernel we want to try "rbf" and "linear".

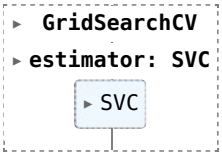
There are other parameters as well in **GridSearchCV** for example "cv" as third parameter to define how many Cross Validation u want to run, GridSearchSV uses cross validation, its just that we have the for loop step here in a line of code

In [23]:
```
clf = GridSearchCV(svm.SVC(gamma='auto'), {
    'C': [1, 10, 20],
    'kernel' : ['rbf', 'linear']
}, cv=5, return_train_score=False)
```

Now we can just do model training

In [24]:
```
clf.fit(iris.data, iris.target)
```

Out[24]:



And then we can print the cross validation result

In [25]:
```
clf.cv_results_
```

Out[25]:
```
{'mean_fit_time': array([0.00180287, 0.0023448 , 0.01830688, 0.00427442, 0.00082178,
       0.00105095]),
 'std_fit_time': array([0.00117161, 0.00162545, 0.02684566, 0.00101619, 0.00078414,
       0.00134619]),
 'mean_score_time': array([0.00060406, 0.00099773, 0.0042243 , 0.00054679, 0.00156155,
       0.00088172]),
 'std_score_time': array([0.00080749, 0.00089228, 0.00240197, 0.00045647, 0.00265104,
       0.00132158]),
 'param_C': masked_array(data=[1, 1, 10, 10, 20, 20],
              mask=[False, False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'param_kernel': masked_array(data=['rbf', 'linear', 'rbf', 'linear', 'rbf', 'linear'],
              mask=[False, False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'C': 1, 'kernel': 'rbf'},
  {'C': 1, 'kernel': 'linear'},
  {'C': 10, 'kernel': 'rbf'},
  {'C': 10, 'kernel': 'linear'},
  {'C': 20, 'kernel': 'rbf'},
  {'C': 20, 'kernel': 'linear'}],
 'split0_test_score': array([0.96666667, 0.96666667, 0.96666667, 1.        , 0.96666667,
       1.        ]),
 'split1_test_score': array([1., 1., 1., 1., 1., 1.]),
 'split2_test_score': array([0.96666667, 0.96666667, 0.96666667, 0.9       , 0.9       ,
       0.9       ]),
 'split3_test_score': array([0.96666667, 0.96666667, 0.96666667, 0.96666667, 0.96666667,
       0.93333333]),
 'split4_test_score': array([1., 1., 1., 1., 1., 1.]),
 'mean_test_score': array([0.98      , 0.98      , 0.98      , 0.97333333, 0.96666667,
       0.96666667]),
 'std_test_score': array([0.01632993, 0.01632993, 0.01632993, 0.03887301, 0.03651484,
       0.0421637 ]),
 'rank_test_score': array([1, 1, 1, 4, 5, 6])}
```

Now, when we run the above code we can see that we get the above result, cv result are not easy to view but luckily SKLearn provides a way to download cv result to Dataframe as below

In [26]:
```
newdf = pd.DataFrame(clf.cv_results_)
newdf
```

Out[26]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_kernel	params	split0_test_score	split1_test_sc
0	0.001803	0.001172	0.000604	0.000807	1	rbf	{'C': 1, 'kernel': 'rbf'}	0.966667	
1	0.002345	0.001625	0.000998	0.000892	1	linear	{'C': 1, 'kernel': 'linear'}	0.966667	
2	0.018307	0.026846	0.004224	0.002402	10	rbf	{'C': 10, 'kernel': 'rbf'}	0.966667	
3	0.004274	0.001016	0.000547	0.000456	10	linear	{'C': 10, 'kernel': 'linear'}	1.000000	
4	0.000822	0.000784	0.001562	0.002651	20	rbf	{'C': 20, 'kernel': 'rbf'}	0.966667	
5	0.001051	0.001346	0.000882	0.001322	20	linear	{'C': 20, 'kernel': 'linear'}	1.000000	

As u can see above its better now to undersand it, we can see that we have "C" parameter value then "kernel" values columns and also the scores of each individuals splits, we did 5 cross validation hence we got split0 to split4, and then u have mean test score column as well.

Some of these columns in the DF might not be useful so we can just trim it down and just look at C, kernel and mean test score columns

In [28]:
```
newdf[['param_C','param_kernel','mean_test_score']]
```

Out[28]:

	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

So u can see that these are the possible values of C then kernel and mean test score, based on this we can say that we can supply the first 3 values into our parameters to get the best performance so we have already did Hypertuning of these parameters.

And now u can have many many parameters, all u have to do is supply them in parameter grid inside GridSearchCV and it will take care of it and show u the scores in this nice dataframes

We can do dir method in our clf to see what other properties it has

In [30]:
```
dir(clf)
```

Out[30]:
```
['__abstractmethods__',
 '__annotations__',
 '__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__',
 '__sklearn_clone__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_abc_impl',
 '_build_request_for_signature',
 '_check_feature_names',
 '_check_n_features',
 '_check_refit_for_multimetric',
 '_estimator_type',
 '_format_results',
 '_get_default_requests',
 '_get_metadata_request',
 '_get_param_names',
 '_get_tags',
 '_more_tags',
 '_parameter_constraints',
 '_repr_html_',
 '_repr_html_inner',
 '_repr_mimebundle_',
 '_required_parameters',
 '_run_search',
 '_select_best_index',
 '_validate_data',
 '_validate_params',
 'best_estimator_',
 'best_index_',
 'best_params_',
 'best_score_',
 'classes_',
 'cv',
 'cv_results_',
```

```
 'decision_function',
 'error_score',
 'estimator',
 'fit',
 'get_metadata_routing',
 'get_params',
 'inverse_transform',
 'multimetric_',
 'n_features_in_',
 'n_jobs',
 'n_splits_',
 'param_grid',
 'pre_dispatch',
 'predict',
 'predict_log_proba',
 'predict_proba',
 'refit',
 'refit_time_',
 'return_train_score',
 'score',
 'score_samples',
 'scorer_',
 'scoring',
 'set_fit_request',
 'set_params',
 'transform',
 'verbose']
```

Above we can see some of the properties such as "best_estimators_", "best_params_" and "best_score_". So lets try "best_score_" first

In [31]:
```
clf.best_score_
```

Out[31]:
```
0.9800000000000001
```

As u can see from the above "newdf" DF the best score is also 0.98

We can also do "best_params_" and it will tell the best parameters

In [33]:
```
clf.best_params_
```

Out[33]:
```
{'C': 1, 'kernel': 'rbf'}
```

RandomizedSearchCV

One issue that can happen with **GridSearchCV** is the Computation cost, our dataset now is very limited but imagine if u have millions of datapoints and then for parameters u have so many values, right now "C" values are only 1,10,20 but what if i just want to try range lets say from 1 to 50 then our Computation cost will go very high because this will literelly try Permutation and Combinations for every value of each of these parameters.

To tackle this Computation problems, SKLearn libraries comes up with another class called **RandomizedSearchCV**.

RandomizedSearchCV will not try every single Permutatuon and combination of parameters but it will try random combinations of these parameters value and u can chose what those iterations could be so lets see how it works

In [34]:
```
from sklearn.model_selection import RandomizedSearchCV
```

The API kind of look the same as GridSearchCV, we supplid our Parameters grid, our cross validation value which is 5 and the most interesting parameter here is "n_iter=", which we passed in 2 as we want to try only 2 combinations so here, Above in GridSearchCV it did 6 as u can go back and see in the newdf that it has 0 to 5 rows, here we will only try 2 combinations and then we will call fit method and download the result into pandas DataFrame

In [36]:
```
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[36]:

	param_C	param_kernel	mean_test_score
0	1	rbf	0.98
1	1	linear	0.98

As we can see above, it randomly tried C value and the kernel as well, if we run it again it will randomly change.

This way it just randomly tries the value of C and kernel and it gives u the best score.

This works well in a practical life cuz if u dont have too much computation power then u just want to try random value or parameters and just go with whatever comes out the best

Choosing best Model

How about different models with different hyperparameters?

Alright so now we looked into parameter tuning, now lets see how do we chose the best model. For our iris dataset we are going to try SVM, Random Forest and Logistic Regression and we will figure out which one will give u the best performant, u have to define ur parameter grid and we are just defining them as a simple Python Dictionary where we are saying i want to try SVM model with these parameters, Random forest with this other parameters, i want the tree value in Random forest 1,5 and 10, Similarly the value "C" is a param in Logistic Regression

In [37]:
```
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
```

In [38]:
```
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]
        }
    }
}
```

Once we have initialized this Dictionoary, we can write a simple for loop and this for loop is doing nothing but its just going through this Dictionary values and for each of the values it will **GridSearchCV** and in **GridSearchCV** the first param is the model, just trying each of the model from the Dictionary one by one with the corresponding parameters grid that we have specified in the above Dictionary.

Then we run the Training and just append the scores into the scores list

In [39]:
```
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_
    })
```

Now we will just convert those scores as a dataframe

In [40]:
```
newdf2 = pd.DataFrame(scores,columns=['model','best_score','best_params'])
newdf2
```

Out[40]:

	model	best_score	best_params
0	svm	0.980000	{'C': 1, 'kernel': 'rbf'}
1	random_forest	0.946667	{'n_estimators': 10}
2	logistic_regression	0.966667	{'C': 5}

As we can see above a nice table view of the model, best score and best parameters of each model, so here we have a Conclusion that best model for our dataset is **SVM** as it will give us a **98% score** with **C : 1 and kernel : "rbf"** parameters

So not only we did Hyperparameters tuning but we also selected the best model, above we have used only 3 models for demonstration but u can use 100 models if u like so this is more like Trial and Error approach but in Practical life this works really well and this is what is used to figure out the best model and the best parameters