Chapter 11 : Part 1

**Model Selection**

K-fold Cross Validation, Grid Search

**11.1.1 Evaluating & Improving Model Performance**

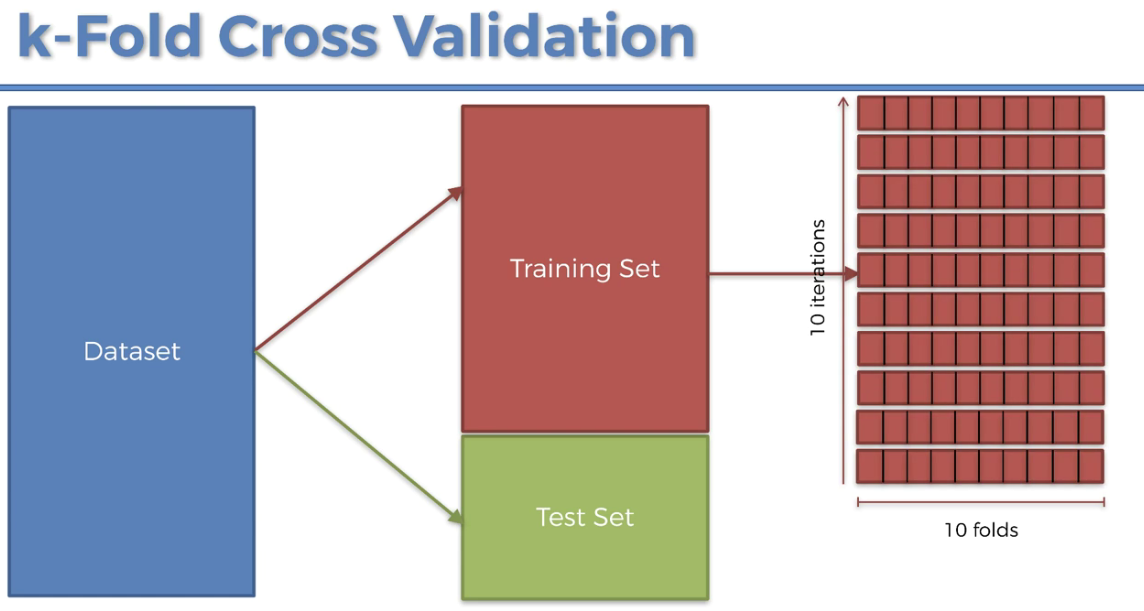
* Model parameters and Hyper parameters: In a ML model we have two types of parameters. Model parameters and Hyper parameters.
* Model parameters: These are the parameters that the *model* *learns* and found *optimal* *values* by running the *model*.
* Hyper parameters: This type of *parameters* are the *parameters* that we *choose* *ourselves*. For example the *kernel* *parameter* in the *kernel-SVM* or the ***penalty*** ***parameter*** or even some ***regularisation*** ***parameter***.

So there is still room to improve the model because we can still choose some optimal values for these parameters.

* In this chapter we will do two things:

1. Evaluating our model performance: To evaluate our models in a efficient way, we use K-fold-Cross-validation instead of just train-test-split. Then to tune- Hyper parameters we'll use Grid Search.
2. Improving our model performance: We'll use the most powerful algorithm in ML, XGBoost.

* Improving the model performance can be done with a technique called *model selection*. That consists of choosing the best parameters of your ML model.
* Grid Search: Since the model parameter is learned by the model then we need to figure out a way to choose the optimal values for these Hyper parameters. One of the powerful techniques to tune- Hyper parameters is Grid Search.
* K-fold-Cross-validation: We need to optimize a way to evaluate our models before we start grid search. Previously we just *split* our *dataset* between the *training* *set* and a *test set*.
* We trained our model on the *training* *set* and we *tested* its *performance* on *the test*. That's a correct way of evaluating the model performance.
* But that's not the *best* one because we actually have a *variance* *problem*.
* The variance problem can be explained by the fact that when we get the *accuracy* on the *test set*. And if we run the model again and test again it's performance on another test set, we can get a very different accuracy.
* Hence judging our *model* *performance* only on one *test* *set* is actually not a *good* *idea*. That's not the most relevant way to evaluate the model performance.
* K-fold-Cross-validation: K-fold-Cross-validation is a technique that resolve the *variance* *problem* by *splitting* the *training set* into K-fold.



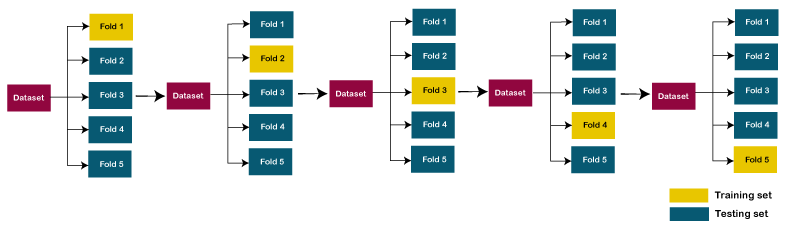
* No. of Folds: Most of the time ***K = 10***, and we train our model a nine fold and we test it on the last remaining fold.
* No. of Iterations: Since we have ***10 folds***, we can make ***10*** different ***combinations*** of ***9+1 fold*** to train the model. Thus the model can be train in 10 iterations for each combination of 10-fold.
* Once we ***train*** the model and ***test*** them all on ***ten*** ***combinations*** (10 iterations) of training and test sets. We'll get ***10*** ***different*** ***accuracies***.
* We take an ***average*** of ***10*** different accuracies to model-performance-evaluations and also compute the ***standard*** ***deviation*** to have a look at the variance. These will give us a better idea of our model's performance.

|  |  |
| --- | --- |
| * Eventually our analysis will be much more relevant and we'll know in which of these four categories will be:  1. Good accuracy (low bias) and a small variance - lower left 2. Large accuracy and a high variance - lower right 3. Small accuracy (high bias) & low variance - upper left 4. Low accuracy and a high variance – upper right. |  |

* *K-fold cross-validation* approach *divides* the *input* *dataset* into *K groups* of *samples* of equal sizes. These *samples* are called folds. For each learning set, the prediction function uses k-1 folds, and the rest of the folds are used for the test set. This approach is a very popular CV approach because it is easy to understand, and the output is less biased than other methods.
* The steps for k-fold cross-validation are:

1. Split the input dataset into K groups
2. For each group:

* Take one group as the reserve or test data set.
* Use remaining groups as the training dataset
* *Fit* the model on the *training set* and *evaluate* the *performance* of the model using the *test set*.
* Let's take an example of 5-fold cross-validation. So, the dataset is grouped into 5 folds.
* On 1st iteration, the *first* *fold* is reserved for *test* the model, and rest are used to *train* the model.
* On 2nd iteration, the *second* *fold* is used to *test* the model, and rest are used to *train* the model. This process will continue until each fold is not used for the test fold. Consider the below diagram:



**11.1.2 K-fold cross-validation :** Evaluate model performance

Let's start with this K-fold cross-validation, our first technique of model selection. We are going to use one of the model that we've built previously and apply K-folds cross-validation on it.

* We're gonna use the kernel-SVM model from Chapter -3 : Classification. Where we used the Social Network Ads data and kernel-SVM used to predict if the customers are going to click on the ads on the social network to buy the SUV (yes or no).
* We'll use the dataset ***Social\_Network\_Ads.csv*** contains the information of various users obtained from the *social* *networking* *sites*. There is a car making company that has recently launched a new SUV car. So the company ***wanted to check how many users from the dataset, wants to purchase the car***.
* Since the model is *already* *built* and we already have *everything*. We just *copy* the whole *code* and implement K-fold cross-validation in right place (on a new section of code).
* Where to apply K-folds cross-validation code section: Since that consists of evaluating the model performance, the most relevant location to put it is right after we *build* our kernel-SVM model that is right we built the model.
* After y\_pred and confusion-matrix?: Since getting the predictions of the test results and evaluating the confusion-matrix is actually a good way of evaluating the model. But not the best way, we apply K-fold cross-validation after ***y\_pred*** and ***confusion-matrix***.
* We import the ***cross\_val\_score*** class from ***sklearn.model\_selection*** (the same module already used for ***train\_test\_split***).

**from** sklearn.model\_selection **import** cross\_val\_score

* Now we apply *k-fold-cross-validation* on our *training* *set*: we create an object of ***cross\_val\_score*** called ***accuRacies*** (this is actually a vector).
* ***accuRacies*** will return 10 accuracy's (a *vector of accuracies* of the model) for each one of the 10 combinations that will be created through ***10****-fold-cross-validation*, (K =10 here).

accuRacies = **cross\_val\_score**(estimator=clsFier, X = X\_train, y = y\_train, cv= 10)

* Parameters:

1. estimator : estimator is the object that implementing 'fit' on the data. This is our SVM-classifier object, so we set **estimator = clsFier**,
2. X : is the data to fit. It is actually our feature-matrix of the training sets ***X\_train***. So we set **X = X\_train**,
3. y : is the target variable, (i.e the ***dependent variable vector***) to try to predict in the case of supervised learning. So we set **y = y\_train**,
4. cv : Determines the cross-validation splitting strategy. It is the number of folds of *k-fold-cross-validation.* Here we wanna apply***10 folds*** so we set**cv = 10***.*

* The most common choice for this CV number is actually 10. Most of the time you'll use ***10****-fold-cross-validation* where you'll get ***10*** accuracy's and ***10*** accuracy's is actually enough to get a *relevant* *idea* of the *model* *performance*.

1. **n\_jobs** (optional): Number of jobs to run in parallel. Training the estimator (classifier/regressor) and computing the score are parallelized over the cross-validation (CV) splits.

* **None** means **1** unless in a joblib.parallel\_backend context.
* **-1** means using all processors. It means that you will use all the CPU on your machine and therefore your can run *k-fold-cross-validation* even faster in case you are working on a very large dataset.
* Calculating Mean of accuracies and slandered deviation: We use **mean**() to return mean, and **std**() to return standard deviation.

mean\_accu = **accuRacies.mean**()

std\_accu = **accuRacies.std**()

# *Applying K-folds cross-validation*

**from** sklearn.model\_selection **import** cross\_val\_score

accuRacies = **cross\_val\_score**(estimator=clsFier, X = X\_train, y = y\_train, cv= 10)

mean\_accu = **accuRacies.mean**()

std\_accu = **accuRacies.std**()

* After executing following code:

# *--------------- Model-Selection and Boosting ------------------*

# *Library*

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

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet.iloc[:, [2,3]].values

y = dataSet.iloc[:, 4].values

# *Data Split*

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

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Fit train set to Kernel-SVM classifier*

**from** sklearn.svm **import** SVC

# *since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.*

# *kernel="rbf" instead of kernel="linear"*

clsFier = **SVC**(kernel="rbf", random\_state=0)

**clsFier.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **clsFier.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *--------  Applying K-folds cross-validation  -------------*

**from** sklearn.model\_selection **import** cross\_val\_score

accuRacies = **cross\_val\_score**(estimator=clsFier, X = X\_train, y = y\_train, cv= 10)

mean\_accu = **accuRacies.mean**()

std\_accu = **accuRacies.std**()

|  |  |
| --- | --- |
| * Accuracy vector: Here is our accuracy's vector. The first accuracy is 80% but then the second accuracy is 96% and then 80% followed by other accuracies. * It clearly shows that, the performance of your model changes when test-set varies. So evaluating the medel performance on one test set is not very relevant. * With 10-fold-cross-validation we are testing it on 10 test sets. * Mean: Now we're going to take the mean of all these ***10 accuracies*** and that better idea of the average model-performance of our model.   mean\_accu = **accuRacies.mean**()  The mean of these 10 accuracy's here is actually 90.33%. So in conclusion this 90% accuracy is the *relevant* *evaluation* of our *model* *performance*.   * Standard Deviation: If we want to put the analysis further, we can compute also the standard deviation of this accuracy's vector that will tell us if there is a high variance or low variance.   std\_accu = **accuRacies.std**() |  |
| * We get a 6.5% standard deviation. That means the average of the differences between the different accuracies and 90.3% is 6.5%. * That's actually not too high variance. It means that when we evaluate our model performance, most of the time will be around 84% and 96% so eventually that means that we are in this low bias and low variance category. |  |

***All code for k-fold-cross-validation at once (practiced version)***

# *--------------- Model-Selection and Boosting ------------------*

# *Library*

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

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet.iloc[:, [2,3]].values

y = dataSet.iloc[:, 4].values

# *Data Split*

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

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Fit train set to Kernel-SVM classifier*

**from** sklearn.svm **import** SVC

# *since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.*

# *kernel="rbf" instead of kernel="linear"*

clsFier = **SVC**(kernel="rbf", random\_state=0)

**clsFier.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **clsFier.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *Applying K-folds cross-validation*

**from** sklearn.model\_selection **import** cross\_val\_score

accuRacies = **cross\_val\_score**(estimator=clsFier, X = X\_train, y = y\_train, cv= 10)

mean\_accu = **accuRacies.mean**()

std\_accu = **accuRacies.std**()

# *Visualising the Training set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **clsFier.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.3, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Kernel-SVM (Training set)')

**pLt.xlabel**('Age')

**pLt.ylabel**('Estimated Salary')

**pLt.legend**()

**pLt.show**()

# *Visualising the Test set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **clsFier.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.3, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Kernel-SVM (Test set)')

**pLt.xlabel**('Age')

**pLt.ylabel**('Estimated Salary')

**pLt.legend**()

**pLt.show**()

**11.1.3 Grid-Search :** Improve models performance

In the previous section we used K-fold-cross-validation to *evaluate* our model *performance*. In this section we're going to learn Grid-Search which is used to *improve* models *performance*.

* We'll *improve* our model by finding the *optimal* values of the *hyper parameters*, the parameters that we choose.
* This Grid Search technique will help us to choose appropriate hyper parameters for our model and ***tune*** them to the ***optimal value***.
* *How do I know which model to choose for my machine learning problem? I have a machine learning problem that comes with a specific dataset, how do I know which model to choose to solve my business problem? Which model would be the best one?*
* Step 1: Figure out, if your problem is a ***regression*** problem or a ***classification*** problem or a ***clustering*** problem.
* You just need to look at your dependent variable.
* If you don't have a dependent variable then it's a **clustering** *problem*.
* And if you have a *dependent* *variable* you see if it's a *continuous outcome* or a *categorical outcome*.
* If it's a *continuous* *outcome* then your problem is a **Regression** problem.
* If it's a *categorical* *outcome* then your problem is a **Classification** problem.
* Step 2: find out your problem is linear problem or nonlinear problem.
* When you have a ***large data set*** you ***cannot figure out*** the ***linearity*** of your dataset ***easily***. For large data-set you have to try both situation, to choose a *linear* model like *SVM* or a *nonlinear* model like *kernel-SVM* (if you're doing classification).
* In this kind of situation where we have large-dataset we can get help from Grid search. *Grid search* will *tell* us if we should rather choose a *linear model* *like* SVM or a *non-linear model* *like* kernel-SVM.
* Problem description: We're going to work on the same problem as in the previous section (k-fold-cross-validation). *Classify* the *users* in the *social network* and predict if they're going to click *yes* or *no* on the ad to *buy* the *SUV*.
* So we use the same data-set ***Social\_Network\_Ads.csv***.
* ***Grid search*** will tell us if we should rather choose an ***SVM*** model or a ***kernel-SVM*** model.
* Since the *kernel SVM* model has *many* *parameters*, like ***penalty***, ***gamma*** parameter and ***grid search*** will tell us exactly which ***values*** we should choose for these ***hyper-parameters***.
* Fitting grid search: Basically *grid search* can be *applied* after *fitting* your *model* to the *training set* because one of the *paramter* of *grid search* will be the ***classifier***.
* Since we first used *K-fold-cross-validation* to evaluate the model performance and now we are working on improving the model performance, we put the **grid search** section right **after** *K-fold-cross-validation* section.
* We import ***GridSearchCV*** package from ***sklearn.model\_selection*** because grid search is a *model selection technique*.

sklearn.model\_selection **import** GridSearchCV

* Specifying the different parameters: To use Grid-search for Hyper-parameter tuning, we need to ***input*** those ***Hyper-parameters*** as a ***list*** of ***dictionary***. So first we create this dictionary of parameters. We name this dictionary of parameters as ***params\_dic***.
* When we built our model we only used 2 parameters: clsFier = **SVC**(kernel="rbf", random\_state=0)
* But there are other parameters for ***SVC()*** that we didn't specify, and we used their default values. For example: penalty parameter ***C***, ***degree*** (in case of polynomial), ***gamma*** etc.
* Say we want to tune C, kernel & gamma, 3 hyper-parameters. Then we create dictionaries "key-value" pair, parameter-name as key and *list of values* that we want to set for tuning as value of the dictionary. Those values will be tested by the grid search model. And among these values the grid search model will find the best one.

# *--------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------*

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

params\_dic = [

            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},

            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [0.7, 0.5, 0.1, 0.01 ,0.001, 0.0001]},

            ]

* This ***penalty*** parameter is for *regularization* to prevent *overfitting*, the default value is 1. We tune this for our model to prevent overfitting.
* The more you increase this *penalty* parameter *C* the *more* it will prevent *overfitting*. But be careful you should *not increase* it *too* *much* because otherwise you will get a new problem which will be Underfitting. For example 10000 would be too much for penalization.
* ***kernel*** parameter also included it in the dictionary as well, to find out the best model, a *linear* model or a *non-linear* (Eg: **rbf**)model. If you are dealing with large data-set then it will be very useful for you to decide if you should take a linear or non-linear model.
* We're not going to include ***degree*** in the dictionary because we won't test if we should take polynomial Kernel. We're just examine between *linear* model or a *non-linear* model.
* ***gamma*** is a parameter for the nonlinear kernels like **'rbf'**, **'poly'** and **'sigmoid'**. Since we are using **'rbf'**, We'll try to optimize this *gamma* parameter and find the *best* *value*.
* The default value is **auto** and if gamma is **auto** than **1/n** features will be used instead. So we set the values in the *range* ***[0, 1]***.
* Here in this problem, we have ***2 features*** so we set **(1/2)=0.5**, also **0.7** and other values if **0.5** is not the optimal.
* If you have a *lot more features* for example **100** *features*, or **1000** *features* you can even try a *smaller* *number* for the **gamma** parameter like **0.001**.
* The first option is: Grid Search will investigate is a linear model that is a classic SVM with a linear kernel. And it will try to find the optimal value for the penalty parameter C.
* The second option is: Grid Search will investigate the nonlinear model 'rbf' kernel-SVM, it will tune gamma, and also the penalty parameter C.

Eventually what we'll get is a *single combination* of all this *different* *parameters* and options. And *grid-search* will find that *combination* for us.

* Applying Grid-search: In this step we are going to implement ***grid search***. We do that right after the *list* of *parameters-dictionary* that we just created.
* We'll create an *object* of ***GridSearchCV*** class and we are going to fit this *object* on our *training* set ***X\_train***.

griD\_Srch = **GridSearchCV**(estimator = clsFier,

                        param\_grid= params\_dic,

                        scoring= 'accuracy',

                        cv = 10,

                        n\_jobs= -1     )

griD\_Srch = **griD\_Srch.fit**(X\_train, y\_train)

best\_accu = griD\_Srch.best\_score\_

best\_parameters = griD\_Srch.best\_params\_

* Parameters of **GridSearchCV**:

1. estimator : This is our SVM-classifier object, so we set **estimator = clsFier**,
2. **param\_grid:** is the *list* of *parameters-dictionary* that we want to tune using Grid-Search.

***param\_grid= params\_dic***,

1. **scoring:** Is the scoring matrix that is used to decide the best parameters. We set **scoring= 'accuracy'**,

* *Grid search* is going to *select* the best parameters based on one performance matrix.
* It can be the ***accuracy*** matrix or it can be the ***precision*** matrix, it can be the ***recall***. So, it can be different performance metrics.
* We are going to take the most common one like we did for deep learning the accuracy matrix. So , we set **scoring= 'accuracy'**.

1. cv : *grid search* is going to evaluate the *performance* of each of the model with their own set of parameters, using *K-fold-cross-validation*. So we set ***10*** *folds*, ***cv = 10***.

* The most common choice for this CV number is actually 10. Most of the time you'll use ***10****-fold-cross-validation* where you'll get ***10*** accuracy's and ***10*** accuracy's is actually enough to get a *relevant* *idea* of the *model* *performance*.

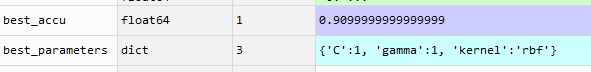
*These above 4-parameters will do the job for grid-search.* We can also use following optional parameter.

1. **n\_jobs** (optional): If you are working on a very large dataset you can set, ***n\_jobs= -1.*** It should get all the power available from your machine.

* **-1** means using all processors. It means that you will use all the CPU on your machine and therefore your can run *k-fold-cross-validation* even faster in case you are working on a very large dataset. (otherwise use **None**)
* We then use ***fit()*** to apply this Grid-search to our train-set ***X\_train*** and ***y\_train*** (in case of unsupervised model we don’t need ***y\_train***).

griD\_Srch = **griD\_Srch.fit**(X\_train, y\_train)

* To look at the results, we interested in best accuracy and best parameters. We'll ***usebest\_score\_*** and ***best\_params\_*** attributes
* Best Accuracy: ***best\_accu = griD\_Srch.best\_score\_*** returns the best accuracy that Grid-search found for the given hyper-parameters.
* Best Parameters: ***best\_parameters = griD\_Srch.best\_params\_*** returns the best dictionary of parameters that Grid-search found for the given hyper-parameters.



Above result shows us, we should use a non-linear model ***kernel SVM*** using "**rbf**", with C=1 and gamma =1.

# *--------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------*

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

params\_dic = [

            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},

            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [1, 0.9, 0.8, 0.7, 0.5, 0.1, 0.01]},

            ]

griD\_Srch = **GridSearchCV**(estimator = clsFier,

                        param\_grid= params\_dic,

                        scoring= 'accuracy',

                        cv = 10,

                        n\_jobs= -1     )

griD\_Srch = **griD\_Srch.fit**(X\_train, y\_train)

best\_accu = griD\_Srch.best\_score\_

best\_parameters = griD\_Srch.best\_params\_

**All code at once (with k-fold-cross-validation and Grid-search)**

# *--------------- Model-Selection and Boosting ------------------*

# *Library*

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

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet.iloc[:, [2,3]].values

y = dataSet.iloc[:, 4].values

        # *Feature-Scaling after Data Split*

# *Data Split*

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

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Fit train set to Kernel-SVM classifier*

**from** sklearn.svm **import** SVC

# *since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.*

# *kernel="rbf" instead of kernel="linear"*

clsFier = **SVC**(kernel="rbf", random\_state=0)

**clsFier.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **clsFier.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *--------   K-folds cross-validation  -------------*

**from** sklearn.model\_selection **import** cross\_val\_score

accuRacies = **cross\_val\_score**(estimator=clsFier, X = X\_train, y = y\_train, cv= 10)

mean\_accu = **accuRacies.mean**()

std\_accu = **accuRacies.std**()

# *--------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------*

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

params\_dic = [

            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},

            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [1, 0.9, 0.8, 0.7, 0.5, 0.1, 0.01]},

            ]

griD\_Srch = **GridSearchCV**(estimator = clsFier,

                        param\_grid= params\_dic,

                        scoring= 'accuracy',

                        cv = 10,

                        n\_jobs= -1     )

griD\_Srch = **griD\_Srch.fit**(X\_train, y\_train)

best\_accu = griD\_Srch.best\_score\_

best\_parameters = griD\_Srch.best\_params\_