DATA FOLKZ

CATAPULT DATA LEADERS

CROSS VALIDATION

Cross Validation

Validating the stability of the model is really important . We want our model to capture most of the patterns of the data correctly and not pick up the noise of the data . In other words we need to avoid the problems of underfitting and overfitting . For this purpose, we use the cross-validation technique.

Underfitting is the case when our model fails to capture the important patterns or trends of the data .
And, overfitting is the case in which our model captures the each deviation of the data point including noise .

What is Cross Validation ?

Cross-validation is a technique in which we train our model using the subsets of the dataset and then evaluate using the complementary subset of the dataset . Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data.
Cross validation has a parameter k which refers to the number of groups the data will be divided into .

How is Cross validation performed :
We reserve a group from our dataset and train the model on the rest of the dataset . Once the model is trained we test the model on the reserved group . This will help us in checking the performance of the model. If the model performance is positive, then we can go ahead with the model.

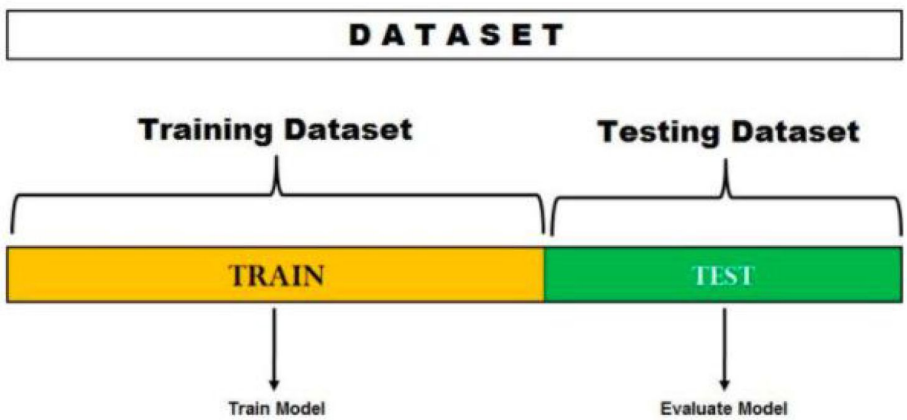
Types of Cross Validation

1. Hold out Validation
2. Leave One Out Cross Validation (LOOCV)
3. K- Fold Cross Validation
4. Stratified K- fold Cross Validation

Let's discuss each of them in detail .

Hold out Validation

This is the simplest kind of validation . In this technique , the dataset is divided into two parts i.e. training and testing . The model is trained on **Training data** and then tested on **Testing data** and error rate is estimated to check the model performance .



Practical implementation in Python :

We are using house pricing dataset for this study . All the preprocessing such as imputing missing values , encoding categorical variables and scaling is done .
We will see the how the train and test split is done .

Our dataset :

```
house=pd.read_excel('housing_dataset.xlsx')
house.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41	880	129.0	322	126	8.3252	NEAR BAY	452600
1	-122.22	37.86	21	7099	1106.0	2401	1138	8.3014	NEAR BAY	358500
2	-122.24	37.85	52	1467	190.0	496	177	7.2574	NEAR BAY	352100
3	-122.25	37.85	52	1274	235.0	558	219	5.6431	NEAR BAY	341300
4	-122.25	37.85	52	1627	280.0	565	259	3.8462	NEAR BAY	342200

Applying Hold out validation

```
#splitting the data into train and test
from sklearn.model_selection import train_test_split
ind_train,ind_test,dep_train,dep_test=train_test_split(house_ind,house_dep,test_size=0.2,random_state=0)
```

X x y y

Testing data can be of any size, but in this example, I took test_size = 0.20 i.e. 20% of the entire dataset.

Cons of this method :

- If the train or test dataset are not able to represent the actual patterns of complete data as it is not certain that which data points will be in which set then the results from the test sets can be skewed and the model will not perform well on the new data points given .
- Errors found in the test dataset can highly depend on the observations included in the train and test dataset. As we change the *random_state* parameter value in *train_test_split* function our data points changes in both the training and test dataset and results are different scores which makes the model highly different for different data points .

Example of the above limitation in the codes below :

In the first case we are using random state as 0

```
#splitting the data into train and test
from sklearn.model_selection import train_test_split
ind_train,ind_test,dep_train,dep_test=train_test_split(house_ind,house_dep,test_size=0.2,random_state=0)
```

Applying Linear regression model (this is not the best model of the dataset , using it for understanding purpose)

```
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(ind_train,dep_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
#prediction
pred=regressor.predict(ind_test)
```

Evaluating the model using R2 score and RMSE score

```
from sklearn import metrics
print("R square :",metrics.r2_score(dep_test,pred))
print("RMSE :",np.sqrt(metrics.mean_squared_error(dep_test,pred)))
```

```
R square : 0.6380862497737064
RMSE : 68696.37609238942
```

Now, in second case we are going the change our *random_state* to 45

```
#splitting the data into train and test
from sklearn.model_selection import train_test_split
ind_train,ind_test,dep_train,dep_test=train_test_split(house_ind,house_dep,test_size=0.2,random_state=45)
```

Applying Linear regression and evaluating the model

```
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(ind_train,dep_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
#prediction
pred=regressor.predict(ind_test)
```

```
from sklearn import metrics
print("R square :",metrics.r2_score(dep_test,pred))
print("RMSE :",np.sqrt(metrics.mean_squared_error(dep_test,pred)))
```

```
R square : 0.6270205055928706
RMSE : 70350.36939111221
```

We can observe from the R2 and RMSE scores that as we changed the value of random state , the scores of the model changed .
So to overcome this limitation of train test split method , we can use other Cross Validation methods . Most commonly used methods are K-Fold and Stratified K-Fold methods .

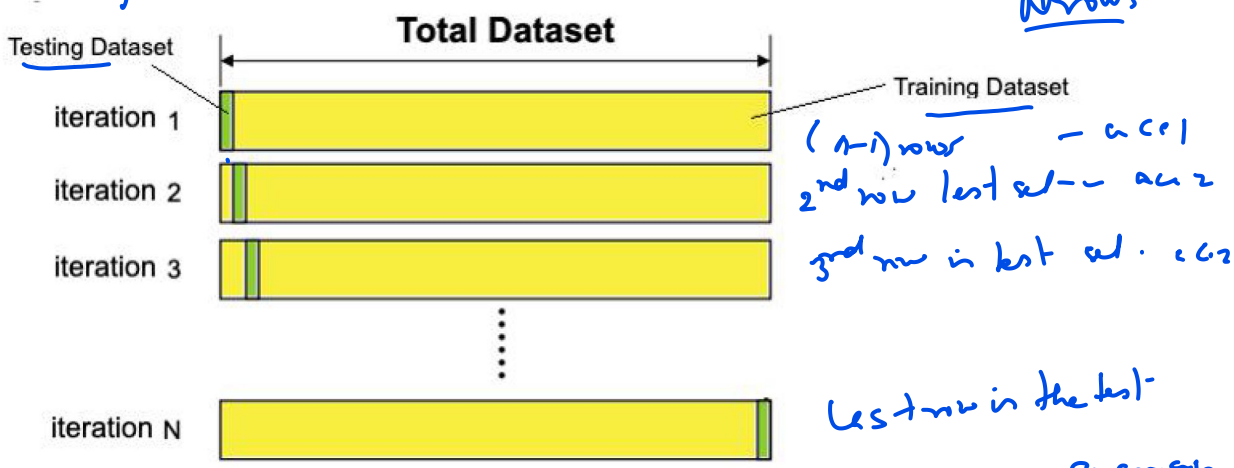
We will discuss them in detail .

Leave One Out Cross Validation (LOOCV)

This approach leaves 1 data point as testing data and trains the model on the rest of the data. This process iterates for each data point i.e if there are n data points in the original sample

then, n-1 samples are used to train the model and 1 datapoint will be used for testing .
For example : If you have 1000 data points the model will take 1 data point at time for testing and 999 points will be used for training purposes . This process will repeat for 1000 times .

This method will cover all the data points and learns everything, hence the bias will be very low but as we repeat the process *n* time where *n* is the total number of data points, which results in high execution time. Also , this is the old method of CV and is not used now .



Python Implementation of LOOCV :

```
from sklearn.model_selection import LeaveOneOut

loo=LeaveOneOut()
loo.get_n_splits(house_ind) #house_ind contains independent variables
```

```
20640
```

```
for train_index,test_index in loo.split(house_ind):
    X_train,X_test=house_ind.iloc[train_index],house_ind.iloc[test_index]
    y_train,y_test=house_dep[train_index],house_dep[test_index]
```

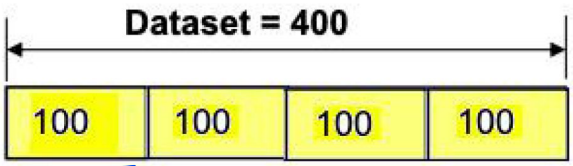
K- Fold Cross Validation

When we split our data into training and testing sets, there is always a chance of losing the crucial information or losing patterns that might go unnoticed by the model.
This will lead to underfitting or overfitting.

To overcome this problem, we need enough data in both the training and testing sets. For this we have **K-fold cross validation**

This how K fold CV works :

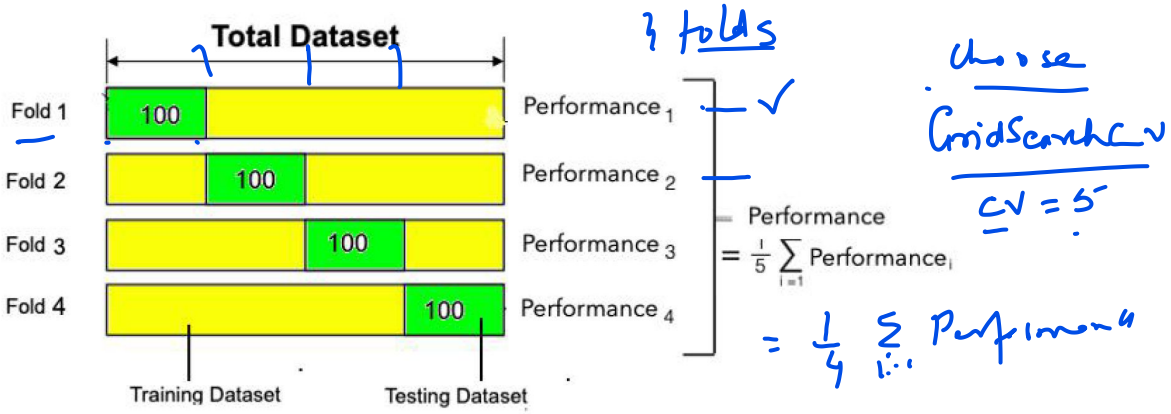
1. Firstly , Dataset is shuffled randomly .
2. Then it is splitted into **k** equal size subsets called **fold**, where k will be any integer value. Suppose we have a dataset of size 400 and we **choose k = 4**, then we will divide the dataset into 4 groups with (400/4) = 100 data points each .



3. For each **fold** (i.e. 4 groups in our case) these steps are repeated :

   First 100 data points i.e one group is taken as test data and rest 300 data points i.e 3 groups will be used for training . Model is fitted on the training dataset and evaluated on the test dataset and the **performance score** is recorded .

4. The **average** of k recorded (in our case 4) **performance score** is called the cross-validation error and will serve as the performance metric for the model .



Python Implementation of K-fold Cross Validation :

We will use *cross_val_score* function from the sklearn library .

```
from sklearn.model_selection import cross_val_score

scores=cross_val_score(regressor,X,y,cv=10)
scores
```

```
array([0.22718749, 0.68255467, 0.49496942, 0.4776349 , 0.37260517,
       0.5798987 , 0.49799184, 0.41303756, 0.50747594, 0.6457295 ])
```

Parameters used in *cross_val_score* :

- Model object , here we are using regressor which is a Linear model object .
- Independent variables , here X
- Dependent variable , here y
- Value of k i.e number of groups we want to divide our data into , here 10

The output which we got are the scores of each iteration , so we got 10 scores .

Now, we will average all the scores to get the final score of our model . Our final score is 0.4899

```
scores.mean()
```

```
0.4899085185712197
```

This technique significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in the test set.

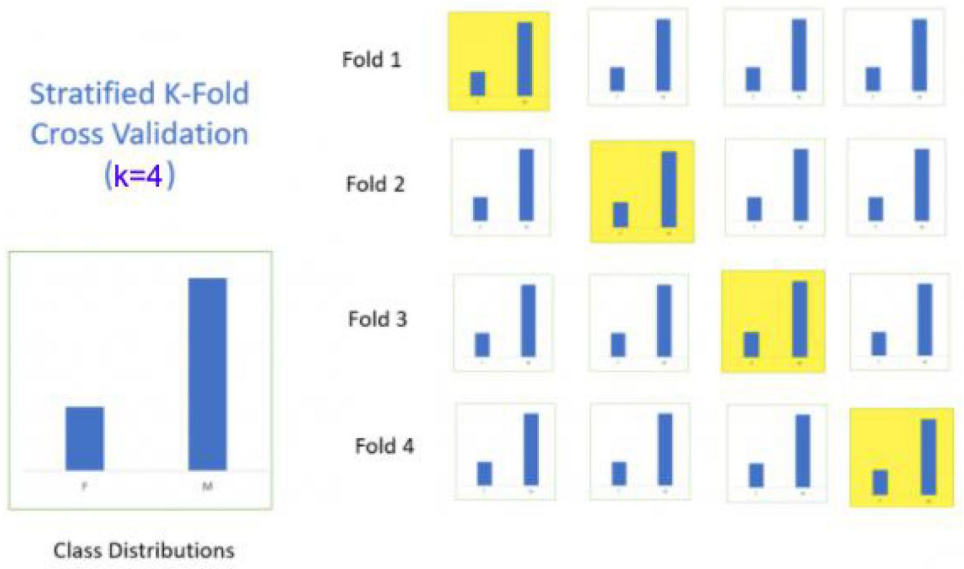
But , one limitation with K fold Cross validation is that for classification problems or when our dataset is imbalanced , it can take the same type of data points in the majority of the folds. For example, in a classification problem, there may be a large number of negative outcomes in a testing group, and in another testing group it could be very less. This will again lead to bias and high variance in the outcome.
To solve this problem we have another technique called **Stratified K fold Cross Validation .** Let's discuss more about it .

Stratified K fold Cross Validation

This works almost the same as K fold Cross validation , just a slight variation is that it makes sure to divide the dataset into groups such that each fold contains approximately the same percentage of samples of each target class as the complete set, or in case of prediction

problems, the mean response value is approximately equal in all the folds.

In below image, the stratified k-fold validation is set on basis of Gender whether M or F



Practical Implementation in Python :

Stratified K Fold is available in the sklearn library in model selection package. We are dividing our dataset into 10 folds and then storing the scores of every iteration in the list r2_score .

```
from sklearn.model_selection import StratifiedKFold
```

```
skf=StratifiedKFold(n_splits=10,random_state=None)

#X is the feature set and y is target variable
r2_scores=[]
for train_index,test_index in skf.split(X,y):
    X_train,X_test=X.iloc[train_index],X.iloc[test_index]
    y_train,y_test=y.iloc[train_index],y.iloc[test_index]

    #fitting and evaluating the model
    regressor.fit(X_train,y_train)
    prediction=regressor.predict(X_test)
    score=metrics.r2_score(y_test,prediction)
    r2_scores.append(score)
print(r2_scores)
```

```
[0.5719868242300299, 0.6406983279348772, 0.6506918551474856, 0.6800737445782251, 0.6362814394493529, 0.6353205730729896, 0.5991
96089274085, 0.600244595768096, 0.6773443407472113, 0.6773058606618332]
```

Now, taking the mean of the scores to get the final score .

```
np.array(r2_scores).mean()
```

```
0.6369143650864186
```