

Scikit-learn is the most useful library for machine learning in Python. It provides a selection of efficient tools for *machine learning* and *statistical modeling* including classification, regression, clustering and dimensionality reduction. Also scikit-learn focused on modeling the data. This library is built upon NumPy, Scipy and Matplotlib. This tutorial will explore statistical learning with scikit-learn.

1- Installation

```
pip install scikit-learn
```

or in Anaconda:

```
conda install scikit-learn
```

2- Features:

Some of models provided by sklearn are as follows:

- Supervised Learning algorithm
- Unsupervised Learning algorithm
- Clustering
- Cross Validation
- Dimensionality Reduction
- Ensemble methods
- Feature Extraction
- Feature Selection
- Open Source

3- Modelling Process

3-1- Dataset Loading

Dataset have two components:

- **Features:** variable of data are called its features. They are also known as predictors, inputs or attributes:

- Feature matrix: collection of features
- Feature names: list of all names of the features

- **Response:** output variable that basically depends upon the feature variables. They are also known as target, label or output:

- Response Vector: it is used to represent response column
- Target Names: it represent the possible values taken by a response vector

Scikit-learn have few example datasets like *iris* and *digits* for classification and the *Boston house price* for regression.

```
from sklearn.datasets import load_iris
iris=load_iris()
X=iris.data
y=iris.target
feature_names=iris.feature_names
target_names=iris.target_names
print(f'Feature_names are : {feature_names}')
print(f'Target_names are: {target_names}')
print(f'First 10 rows of X is:\nX[:10]')
```

```
Feature_names are : ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target_names are: ['setosa' 'versicolor' 'virginica']
First 10 rows of X is:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]]
```

output:

3-2- Splitting the dataset

To check the accuracy of model, we can split the dataset into two pieces- *a training set* and *a test set*. then use the training set to train the model and testing set to test the model.

`train_test_split(X,y,test_size,random_size):`

-X: is feature matrix

-y: is response vector

-test_size: this represent the ratio of test data to the total given data.

-random_size: it is used to guarantee that the split will always be the same.

```
from sklearn.datasets import load_iris
iris=load_iris()

X=iris.data
y=iris.target

from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,random_state=1)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

output:

```
(105, 4)
(45, 4)
(105,)
(45,)
```

3-3- Train the model

Now, we can use our dataset to train some prediction-model. here for example we use KNN(have seprate chapter for that)

```
from sklearn.datasets import load_iris
iris=load_iris()
X=iris.data
y=iris.target
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,random_state=1)
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
knn=KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train,y_train)
y_pred=knn.predict(X_test)
print(f'Accuracy is: {metrics.accuracy_score(y_test,y_pred)}')
```

output:

Accuracy is: 0.9777777777777777

3-4- Preprocessing the Data

Before inputting data to machine learning algorithms, we need to convert it into meaningful data. This process is called **Preprocessing the Data**. Scikit-learn has package named *preprocessing*.

3-4-1- Binarisation

for example we need to convert numerical values into Boolean values. here , use threshold value=0.5 , then all values above 0.5 would be converted to 1, and other value converted to 0.

```
import numpy as np
from sklearn import preprocessing
data=np.array([[2.1, -5, 6.3, -0.1],
               [3.9, 9.8, -6.3, -7],
               [2, 3.8, 6, 5],
               [-8, 0, 2.1, 6]])

print(f'Binarized data is :\n {preprocessing.Binarizer(threshold=0.5).transform(data)}')
```

output:

```

Binarized data is :
[[1. 0. 1. 0.]
 [1. 1. 0. 0.]
 [1. 1. 1. 1.]
 [0. 0. 1. 1.]]

```

3-4-2- Mean and Standard division

```

import numpy as np
from sklearn import preprocessing
data=np.array([[2.1,-5,6.3,-0.1],
               [3.9,9.8,-6.3,-7],
               [2,3.8,6,5],
               [-8,0,2.1,6]])
print(f'Mean of data in axis=0 is : {data.mean(axis=0)}')
print(f'Standard division of data in axis=0 is : {data.std(axis=0)}')

```

output:

```

Mean of data in axis=0 is : [0.    2.15  2.025 0.975]
Standard division of data in axis=0 is : [4.68027777 5.40809578 5.0839822  5.15285115]

```

3-4-3- Scaling

We use scaling technique, because the features should not be synthetically large or small.

```

import numpy as np
from sklearn import preprocessing
data=np.array([[2.1,-5,6.3,-0.1],
               [3.9,9.8,-6.3,-7],
               [2,3.8,6,5],
               [-8,0,2.1,6]])
scaler=preprocessing.MinMaxScaler(feature_range=(0,1))
data_scaled=scaler.fit_transform(data)
print(f'minmax scaled data in is \n : {data_scaled}')

```

output:

```

minmax scaled data is:
[[0.8487395  0.    1.    0.53076923]
 [1.    1.    0.    0.    ]
 [0.84033613 0.59459459 0.97619048 0.92307692]
 [0.    0.33783784 0.66666667 1.    ]]

```

3-4-4- Normalization

We use Normalization technique for modify the feature vectors. Normalization is necessary so that the feature vectors can be measured t common scale. there are two type of normalization: L1 Normalization and L2 Normalization

L1 Normalization also called Least Absolute Deviations. It modifies the value in such a manner that the sum of the absolute values remains always up to 1 in each row.

```

import numpy as np
from sklearn import preprocessing
data=np.array([[1,6.2,-5,3],
               [-3.6,2,8,7],
               [-3.6,5.2,0,1],
               [-9,5.2,14,4]])
normalize_data=preprocessing.normalize(data,norm='l1')
print(f'L1 Norm of data is:\n{normalize_data}')

```

output:

```

L1 Norm of data is:
[[ 0.06578947  0.40789474 -0.32894737  0.19736842]
 [-0.17475728  0.09708738  0.38834951  0.33980583]
 [-0.36734694  0.53061224  0.    0.10204082]
 [-0.27950311  0.16149068  0.43478261  0.1242236 ]]

```

L2 Normalization also called Least Squares. It modifies the value in such a manner that the sum of the squares remains always up to 1 in each row.

```
import numpy as np
from sklearn import preprocessing
data=np.array([[1,6.2,-5,3],
               [-3.6,2,8,7],
               [-3.6,5.2,0,1],
               [-9,5.2,14,4]])
normalize_data=preprocessing.normalize(data,norm='l2')
print(f'L2 Norm of data is:\n{normalize_data}')
```

output:

```
L2 Norm of data is:
[[ 0.11669001  0.72347804 -0.58345004  0.35007002]
 [-0.31578947  0.1754386   0.70175439  0.61403509]
 [-0.56222554  0.81210356  0.         0.15617376]
 [-0.50308385  0.29067067  0.78257488  0.22359282]]
```

4- Data Representation

As we know machine learning is about to create model from data. for this purpose, computer must understand the data first. here, we are going to discuss various ways to represent the data:

4-1- Data as Table

the best way to represent data is the form of tables. A table represents a 2-D grid of data where rows represent the individual elements of the database and columns represents the quantities related to those individual elements. as you see, each column of the data represents a quantitative information describing each sample.

```
import seaborn as sns
dat=sns.load_dataset('iris')
dat.head()
```

output:

	sepal_length	sepal_width	petal_length	petal_width	species
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

4-2- Data as Feature Matrix

Feature matrix may be defined as the table layout where information can be thought of as a 2-D matrix. it is stored in a variable named **X** and assumed to be two dimensional with shape[n_samples,n_features]. Mostly, it is contained in a NumPy array or Pandas DataFrame. The samples always represent the individual objects described by the dataset and the features represent the distinct observations that describe each sample in a quantitative manner.

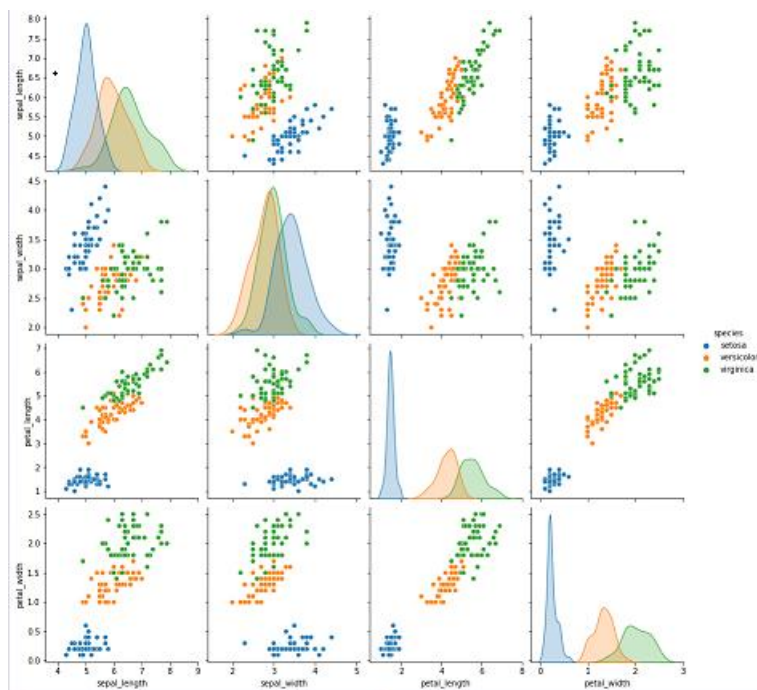
4-3- Data as Target array

It is also called **label**. it is denoted by **y**. The label or target array is usually one-dimensional having length n_samples. It is generally contained in NumPy **array** or Pandas **Series**. Target array may have both the values, continuous numerical values and discrete values.

we can distinguish target array from feature columns by one point that the target array is usually the quantity we want to predict from the data. in statistical term it is the dependent variable.

in the example, from iris dataset predict the species of flower based on other measurement.

```
import seaborn as sns
data=sns.load_dataset('iris')
sns.pairplot(data,hue='species',height=3)
```



output:

5- Estimator API

It is one of the main APIs implemented by Scikit-learn. It provides a consistent interface for a wide range of ML applications that's why all machine learning algorithms in Scikit-Learn are implemented via Estimator API. The object that learns from the data (fitting the data) is an estimator. It can be used with any of the algorithms like classification, regression, clustering or even with a transformer, that extracts useful features from row data.

For fitting the data, all estimator objects expose a fit method that takes a dataset shown as follows:

```
estimator.fit(data)
```

Next, all the parameters of an estimator can be set, as follows, when it is instantiated by the corresponding attribute:

```
estimator= Estimator (param1=1, param2=2)
```

Once data is fitted with an estimator, parameters are estimated from the data at hand.

5-1- Use of Estimator API

Estimator object is used for estimation and decoding of a model. Furthermore, the model is estimated as a deterministic function of the following:

- The parameters which are provided in object construction.
- The global random state(`numpy.random`) if the estimator's `random_state` parameter is set to none.
- Any data passed to the most recent call to **fit, fit_transform, fit_predict**
- Any data passed in a sequence of calls to **partial_fit**

5-2- Steps in using Estimator API 1. Choose a class of model 1. Choose model hyperparameters 1. Arranging the data 1. Model Fitting 1. Applying the model

6- Scikit Learn Conventions

Scikit-learn's objects share a uniform basic API that consists of the following three complementary interfaces:

- Estimator interface: it is for building and fitting the models.
- Predictor interface: it is for making predictions.
- Transformer interface: it is for converting data.

Various Conventions 6-1- Type Casting

It states that the input should be cast to float64.

```
import numpy as np
from sklearn import random_projection
rannage=np.random.RandomState(0)
X=rannage.rand(10,2000)
X=np.array(X,dtype='float32')
print(X.dtype)
Transformer_data=random_projection.GaussianRandomProjection()
X_new=Transformer_data.fit_transform(X)
print(X_new.dtype)
```

output:

float32

float64

6-2- Refitting and Updating Parameters

Hyper-parameters of an estimator can be updated and refitted after it has been constructed via the `set_params()`

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.svm import SVC
X,y=load_iris(return_X_y=True)
clf=SVC()
clf.set_params(kernel='linear').fit(X,y)
clf.predict(X[:5])
```

output:

```
array([0, 0, 0, 0, 0])
```

Now we can change back the kernel to rbf to refit the estimator and to make a second prediction:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.svm import SVC
X,y=load_iris(return_X_y=True)
clf=SVC()
# clf.set_params(kernel='linear').fit(X,y)
# clf.predict(X[:5])
clf.set_params(kernel='rbf',gamma='scale').fit(X,y)
clf.predict(X[:5])
```

output:

```
array([0, 0, 0, 0, 0])
```

6-3- Multiclass and Multilabel fitting

In case of multiclass fitting, both learning and the prediction takes are dependent on the format of the target data fit upon. The module used is **sklearn.multiclass**.

```
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import LabelBinarizer
X=[[1,2],[3,5],[4,8],[1,1],[5,2]]
y=[0,0,1,1,2]
classif=OneVsRestClassifier(estimator=SVC(gamma='scale',random_state=0))
classif.fit(X,y).predict(X)
```

output:

```
array([0, 0, 1, 1, 2])
```

```
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import LabelBinarizer
X=[[1,2],[3,5],[8,4],[1,2],[5,3]]
y=LabelBinarizer().fit_transform(y)
classif.fit(X,y).predict(X)
```

output:

```
array([[0, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

7- Scikit Learn - Linear Modeling Let us begin by understanding what is Linear Regression in Sklearn.

7-1: Linear Regression studies the relationship between a dependent variable(Y) with a given set of independent variables(X).

7-2: Logistic Regression is a classification algorithm rather than regression algorithm. Based on a given set of independent variables, it is used to estimate discrete value(0 or 1, yes/no, true/false)

7-3: Ridge Regression or Tikhonov regularization technique that perform L2 regularization. It modifies the loss function by adding the penalty equivalent to the square of the magnitude of coefficients.

7-4: Bayesian Ridge Regression allows a natural mechanism to survive insufficient data or poorly distributed data by formulating linear regression using probability distributors rather than point estimates.

7-5: LASSO is the regularization technique that perform L1 regularization. It modifies the loss function by adding the penalty equivalent to the summation of the absolute value of coefficients.

7-6: Multi-task LASSO allows to fit multiple regression problems jointly enforcing the selected features to be same for all the regression problems, also called tasks. Sklearn provides a linear model named MultiTaskLasso, trained with a mixed L1, L2-norm for regularization, which estimates sparse coefficients for multiple regression problems jointly.

7-7: Elastic-Net is a regularized regression method that linearly combines both penalties. It is useful when there are multiple correlated features.

7-8: Multi-task Elastic-Net allows to fit multiple regression problems jointly enforcing the selected features to be same for all the regression problems, also called tasks.

8- Stochastic Gradient Descent

SGD is an optimization algorithm in Sklearn. This algorithm used to find the values of parameters/coefficients of functions that minimize a cost function. It has been successfully applied to large-scale datasets because the update to the coefficients is performed for each training instance, rather than at the end of instance.

Stochastic Gradient Descent classifier basically implements a plain SGD learning routine supporting various loss functions and penalties for classification. Scikit-learn provides **SGDClassifier** module to implement SGD classification.

8-1- Implementation Example

Like other Classifiers, SGD has to be fitted with following two arrays:

- An array X holding the training samples. It is of size[n_samples,n_features]
- An array Y holding the target values. It is of size[n_samples]

```
import numpy as np
from sklearn import linear_model
X=np.array([[ -1,1],[ -2, -3],[2,3],[1,2]])
Y=np.array([1,2,5,1])
SGDCLF=linear_model.SGDClassifier(max_iter=1000,tol=1e-3,penalty='elasticnet')
SGDCLF.fit(X,Y)
print(SGDCLF.predict([[2,1]]))
print(SGDCLF.coef_)
```

output:

```
[5]
[[-19.33273544  38.59729232]
 [-19.54811198 -29.32421683]
 [ 37.79761953  -9.39254474]]
```

Scikit-learn provides **SGDRegressor** module to implement SGD regression.

9- Support Vector Machines

Support Vector Machines(SVM) used for classification, regression, and , outlier's detection. SVMs are very efficient in high dimensional spaces and generally are used in classification problems.

The main goal of SVMs is to divide the datasets into number of classes in order to find a *maximum marginal hyperplane(MMH)* which can done in the following two steps:

- Support Vector Machines will first generate hyperplanes iteratively that separates the classes in the best way.
- After that it will choose the hyperplane that separate the classes correctly.

Scikit-learn provides three classes namely **SVC,NuSVC,LinearSVC** which can perform multi-class classification.

9-1: SVC

This module used by scikit-learn is **sklearn.svm.svc**. This class handles the multiclass support according to one-vs-one scheme.

9-1-1:Parameters

- **C**: it is the penalty parameter of the error term, default=0
- **kernel**: it specifies the type of kernel to be used in algorithm, we can choose among 'linear','poly','rbf','sigmoid','precomputed'.
- **degree**: it represents the degree of the 'poly' kernel function and will be ignored by all other kernels.
- **gamma**: it is kernel coefficient for kernels 'rbf','poly' and 'sigmoid'.
- **coef0**: an independent term in kernel function which is only significant in 'poly' and 'sigmoid'.
- **tol**: this parameter represents the stopping criterion for iterations.
- **shrinking**: this parameter represents that whether we want to use shrinking heuristic or not.
- **probability**: this parameter enables or disables probability estimates.
- **cache_size**: this parameter will specify the size of the kernel cache. The value will be in MB(MegaBytes).

- **random_state**: this parameter represents the seed of the psedu random number generated which is used while shuffling the data.
- **class_weight**: this parameter will set the parameter C of class j to $class_weight[j]*C$ for SVC.

9-1-2:Implementation Example

```
import numpy as np
from sklearn.svm import SVC
X=np.array([[ -1,1],[ -2,-3],[2,3],[1,2]])
y=np.array([1,2,2,1])
SVCCLF=SVC(kernel='linear',gamma='scale',shrinking=False)
SVCCLF.fit(X,y)
print(SVCCLF.predict([[ -0.5,-0.8]]))
print(SVCCLF.coef_)
```

output:

```
[2]
[[ 0.58823529 -0.64705882]]
```

9-2: NuSVC

It is another class provided by scikit-learn which can perform multi-class classification. it is like SVC but NuSVC accepts slightly different sets of parameters:

- **nu** represents an upper bound on the fraction of training errors and a lower bound of the fraction of suport vectors. its value should be in the interval of (0,1].

9-2-1:Implementation Example

```
import numpy as np
from sklearn.svm import NuSVC
X=np.array([[ -1,1],[ -2,-3],[2,3],[1,2]])
y=np.array([1,2,2,1])
NuSVCCLF=NuSVC(kernel='linear',gamma='scale',shrinking=False)
NuSVCCLF.fit(X,y)
print(NuSVCCLF.predict([[ -0.5,-0.8]]))
print(NuSVCCLF.coef_)
```

output:

```
[2]
[[ 2.          -1.33333333]]
```

9-3: LinearSVC

It is Linear Support Vector Classification. It is similar to SVC having kernel='linear'. The difference between them is that **LinearSVC** implemented in terms of liblinear while SVC is implemented in **libsvm**. That's the reason **LinearSVC** has more flexibility in the choice of penalties and loss functions. It also scales better to large number of samples.

9-3-1:Implementation Example

```
import numpy as np
from sklearn.svm import LinearSVC
X=np.array([[ -1,1],[ -2,-3],[2,3],[1,2]])
y=np.array([1,2,2,1])
LSVCCLf=LinearSVC(dual=False,random_state=0,penalty='l1',tol=1e-5)
LSVCCLf.fit(X,y)
print(LSVCCLf.predict([[ -0.5,-0.8]]))
print(LSVCCLf.coef_)
```

output:

```
[2]
[[ 0.44804731 -0.38311391]]
```

9-4: Regression with SVM

SVM is used for both classification and regression problems. Scikit-learn's method of Support Vector Classification (SVC) can be extended to solve regression problems as well. That extended method is called Support Vector Regression (SVR).

Scikit-learn provides three classes namely **SVR**,**NuSVR** and **LinearSVR** as three different implementation of SVR.

9-4-1:SVR

It is epsilon-support vector regression whose implementation is based on **libsvm**. here, parameter **epsilon** represents the epsilon-SVR model, and specifies the epsilon-tube within within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

```

from sklearn.svm import SVR
X=[[1,2],[3,4]]
y=[1,3]
SVRREG=SVR(kernel='linear',gamma='auto')
SVRREG.fit(X,y)
print(SVRREG.predict([[1,2]]))

```

output:

```
[1.1]
```

9-4-2: NuSVR

It is like NuSVC, but NuSVR uses a parameter **nu** to control the number of support vectors. And moreover, unlike NuSVC where **nu** replaced C parameter, here it replaces **epsilon**.

```

from sklearn.svm import NuSVR
import numpy as np
n_samples,n_features=20,15
np.random.seed(0)
X=np.random.randn(n_samples,n_features)
y=np.random.randn(n_samples)
NuSVRReg=NuSVR(kernel='linear',gamma='auto',C=1.0,nu=0.1)
NuSVRReg.fit(X,y)
print(NuSVRReg.coef_)

```

output:

```

[[ 0.23180317  0.13549427 -0.38437291  0.21767392 -0.67694681 -0.01261807
 -0.2347462   0.16879733 -0.35119401 -0.21719007 -0.14536424  0.05250018
  0.18204661  0.08182123  0.26425696]]

```

9-5: LinearSVR

It is similar to SVR having kernel='linear'. The difference between them is that LinearSVR implemented in terms of **liblinear**. while SVC implemented in **libsvm**. That's the reason **LinearSVR** has more flexibility in the choice of penalties and loss functions. It also scales better to large number of samples. Also it does not support **'kernel'** because it is assumed to be linear.

```

from sklearn.svm import LinearSVR
from sklearn.datasets import make_regression
X,y=make_regression(n_features=4,random_state=0)
LSVRReg=LinearSVR(dual=False,random_state=0, loss='squared_epsilon_insensitive',tol=1e-5)
LSVRReg.fit(X,y)
print(LSVRReg.predict([[0,1,2,1]]))
print(LSVRReg.coef_)

```

output:

```

[256.00973816]
[20.47354746 34.08619401 67.23189022 87.47017787]

```

10- Scikit-Learn: Anomaly Detection

Anomaly Detection is a technique used to identify data points in dataset that does not fit well with the rest of the data. Two methods namely **outlier detection** and **novelty detection** can be used for anomaly detection.

- **outlier detection:** The training data contains outliers that are far from the rest of the data. Such outliers are defined as observations. That's the reason, outlier detection estimators always try to fit the region having most concentrated training data while ignoring the deviant observations.
- **Novelty detection:** It is concerned with detecting an unobserved pattern in new observations which is not included in training data. Here, the training data is not polluted by outliers.

There are set of ML tools, provided by scikit-learn, which can be used for both outlier detection as well as novelty detection. These tools first implementing object learning from the data using `fit()` method as follows:

```
estimator.fit(X_train)
```

Now new observations would be sorted by using `predict()` method:

```
estimator.predict(X_test)
```

10-1-Sklearn algorithms for Outlier Detection

- **Fitting an elliptic envelop:** This algorithm assume that regular data comes from a known distribution such as Gaussian distribution. For Outlier detection, Scikit-learn provides an object named **covariance.EllipticEnvelop**. This object fits a robust covariance estimate to the data, and thus, fits an ellipse to the central data points. It ignores the points outside the central mode.

```
import numpy as np
from sklearn.covariance import EllipticEnvelope
data=np.array([[5,6],[6,9]])
X=np.random.RandomState(0).multivariate_normal(mean=[0,0],cov=data,size=500)
cov=EllipticEnvelope(random_state=0).fit(X)
cov.predict([[-5,-3],[2,4]])
```

output: array([-1, 1])

- **Isolation Forest:** In case of high-dimensional dataset, one efficient way for outlier detection is to use random forests. Scikit-learn provides **ensemble.IsolationForest** method that isolates the observations by randomly selecting a feature. Afterwards, it randomly selects a value between the maximum and minimum values of the selected features.

```
from sklearn.ensemble import IsolationForest
import numpy as np
X=np.array([[-1,-2],[-2,-3],[0,0],[-6,8]])
OUTCLF=IsolationForest(n_estimators=10)
OUTCLF.fit(X)
OUTCLF.predict([[-5,3],[0,5]])
```

output: array([1, 1])

- **Local Outlier Factor:** LOF algorithm is another to perform outlier detection on high dimension data. Scikit-learn provides **neighbors.LocalOutlierFactor** method that computes a score, called local outlier factor, reflecting the degree of anomaly of the observations. The main logic of this algorithm is to detect the samples that have a substantially lower density than its neighbors.

```
from sklearn.neighbors import NearestNeighbors
X=([[1,2,6],[0,5,8],[6,3,2]])
LOFneigh=NearestNeighbors(n_neighbors=1,algorithm='ball_tree',p=1)
LOFneigh.fit(X)
LOFneigh.kneighbors([[0,3,1.5]])
```

output:

(array([[6.5]]), array([[0]], dtype=int64))

- **One-Class SVM:** This algorithm is very effecient in high-dimensional data and estimates the support of a high dimensional distribution. It is implemented in the **Support Vector Machine** module in the **sklearn.svm.OneClassSVM** object.

```
from sklearn.svm import OneClassSVM
X=[[0],[0.91],[0.8],[1]]
SVMclf=OneClassSVM(gamma='scale').fit(X)
SVMclf.score_samples(X)
```

output: array([0.94999136, 0.99990868, 0.94999136, 0.95040106])

11- K-Nearest Neighbors (KNN)

Neighbor based learning method are both types namely **supervised** and **unsupervised**. The main principle behind nearest neighbor method is:

- To find a predefined number of training sample closest in distance to the new data point
- Predict the label from these number of training samples

Here, the number of samples can be a user-defined constant like in K-nearest neighbor learning or very based on the local density of point like in radius-based neighbor learning. For this, Scikit-learn have **sklearn.neighbors** module.

11-1- Types of algorithms

Different types of algorithms which can be used in neighbor-based methods implementation are as follows:

- **Brute Force:** The brute-force computation of distance between all pairs of points in the dataset provides the most naive neighbor search implementation. for N samples in D dimensions, brute-force approach scales as $O[DN^2]$. For small data samples, this algorithm can be very useful, but it becomes infeasible as and when number of samples grows. Brute-Force neighbor search can be enabled by writing keyword **algorithm='brute'**.
- **K-D Tree:** K-D Tree is a binary tree structure which is called K-dimensional tree. It recursively partitions the parameters space along the data axes by dividing it into nested orthographic regions into which the data points are filled. It have been invented to address the computational inefficiencies of the brute-force approach. This algorithm takes very less distance computations to determine the nearest neighbor of a query point and takes $O[\log(N)]$ distance computations. K-D tree neighbor searches can be enabled by writing the keyword **algorithm='kd_tree'**.
- **Ball Tree:** KD Tree is inefficient in higher dimensions. For this, Ball Tree was developed. This algorithm recursively divides the data, into nodes defined by a centroid C and radius r. It uses triangle inequality which reduces the number of candidate points for a neighbor search:

$$|X+Y| \leq |X| + |Y|$$

Ball Tree neighbor searches can be enabled by writing keyword **algorithm='ball_tree'**.

11-2- Choosing Nearest Neighbors Algorithm

There are most important factors to be considered while choosing Nearest Neighbor algorithm:

- The query time of Brute Force algorithm grows as $O[D(N)]$
- The query time of Ball Tree algorithm grows as $O[D \log(N)]$
- The query time of KD Tree algorithm changes with D in a strange manner: when $D < 20$, the cost is $O[D \log(N)]$ and when $D > 20$ cost increase to nearly $O[DN]$
- The query times of Ball Tree and KD Tree algorithms can be greatly influenced by intrinsic dimensionality of the data or sparsity of the data, Whereas, the query time of Brute Force algorithm is unchanged by data structure. The query time of Ball Tree and KD Tree becomes slower as number of neighbors (k) increases.

KNN is non-parametric and lazy in nature. Non-parametric means that there is no assumption for the underlying data distribution i.e. the model structure is determined from the dataset. Lazy or instance-based learning means that for the purpose of model generation, it does not require any training data points and whole training data is used in the testing phase.

The KNN algorithm consist of two steps:

1. in this step, it computes and stores the k nearest neighbors for each sample in the training set.
2. in this step, for an unlabeled sample, it retrieves the k nearest neighbors from dataset. Then among these k-nearest neighbors, it predicts the class through voting(class with majority votes wins).

sklearn.neighbors.NearestNeighbors is the module used to implement unsupervised nearest neighbor learning. It uses nearest neighbor algorithms named BallTree, KDTree or Brute Force.

11-3- Supervised KNN Learning

The supervised neighbors-based learning is used for following:

- Classification, for the data with discrete labels.
- Regression, for the data with continuous labels.

11-4- Implementation Example

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

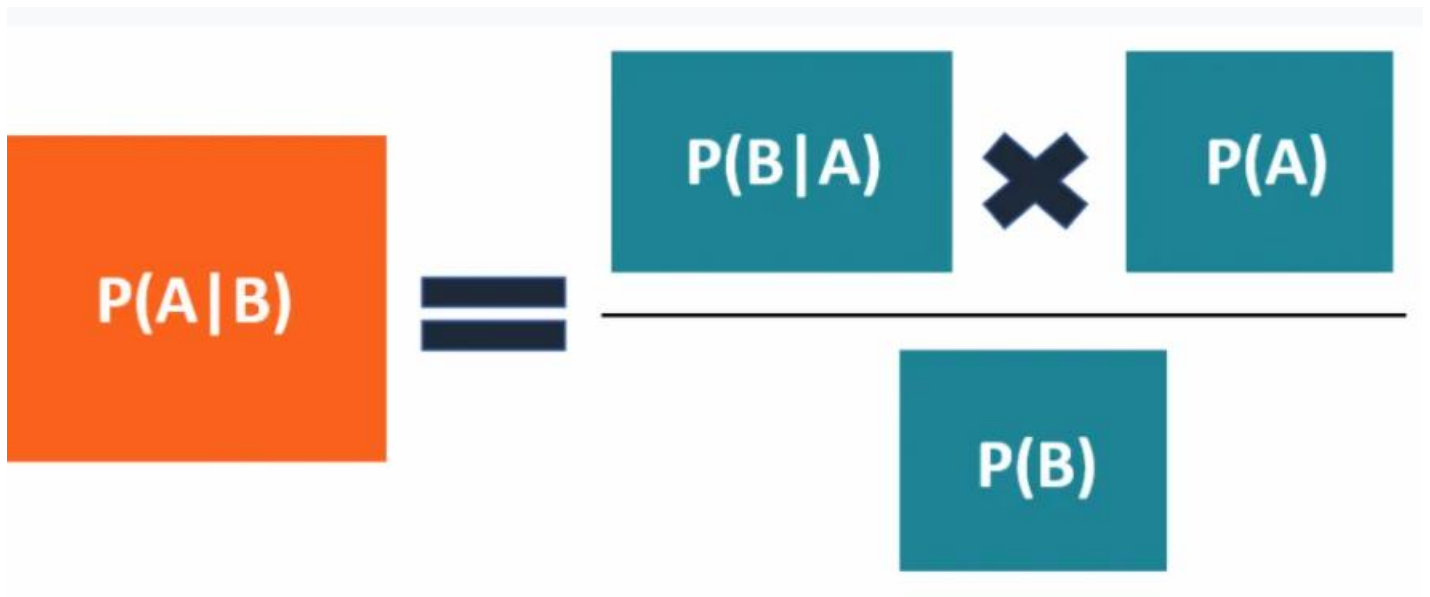
iris=load_iris()
X=iris.data[:, :4]
y=iris.target
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3)
scaler=StandardScaler()
scaler.fit(X_train)
X_train=scaler.transform(X_train)
X_test=scaler.transform(X_test)
knnr=KNeighborsRegressor(n_neighbors=5)
knn=knnr.fit(X_train,y_train)
print ("The MSE is:",format(np.power(y-knnr.predict(X),4).mean()))
```

output: The MSE is: 3.7536533333333333

12-Scikit Learn- Classification with Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes theorem with a strong assumption that all the predictors are independent to each other.

Bayes theorem is a mathematical formula used to determine the condition probability of events. Essentially, the Bayes theorem describes the **probability** of an event based on **prior knowledge of the conditions** that might be relevant to the event.



The diagram illustrates Bayes' Theorem using colored boxes and mathematical symbols. On the left, an orange box contains the expression $P(A|B)$. This is followed by an equals sign. To the right of the equals sign is a fraction. The numerator of the fraction consists of two teal boxes: the first contains $P(B|A)$ and the second contains $P(A)$, with a large dark blue multiplication symbol (\times) between them. A horizontal line separates the numerator from the denominator, which is a single teal box containing $P(B)$.

where:

- $P(A|B)$: the probability of event A occurring, given event B has occurred.
- $P(B|A)$: the probability of event B occurring, given event A has occurred.
- $P(A)$: the probability of event A
- $P(B)$: the probability of event B

The Scikit-learn provides different naive Bayes classifier models namely **Gaussian, Multinomial, Complement and Bernoulli**.

- Gaussian Naive Bayes: this classifier assumes that the data from each label is drawn from a simple Gaussian distribution.
- Multinomial Naive Bayes : It assumes that the features are drawn from a simple Multinomial distribution.
- Bernoulli Naive Bayes: Assumption in this model is that features binary (0s and 1a) in nature.
- Complement Naive Bayes: it was designed to correct the severe assumptions made by Multinomial Bayes classifier.

12-1- Building Naive Bayes Classifier

```
import sklearn
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
data = load_breast_cancer()
label_names=data['target_names']
labels=data['target']
feature_names=data['feature_names']
feature=data['data']
train,test,train_labels,test_labels=train_test_split(feature,labels,test_size=0.40,random_state=42)
from sklearn.naive_bayes import GaussianNB
GNBclf=GaussianNB()
model=GNBclf.fit(train,train_labels)
pred=GNBclf.predict(test)
print(pred)
```

output:

```
[1 0 0 1 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0
 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 1 0 0 1 0
 1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 0 1 1 0
 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1 1 1 0 1 1 0 0 1 0 1 1 0 0 0
 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1
 0 0 1 1 0 1]
```

13- Scikit Learn- Decision Tree

Decision Tree re the most powerful non-parametric supervised earning method. They can be used for classification and regression tasks. The main goal of DTree is to create a model predicting target variable value by learning simple decision rules deduced from data features. Decision Tree have two main entities; one s root node,where the data split, and other is decision nodes or leaves,where we got final output.

13-1: Decision Tree algorithms

- **ID3**: It is also called Iterative Dichotomiser 3. The main goal of this algorithm is to find those categorical features, for every node, that will yield the largest information gain for categorical targets.
- **C4.5**: It defines a discrete attribute that partition the continuous attribute value into a discrete set of intervals. That's the reason it removed the restriction of categorical features.
- **C5**: It works similar as C4.5 but it uses less memory and build smaller rulesets. It is more accurate than C4.5.
- **CART**: It is called Classification and Regression Tree algorithm. It basically generates binary splits by using the features and threshold yielding the largest information gain at each node.

13-2: Implementation Example

```
from sklearn import tree
from sklearn.model_selection import train_test_split
X=[[14,25],[98,52],[987,254],[125,653],[214,410],[235,21],[28,54],[451,712],[111,214],[985,437],[965,439],
   [321,124],[354,842],[85,32],[963,41],[854,654],[960,370],[96,526]]
Y=['Woman','Man','Man','Man','Woman','Man','Woman','Woman','Woman','Man','Man','Woman',
   'Man','Woman','Woman','Man','Man','Man']

data_feature_name=['height','length of hair']
DTclf=tree.DecisionTreeClassifier()
clf=DTclf.fit(X,Y)
prediction=clf.predict([[125,653]])
print(prediction)
```

output:

[Man]

14-Randomized Decision Tree Algorithms

Decision Tree is usually trained by recursively splitting the data, but being prone to overfit, they have been transformed to random forests by training many trees over various sub-samples of the data. **sklearn.ensemble** having two algorithms based on randomized decision trees.

In this algorithm, each decision tree in the ensemble is built from a sample drawn with replacement from the training set and then gets the prediction from each of them and finally selects the best solution by means of voting. It can be used for both classification as well as regression tasks.

- **Classification by Random Forest**: For creating a **random forest classifier**, Scikit-learn module provides **sklearn.ensemble.RandomForestClassifier**. Main parameters are **max_features** and **n_estimators**. **max_features** is the size of the random subsets of features to consider when splitting a node. **n_estimators** are the number of trees in the forest.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
X,y=make_blobs(n_samples=10000,n_features=10,centers=100,random_state=0)
RFclf=RandomForestClassifier(n_estimators=10,min_samples_split=2)
score=cross_val_score(RFclf,X,y,cv=5)
print(score.mean())
```

output:

0.9998000000000001

- **Regression with Random Forest**: For creating a random forest regression, scikit-learn module provides **sklearn.ensemble.RandomForestRegressor**. it will use the same parameters as used by **sklearn.ensemble.RandomForestClassifier**.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestRegressor
X,y=make_blobs(n_samples=10000,n_features=10,centers=100,random_state=0)
RFclf=RandomForestRegressor(n_estimators=10,min_samples_split=2)
score=cross_val_score(RFclf,X,y,cv=5)
print(score.mean())
```

output:

0.9705321938477912

15- Extra Tree Methods

For each feature under consideration, it selects a random value for the split. benefit of using extra tree methods is that it allows to reduce the variance of the model. the disadvantage of using these methods is that slightly increases the bias.

- **Classification with Extra-Tree Method**: For creating a classifier using Extra-Tree method scikit-learn module provides **sklearn.ensemble.ExtraTreesClassifier**. It uses the same parameters as used by **sklearn.ensemble.RandomForestClassifier**.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_blobs
from sklearn.ensemble import ExtraTreesClassifier
X,y=make_blobs(n_samples=10000,n_features=10,centers=100,random_state=0)
ETclf=ExtraTreesClassifier(n_estimators=10,min_samples_split=2)
score=cross_val_score(ETclf,X,y,cv=5)
print(score.mean())
```

output:

0.9998000000000001

- **Regression with Extra-Tree Method:** For creating a Extra-Tree regression, scikit-learn module provides sklearn.ensemble.ExtraTreesRegressor. it use the same parameters as used by sklearn.ensemble.ExtraTreesClassifier.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_blobs
from sklearn.ensemble import ExtraTreesRegressor
X,y=make_blobs(n_samples=10000,n_features=10,centers=100,random_state=0)
ETclf=ExtraTreesRegressor(n_estimators=10,min_samples_split=2)
score=cross_val_score(ETclf,X,y,cv=5)
print(score.mean())
```

output:

0.9914243964799774

19-Scikit Learn-Clustering Performance Evaluation

There are many functions with the help of which can evaluate the performance of clustering algorithms. Following are some important and mostly used functions given by Scikit Learn for evaluating clustering performance:

19-1-Adjust Rand Index

Rand Index is a function that computes a similarity measures between two clustering. For this computation rand index considers all pairs of samples and counting pairs that are assigned in the similar or different clusters in the predicted and true clustering.

$$Adjusted\ RI = (RI - Expected_RI) / (max(RI) - Expected_RI)$$

It has two parameters namely **labels_true**, which is ground truth class labels, and **labels_pred**, which are clusters label to evaluate.

```
from sklearn.metrics.cluster import adjusted_rand_score

labels_true=[0,0,1,1,2]
labels_pred=[0,1,1,1,0]
print(adjusted_rand_score(labels_true,labels_pred))
```

output:

0.090

19-2- Mutual Information Based Score

Mutual Information is a function that compute the agreement of two assignments. It ignores the permutations. There are following versions available:

- **Normalized Mutual Information(NMI)**

```
from sklearn.metrics.cluster import normalized_mutual_info_score

labels_true=[0,0,1,1,2]
labels_pred=[0,1,1,1,0]
print(normalized_mutual_info_score(labels_true,labels_pred))
```

output:

0.45

- **Adjusted Mutual Information(AMI)**

```
from sklearn.metrics.cluster import adjusted_mutual_info_score

labels_true=[0,0,1,1,2]
labels_pred=[0,1,1,1,0]
print(adjusted_mutual_info_score(labels_true,labels_pred))
```

output:

0.10

19-3-Fowlkes-Mallows Score

This function measures the similarity of two clustering of a set of points. It may be defined as the geometric mean of the pairwise precision and recall.

$$FMS = \frac{TP}{\sqrt{(TP + FP)(TP + FN)}}$$

TP=True Positive: number of pair of points belonging to the same clusters in true as well as predicted labels both.

FP=False Positive: number of pair of points belonging to the same clusters in true labels but not in the predicted labels.

FN=False Negative: number of pair of points belonging to the same clusters in the predicted labels but not in the true labels.

```
from sklearn.metrics.cluster import fowlkes_mallows_score

labels_true=[0,0,1,1,2]
labels_pred=[0,1,1,1,0]
print(fowlkes_mallows_score(labels_true,labels_pred))
```

output:

0.35

19-4-Silhouette Coefficient

This function will compute the mean Silhouette Coefficient of all samples using the mean intra-cluster distance and the mean nearest-cluster distance for each sample.

$$S=(b-a)/\max(a,b)$$

a is intra-cluster distance, b is nearest-cluster distance.

```
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances
from sklearn import datasets
import numpy as np
from sklearn.cluster import KMeans
data=datasets.load_iris()
X=data.data
y=data.target
kmeans_model=KMeans(n_clusters=3,random_state=1).fit(X)
labels=kmeans_model.labels_
print(silhouette_score(X,labels,metric='euclidean'))
```

output:

0.55

20- Dimensionality Reduction using PCA

Dimensionality reduction is one of the popular algorithms for dimensionality reduction, an unsupervised machine learning method is used to reduce the number of feature variable for each data sample selecting set of principal features.

PCA is used for linear dimensionality reduction using **Singular Value Decomposition** of the data to project it to a lower dimensional space.

Scikit learn provides **sklearn.decomposition.PCA** module that is implemented as a transformer object which learns n components in its fit() method. in this example PCA used to find best 5 principal components.

```
import pandas as pd
from pandas import read_csv
from sklearn.decomposition import PCA
data= pd.read_csv('pima-indians-diabetes.csv')
data=pd.DataFrame(data)
data.columns=['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
array=dataframe.values
X=array[:,0:8]
Y=array[:,8]
pca=PCA(n_components=5)
fit=pca.fit(X)
print(fit.components_)
```

output:

```
[[-2.00650656e-03  9.80841939e-02  1.61268541e-02  6.08979238e-02
  9.93074929e-01  1.40318199e-02  5.38629887e-04 -3.44019614e-03]
 [-2.25771000e-02 -9.72225272e-01 -1.41792542e-01  5.92457341e-02
  9.48278991e-02 -4.68497182e-02 -8.07348060e-04 -1.39517868e-01]
 [-2.24493310e-02  1.42870454e-01 -9.22881602e-01 -3.06122095e-01
  2.10400193e-02 -1.32404531e-01 -6.35231522e-04 -1.25255048e-01]
 [-4.95290159e-02  1.21003701e-01 -2.61006207e-01  8.83400591e-01
 -6.57727689e-02  1.93313614e-01  2.68446301e-03 -3.04435083e-01]
 [ 1.51601087e-01 -8.66920903e-02 -2.32633508e-01  2.63333062e-01
 -6.48631917e-04  2.34504383e-02  1.63556108e-03  9.19503085e-01]]
```

20-1- Incremental PCA

Incremental Principal Component Analysis (IPCA) is used to address biggest limitation of PCA and that s PCA only supports batch processing,means all the input data to be processed should fit in the memory. The Scikit-learn provides **sklearn.decomposition.IPCA** module that makes it possible to implement Out-of-Core PCA either by using it partial_fit method on sequentially fetched chunks of data or by enabling use of np.memmap, a memory mapped file, without loading the entire file into memory.

20-2-Kernel PCA

Kernel Prinipal Component Analysis, an extension of PCA, achives on-linear dimensionality reduction using kernels.It supports both ansform and inverse_transform. Scikit-learn provides **sklearn.decomposition.kernelIPCA** module.