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 anomality 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