DATA MINING
FINAL PROJECT

Full Name: Karthik Chandrashekar
NJIT UCID: Kc69
Email: kc69@njit.edu

SUPERVISED DATA MINING

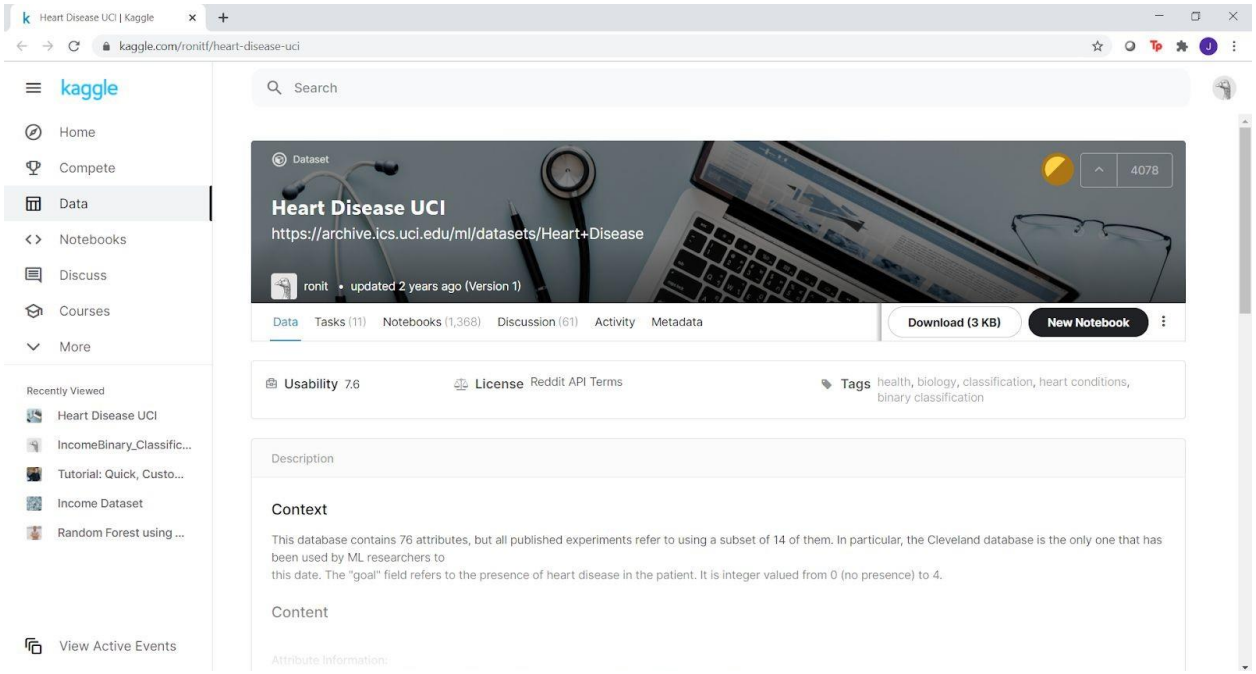
Category 1: Support Vector Machine
Category 2: Random Forest
Category 3: K Nearest Neighbour
Programming Language: Python
Tools: Jupyter Notebook

Dataset: https://www.kaggle.com/ronitf/heart-disease-uci

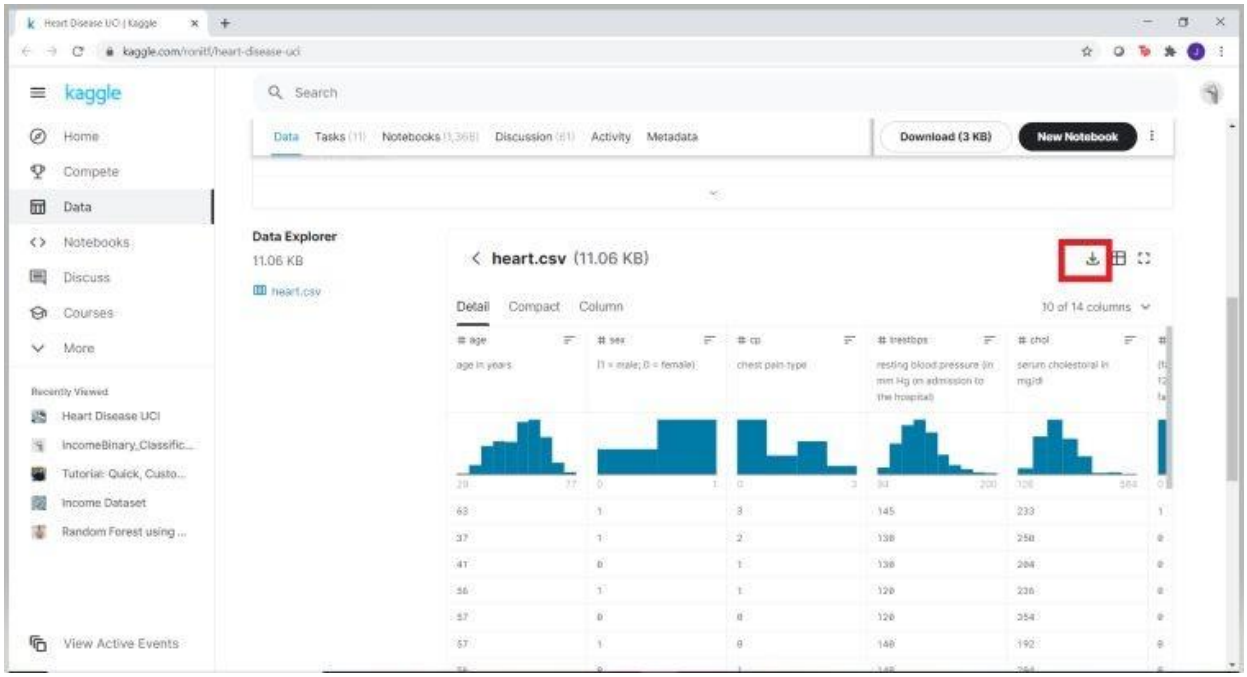
This dataset is a classification dataset about heart disease based on various factors such as:

1. Age
2. Sex
3. Chest pain type (4 values)
4. Resting blood pressure
5. Serum cholesterol in mg/dl
6. Fasting blood sugar > 120 mg/dl
7. Resting electrocardiographic results (values 0,1,2)
8. Maximum heart rate achieved
9. Exercise-induced angina
10. oldpeak = ST depression induced by exercise relative to rest
11. The slope of the peak exercise ST segment
12. Number of major vessels (0-3) colored by fluoroscopy
13. thal: 3 = normal;6 = fixed defect; 7 = reversible defect

To download the dataset, I clicked the URL mention above, and the following will be seen:



Then I scrolled down, clicked on the download option, and downloaded the dataset.



The downloaded data is stored in CSV format.

This is how the dataset “heart.csv” looks. *Please note* this is not the full dataset, it’s just a snippet to show how the data looks.

```
age,sex,cp,trestbps,chol,fbs,restecg,thalach,exang,oldpeak,slope,ca,thal,target
63,1,3,145,233,1,0,150,0,2.3,0,0,1,1
37,1,2,130,250,0,1,187,0,3.5,0,0,2,1
41,0,1,130,204,0,0,172,0,1.4,2,0,2,1
56,1,1,120,236,0,1,178,0,0.8,2,0,2,1
57,0,0,120,354,0,1,163,1,0.6,2,0,2,1
57,1,0,140,192,0,1,148,0,0.4,1,0,1,1
56,0,1,140,294,0,0,153,0,1.3,1,0,2,1
44,1,1,120,263,0,1,173,0,0,2,0,3,1
52,1,2,172,199,1,1,162,0,0.5,2,0,3,1
57,1,2,150,168,0,1,174,0,1.6,2,0,2,1
54,1,0,140,239,0,1,160,0,1.2,2,0,2,1
48,0,2,130,275,0,1,139,0,0.2,2,0,2,1
49,1,1,130,266,0,1,171,0,0.6,2,0,2,1
64,1,3,110,211,0,0,144,1,1.8,1,0,2,1
58,0,3,150,283,1,0,162,0,1,2,0,2,1
50,0,2,120,219,0,1,158,0,1.6,1,0,2,1
58,0,2,120,340,0,1,172,0,0,2,0,2,1
66,0,3,150,226,0,1,114,0,2.6,0,0,2,1
43,1,0,150,247,0,1,171,0,1.5,2,0,2,1
69,0,3,140,239,0,1,151,0,1.8,2,2,2,1
59,1,0,135,234,0,1,161,0,0.5,1,0,3,1
44,1,2,130,233,0,1,179,1,0.4,2,0,2,1
42,1,0,140,226,0,1,178,0,0,2,0,2,1
61,1,2,150,243,1,1,137,1,1,1,0,2,1
40,1,3,140,199,0,1,178,1,1.4,2,0,3,1
71,0,1,160,302,0,1,162,0,0.4,2,2,2,1
59,1,2,150,212,1,1,157,0,1.6,2,0,2,1
51,1,2,110,175,0,1,123,0,0.6,2,0,2,1
65,0,2,140,417,1,0,157,0,0.8,2,1,2,1
53,1,2,130,197,1,0,152,0,1.2,0,0,2,1
41,0,1,105,198,0,1,168,0,0,2,1,2,1
65,1,0,120,177,0,1,140,0,0.4,2,0,3,1
44,1,1,130,219,0,0,188,0,0,2,0,2,1
54,1,2,125,273,0,0,152,0,0.5,0,1,2,1
51,1,3,125,213,0,0,125,1,1.4,2,1,2,1
46,0,2,142,177,0,0,160,1,1.4,0,0,2,1
54,0,2,135,304,1,1,170,0,0,2,0,2,1
54,1,2,150,232,0,0,165,0,1.6,2,0,3,1
65,0,2,155,269,0,1,148,0,0.8,2,0,2,1
65,0,2,160,360,0,0,151,0,0.8,2,0,2,1
51,0,2,140,308,0,0,142,0,1.5,2,1,2,1
48,1,1,130,245,0,0,180,0,0.2,1,0,2,1
45,1,0,104,208,0,0,148,1,3,1,0,2,1
```

Project:

This project runs heart.csv with SVM and Random forest. I am using numpy, pandas, and cross_val_score from sklearn.model_selection. The whole project is shown below.

Heart_Disease - Jupyter Notebo

localhost:8888/notebooks/Python%20Projects/Heart_Disease.ipynb

jupyter Heart_Disease Last Checkpoint: 10/26/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score
```

In [2]:
```
data= pd.read_csv("heart.csv")
```

In [3]:
```
data.head()
```

Out[3]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

In [4]:
```
data.isnull().sum().values.sum()
```

Out[4]: 0

In [5]:
```
x = data.iloc[:,:-1].values
y = data.iloc[:,-1].values
```

In [6]:
```
print(x.shape)
print(y.shape)
```

```
(303, 13)
(303,)
```

In [7]:
```
from sklearn.model_selection import train_test_split
```

Heart_Disease - Jupyter Notebo

localhost:8888/notebooks/Python%20Projects/Heart_Disease.ipynb

jupyter Heart_Disease Last Checkpoint: 10/26/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

In [7]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.20)
```

In [8]:
```
print(x_train.shape)
print(x_test.shape)
```

```
(242, 13)
(61, 13)
```

In [9]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

SVM

In [10]:
```
from sklearn.svm import SVC
classifier= SVC(kernel='linear', random_state= 0)
classifier.fit(x_train,y_train)
```

Out[10]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='linear', max_iter=-1, probability=False, random_state=0,
    shrinking=True, tol=0.001, verbose=False)
```

In [11]:
```
y_pred= classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

In [15]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for SVM", accuracy_score(y_test,y_pred))
```

Heart_Disease - Jupyter Notebo

localhost:8888/notebooks/Python%20Projects/Heart_Disease.ipynb

jupyter Heart_Disease Last Checkpoint: 10/26/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

In [16]:
```
#Cross Validation
scores = cross_val_score(classifier, x_train, y_train, cv=10)
print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
 Accuracy score for SVM using 10-fold cross validation: 0.83 (+/- 0.18)
```

Random Forest

In [17]:
```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(x_train, y_train)
```

Out[17]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False)
```

In [18]:
```
y_pred= classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

In [20]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))
```

```
[[19  5]
 [ 7 30]]
Accuracy score for Random Forest 0.8032786885245902
```

Heart_Disease - Jupyter Notebo

localhost:8888/notebooks/Python%20Projects/Heart_Disease.ipynb

jupyter Heart_Disease Last Checkpoint: 10/26/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

```
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False)
```

In [18]:
```
y_pred= classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

In [20]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))
```

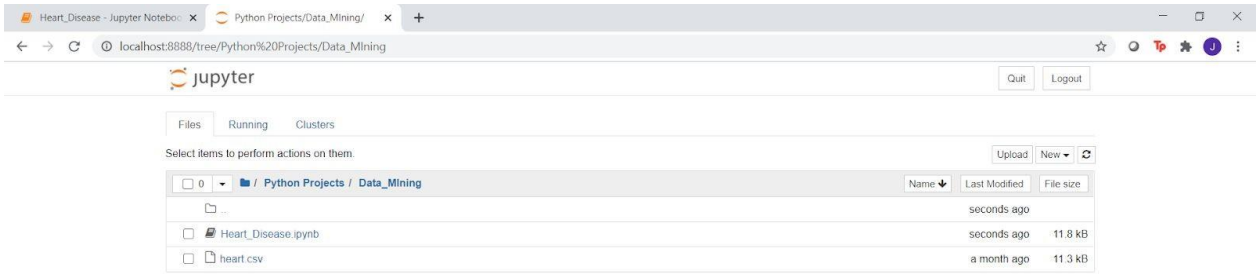
```
[[19  5]
 [ 7 30]]
Accuracy score for Random Forest 0.8032786885245902
```

In [21]:
```
#Cross Validation
scores = cross_val_score(classifier, x_train, y_train, cv=10)
print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

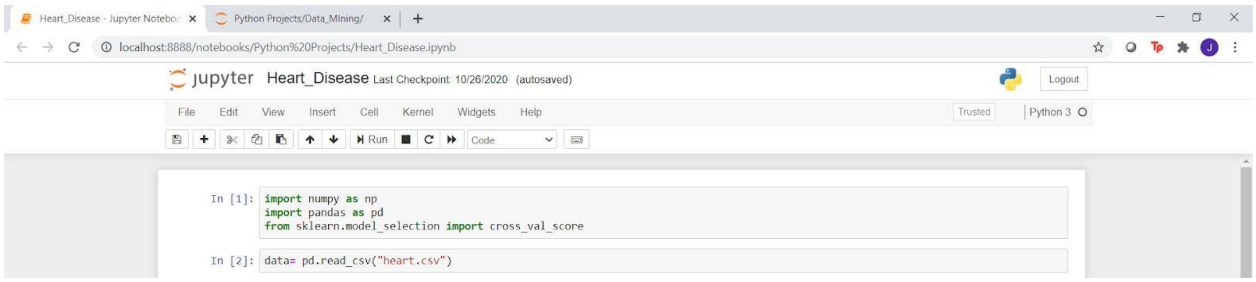
```
 Accuracy score for Random Forest using 10-fold cross validation: 0.80 (+/- 0.20)
```

In [ ]:

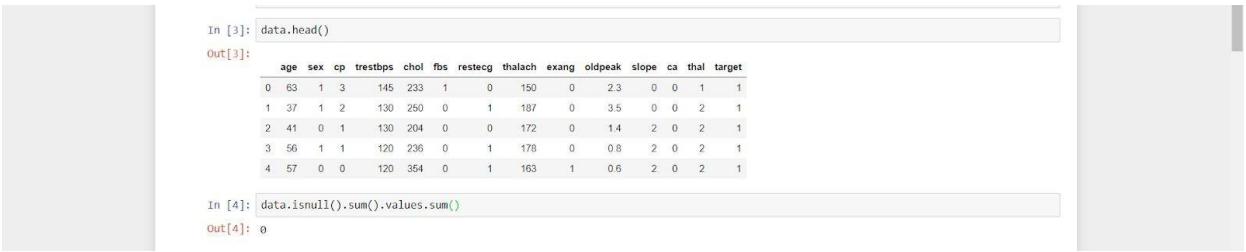
Since I used Jupyter notebook to run the project, keep into consideration to place your CSV file and project file into the same folder.



1. I first imported the required libraries, loaded the data using pandas.read_csv, and then stored the CSV data into a data frame.



2. Then I displayed the first 5 rows of the dataset using head() function. Also I checked if the dataset consists of any NULL values using isnull().



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

3. Separating and storing the dependent and independent variables in x and y.

4. Using train_test_split, the dataset is split into a training set and test set.

5. Feature scaling on the split data using StandardScaler from sklearn.preprocessing.

```
In [9]: from sklearn.preprocessing import StandardScaler
        sc = StandardScaler()
        x_train = sc.fit_transform(x_train)
        x_test = sc.transform(x_test)
```

SVM

6. From sklearn.svm I used SVC. I used the kernel type as linear and the random state as 0 to train the SVC model on the training set

```
SVM

In [10]: from sklearn.svm import SVC
         classifier= SVC(kernel='linear', random_state= 0)
         classifier.fit(x_train,y_train)

Out[10]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
             kernel='linear', max_iter=-1, probability=False, random_state=0,
             shrinking=True, tol=0.001, verbose=False)

In [11]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [15]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for SVM", accuracy_score(y_test,y_pred))

         [[17  7]
          [ 3 34]]
         Accuracy score for SVM 0.8360655737704918

In [16]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

         Accuracy score for SVM using 10-fold cross validation: 0.83 (+/- 0.18)
```

7. After training the model, I predicted the Test set results using predict().

```
SVM

In [10]: from sklearn.svm import SVC
         classifier= SVC(kernel='linear', random_state= 0)
         classifier.fit(x_train,y_train)

Out[10]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
             kernel='linear', max_iter=-1, probability=False, random_state=0,
             shrinking=True, tol=0.001, verbose=False)

In [11]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [15]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for SVM", accuracy_score(y_test,y_pred))

         [[17  7]
          [ 3 34]]
         Accuracy score for SVM 0.8360655737704918

In [16]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

         Accuracy score for SVM using 10-fold cross validation: 0.83 (+/- 0.18)
```

8. Then I Evaluated the Model Performance and got an accuracy of 83.60%

```
SVM

In [10]: from sklearn.svm import SVC
         classifier= SVC(kernel='linear', random_state= 0)
         classifier.fit(x_train,y_train)

Out[10]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
             kernel='linear', max_iter=-1, probability=False, random_state=0,
             shrinking=True, tol=0.001, verbose=False)

In [11]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [15]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for SVM", accuracy_score(y_test,y_pred))

         [[17  7]
          [ 3 34]]
         Accuracy score for SVM 0.8360655737704918

In [16]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

          Accuracy score for SVM using 10-fold cross validation: 0.83 (+/- 0.18)
```

9. In this step, I performed 10-fold cross-validation, and the accuracy score as 0.83 with a standard deviation of +/- 0.18.

```
SVM

In [10]: from sklearn.svm import SVC
         classifier= SVC(kernel='linear', random_state= 0)
         classifier.fit(x_train,y_train)

Out[10]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
             kernel='linear', max_iter=-1, probability=False, random_state=0,
             shrinking=True, tol=0.001, verbose=False)

In [11]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [15]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for SVM", accuracy_score(y_test,y_pred))

         [[17  7]
          [ 3 34]]
         Accuracy score for SVM 0.8360655737704918

In [16]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

          Accuracy score for SVM using 10-fold cross validation: 0.83 (+/- 0.18)
```

Random Forest

10. From sklearn.ensemble I used RandomeForestClassifier. Trained the RandomForestClassifier model on the training set.

```
Random Forest

In [17]: from sklearn.ensemble import RandomForestClassifier
         classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
         classifier.fit(x_train, y_train)

Out[17]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=10,
                                n_jobs=None, oob_score=False, random_state=0, verbose=0,
                                warm_start=False)

In [18]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [20]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))

         [[19  5]
          [ 7 30]]
         Accuracy score for Random Forest 0.8032786885245902

In [21]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

          Accuracy score for Random Forest using 10-fold cross validation: 0.80 (+/- 0.20)
```

11. After training the model, I predicted the Test set results using predict().


```
Random Forest

In [17]: from sklearn.ensemble import RandomForestClassifier
         classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
         classifier.fit(x_train, y_train)

Out[17]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=10,
                                n_jobs=None, oob_score=False, random_state=0, verbose=0,
                                warm_start=False)

In [18]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [20]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))

         [[19  5]
          [ 7 30]]
         Accuracy score for Random Forest 0.8032786885245902

In [21]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

          Accuracy score for Random Forest using 10-fold cross validation: 0.80 (+/- 0.20)
```

12. Then I Evaluated the Model Performance and got an accuracy of 80.32%

Random Forest

```
In [17]: from sklearn.ensemble import RandomForestClassifier
         classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
         classifier.fit(x_train, y_train)

Out[17]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=10,
                                n_jobs=None, oob_score=False, random_state=0, verbose=0,
                                warm_start=False)

In [18]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [20]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))

         [[19  5]
          [ 7 30]]
         Accuracy score for Random Forest 0.8032786885245902

In [21]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

         Accuracy score for Random Forest using 10-fold cross validation: 0.80 (+/- 0.20)
```

13. In this step, I performed 10-fold cross-validation, and the accuracy score as 0.80 with a standard deviation of +/- 0.20.

Random Forest

```
In [17]: from sklearn.ensemble import RandomForestClassifier
         classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
         classifier.fit(x_train, y_train)

Out[17]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=10,
                                n_jobs=None, oob_score=False, random_state=0, verbose=0,
                                warm_start=False)

In [18]: y_pred= classifier.predict(x_test)
         #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

In [20]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test,y_pred)
         print(cm)
         print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))

         [[19  5]
          [ 7 30]]

In [21]: #Cross Validation
         scores = cross_val_score(classifier, x_train, y_train, cv=10)
         print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

         Accuracy score for Random Forest using 10-fold cross validation: 0.80 (+/- 0.20)
```


KNN

14.This is the implementation of KNN accuracy score for KNN using 10-fold cross validation is 0.83 (+/- 0.15)

KNN

```
[ ]   # KNN Model
      from sklearn.neighbors import KNeighborsClassifier
      classifier = KNeighborsClassifier(n_neighbors = 2)
      classifier.fit(x_train, y_train)

[ ]   y_pred= classifier.predict(x_test)
      #print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

[ ]   from sklearn.metrics import confusion_matrix, accuracy_score
      cm = confusion_matrix(y_test,y_pred)
      print(cm)
      print("Accuracy score for KNN", accuracy_score(y_test,y_pred))

[[25 8]
[11 24]]
Accuracy score for KNN 0.7638479245629473

[ ]   #Cross Validation
      scores = cross_val_score(classifier, x_train, y_train, cv=10)
      print(" Accuracy score for KNN using 10-fold cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

Source Code: The code I implemented on the dataset I chose.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score

data= pd.read_csv("heart.csv")

data.head()

data.isnull().sum().values.sum()

x = data.iloc[:,:-1].values
y = data.iloc[:,-1].values
print(x.shape)
print(y.shape)

from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.20)
print(x_train.shape)
print(x_test.shape)

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)


SVM
from sklearn.svm import SVC
classifier= SVC(kernel='linear', random_state= 0)
classifier.fit(x_train,y_train)

y_pred=  classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for SVM", accuracy_score(y_test,y_pred))

#Cross Validation
```

```
scores = cross_val_score(classifier, x_train, y_train, cv=10)
print(" Accuracy score for SVM using 10-fold cross validation: %0.2f (+/- %0.2f)" %
(scores.mean(), scores.std() * 2))
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy',
random_state = 0)
classifier.fit(x_train, y_train)

y_pred=  classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for Random Forest", accuracy_score(y_test,y_pred))

#Cross Validation
scores = cross_val_score(classifier, x_train, y_train, cv=10)
print(" Accuracy score for Random Forest using 10-fold cross validation: %0.2f (+/-
%0.2f)" % (scores.mean(), scores.std() * 2))
```

```
# KNN Model
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 2)
classifier.fit(x_train, y_train)

y_pred= classifier.predict(x_test)
#print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test,y_pred)
print(cm)
print("Accuracy score for KNN", accuracy_score(y_test,y_pred))

#Cross Validation
scores = cross_val_score(classifier, x_train, y_train, cv=10)
print(" Accuracy score for KNN using 10-fold cross validation: %0.2f (+/- %0.2f)" %
(scores.mean(), scores.std() * 2))
```

Related Source Code: Some related source code I used in this project are posted below.

import pandas as pd

pd.read():
LINK

pandas.read_csv(*filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*)**[source]**

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_bufferstr, path object or file-like object

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any os.PathLike.

By file-like object, we refer to objects with a read() method, such as a file handler (e.g. via builtin open function) or StringIO.

sepstr, default ‘,’

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s builtin sniffer tool, csv.Sniffer. In addition, separators longer than 1 character and different from '\s+' will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: '\r\t'.

delimiterstr, default None

Alias for sep.

headerint, list of int, default ‘infer’

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None. Explicitly pass header=0 to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if skip_blank_lines=True, so header=0 denotes the first line of data rather than the first line of the file.

namesarray-like, optional

List of column names to use. If the file contains a header row, then you should explicitly pass header=0 to override the column names. Duplicates in this list are not allowed.

index_colint, str, sequence of int / str, or False, default None

Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: index_col=False can be used to force pandas to *not* use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

usecolslist-like or callable, optional

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). For example, a valid list-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz']. Element order is ignored, so usecols=[0, 1] is the same as [1, 0]. To instantiate a DataFrame from data with element order preserved use pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']] for columns in ['foo', 'bar'] order or pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be lambda x: x.upper() in ['AAA', 'BBB', 'DDD']. Using this parameter results in much faster parsing time and lower memory usage.

squeezebool, default False

If the parsed data only contains one column then return a Series.

prefixstr, optional

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, …

mangle_dupe_colsbool, default True

Duplicate columns will be specified as ‘X’, ‘X.1’, …’X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

dtypeType name or dict of column -> type, optional

Data type for data or columns. E.g. {‘a’: np.float64, ‘b’: np.int32, ‘c’: ‘Int64’} Use str or object together with suitable na_values settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine{‘c’, ‘python’}, optional

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

convertersdict, optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_valueslist, optional

Values to consider as True.

false_valueslist, optional

Values to consider as False.

skipinitialspacebool, default False

Skip spaces after delimiter.

skiprowslist-like, int or callable, optional

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be lambda x: x in [0, 2].

skipfooterint, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c').

nrowsint, optional

Number of rows of file to read. Useful for reading pieces of large files.

na_valuesscalar, str, list-like, or dict, optional

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_nabool, default True

Whether or not to include the default NaN values when parsing the data. Depending on whether na_values is passed in, the behavior is as follows:

- If keep_default_na is True, and na_values are specified, na_values is appended to the default NaN values used for parsing.
- If keep_default_na is True, and na_values are not specified, only the default NaN values are used for parsing.
- If keep_default_na is False, and na_values are specified, only the NaN values specified na_values are used for parsing.
- If keep_default_na is False, and na_values are not specified, no strings will be parsed as NaN.

Note that if na_filter is passed in as False, the keep_default_na and na_values parameters will be ignored.

na_filterbool, default True

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbosebool, default False

Indicate number of NA values placed in non-numeric columns.

skip_blank_linesbool, default True

If True, skip over blank lines rather than interpreting as NaN values.

parse_datesbool or list of int or names or list of lists or dict, default False

The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index cannot be represented as an array of datetimes, say because of an unparseable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv. To parse an index or column with a mixture of timezones, specify date_parser to be a partially-applied **pandas.to_datetime()** with utc=True. See Parsing a CSV with mixed timezones for more.

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_formatbool, default False

If True and parse_dates is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_colbool, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parserfunction, optional

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion. Pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirstbool, default False

DD/MM format dates, international and European format.

cache_datesbool, default True

If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

iteratorbool, default False

Return TextFileReader object for iteration or getting chunks with get_chunk().

chunksizeint, optional

Return TextFileReader object for iteration. See the IO Tools docs for more information on iterator and chunksize.

compression{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer' and filepath_or_buffer is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

thousandsstr, optional

Thousands separator.

decimalstr, default '.'

Character to recognize as decimal point (e.g. use ',' for European data).

lineterminatorstr (length 1), optional

Character to break file into lines. Only valid with C parser.

quotecharstr (length 1), optional

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quotingint or csv.QUOTE_* instance, default 0

Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequotebool, default True

When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements INSIDE a field as a single quotechar element.

escapecharstr (length 1), optional

One-character string used to escape other characters.

commentstr, optional

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as skip_blank_lines=True), fully commented lines are ignored by the parameter header but not by skiprows. For example, if comment='#', parsing #empty\na,b,c\n1,2,3 with header=0 will result in 'a,b,c' being treated as the header.

encodingstr, optional

Encoding to use for UTF when reading/writing (ex. 'utf-8'). List of Python standard encodings .

dialectstr or csv.Dialect, optional

If provided, this parameter will override values (default or not) for the following parameters: delimiter, doublequote, escapechar, skipinitialspace, quotechar, and quoting. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

error_bad_linesbool, default True

Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these "bad lines" will dropped from the DataFrame that is returned.

warn_bad_linesbool, default True

If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

delim_whitespacebool, default False

Specifies whether or not whitespace (e.g. ' ' or ' ') will be used as the sep. Equivalent to setting sep='\s+'. If this option is set to True, nothing should be passed in for the delimiter parameter.

low_memorybool, default True

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser).

memory_mapbool, default False

If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

float_precisionstr, optional

Specifies which converter the C engine should use for floating-point values. The options are None for the ordinary converter, high for the high-precision converter, and round_trip for the round-trip converter.

Returns

Dataframe or TextParser

A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

from sklearn.model_selection import train_test_split

train_test_split():
LINK

sklearn.model_selection.**train_test_split**(**arrays, **options*)[source]
Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and next(ShuffleSplit().split(X, y)) and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the User Guide.

Parameters
arrayssequence of indexables with same length / shape[0]***
Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size*float or int, default=None***
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If train_size is also None, it will be set to 0.25.

train_size*float or int, default=None***
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state*int or RandomState instance, default=None***
Controls the shuffling applied to the data before applying the split. Pass an int for reproducible output across multiple function calls. See Glossary.

shuffle*bool, default=True***
Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify*array-like, default=None***
If not None, data is split in a stratified fashion, using this as the class labels.

Returns
splitting*list, length=2 * len(arrays)***
List containing train-test split of inputs.

New in version 0.16: If the input is sparse, the output will be a scipy.sparse.csr_matrix. Else, output type is the same as the input type.

Examples

```
>>>
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
```

```
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]

>>>
>>> X_train, X_test, y_train, y_test = train_test_split(
...  X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]

>>>
>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

from sklearn.preprocessing import StandartScalar

StandardScalar():
LINK

class sklearn.preprocessing.**StandardScaler**(*, *copy=True*, *with_mean=True*, *with_std=True*)[source]
Standardize features by removing the mean and scaling to unit variance

The standard score of a sample `x` is calculated as:

z = (x - u) / s

where `u` is the mean of the training samples or zero if `with_mean=False`, and `s` is the standard deviation of the training samples or one if `with_std=False`.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using **transform**.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features

correctly as expected.

This scaler can also be applied to sparse CSR or CSC matrices by passing with_mean=False to avoid breaking the sparsity structure of the data.

Read more in the User Guide.

Parameters
copy*boolean, optional, default True*
If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

with_mean*boolean, True by default*
If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_std*boolean, True by default*
If True, scale the data to unit variance (or equivalently, unit standard deviation).

Attributes
scale_*ndarray or None, shape (n_features,)*
Per feature relative scaling of the data. This is calculated using np.sqrt(var_). Equal to None when with_std=False.

*New in version 0.17: scale_*

mean_*ndarray or None, shape (n_features,)*
The mean value for each feature in the training set. Equal to None when with_mean=False.

var_*ndarray or None, shape (n_features,)*
The variance for each feature in the training set. Used to compute scale_. Equal to None when with_std=False.

n_samples_seen_*int or array, shape (n_features,)*
The number of samples processed by the estimator for each feature. If there are not missing samples, the n_samples_seen will be an integer, otherwise it will be an array. Will be reset on new calls to fit, but increments across partial_fit calls.

From sklearn.svm import SVC

SVC():
LINK

class sklearn.svm.**SVC**(*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)[source]
C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using **sklearn.svm.LinearSVC** or **sklearn.linear_model.SGDClassifier** instead, possibly after a

sklearn.kernel_approximation.Nystroem transformer.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how gamma, coef0 and degree affect each other, see the corresponding section in the narrative documentation: Kernel functions.

Read more in the User Guide.

Parameters

C*float, default=1.0*
Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel*{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*
Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape (n_samples, n_samples).

degree*int, default=3*
Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma*{'scale', 'auto'} or float, default='scale'*
Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of gamma changed from 'auto' to 'scale'.

coef0*float, default=0.0*
Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking*bool, default=True*
Whether to use the shrinking heuristic. See the User Guide.

probability*bool, default=False*
Whether to enable probability estimates. This must be enabled prior to calling fit, will slow down that method as it internally uses 5-fold cross-validation, and predict_proba may be inconsistent with predict. Read more in the User Guide.

tol*float, default=1e-3*
Tolerance for stopping criterion.

cache_size*float, default=200*
Specify the size of the kernel cache (in MB).

class_weight*dict or 'balanced', default=None*
Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

verbose*bool, default=False*
Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter*int, default=-1*
Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape*{'ovo', 'ovr'}, default='ovr'*
Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

Changed in version 0.19: decision_function_shape is 'ovr' by default.
New in version 0.17: decision_function_shape='ovr' is recommended.

Changed in version 0.17: Deprecated *decision_function_shape='ovo' and None*.
break_ties*bool, default=False*
If true, decision_function_shape='ovr', and number of classes > 2, predict will break ties according to the confidence values of decision_function; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

New in version 0.22.

random_state*int or RandomState instance, default=None*
Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when probability is False. Pass an int for reproducible output across multiple function calls. See Glossary.

Attributes
support_*ndarray of shape (n_SV,)*
Indices of support vectors.

support_vectors_*ndarray of shape (n_SV, n_features)*
Support vectors.

n_support_*ndarray of shape (n_class,), dtype=int32*
Number of support vectors for each class.

dual_coef_*ndarray of shape (n_class-1, n_SV)*
Dual coefficients of the support vector in the decision function (see Mathematical formulation), multiplied by their targets. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the multi-class section of the User Guide for details.

coef_*ndarray of shape (n_class * (n_class-1) / 2, n_features)*
Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

coef_ is a readonly property derived from dual_coef_ and support_vectors_.

intercept_*ndarray of shape (n_class * (n_class-1) / 2,)*
Constants in decision function.

fit_status_*int*
0 if correctly fitted, 1 otherwise (will raise warning)

classes_*ndarray of shape (n_classes,)*
The classes labels.

probA_*ndarray of shape (n_class * (n_class-1) / 2)*
probB_*ndarray of shape (n_class * (n_class-1) / 2)*
If probability=True, it corresponds to the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, it's an empty array. Platt scaling uses the logistic function 1 / (1 + exp(decision_value * probA_ + probB_)) where probA_ and probB_ are learned from the dataset [2]. For more information on the multiclass case and training procedure see section 8 of [1].

class_weight_*ndarray of shape (n_class,)*
Multipliers of parameter C for each class. Computed based on the class_weight parameter.

shape_fit_*tuple of int of shape (n_dimensions_of_X,)*
Array dimensions of training vector X.

from sklearn.metrics import confusion_matrix, accuracy_score

confusion_matrix():
LINK

sklearn.metrics.**confusion_matrix**(*y_true*, *y_pred*, ***, *labels=None*, *sample_weight=None*, *normalize=None*)[source]
Compute confusion matrix to evaluate the accuracy of a classification.

By definition a confusion matrix

C

is such that

Ci,j

is equal to the number of observations known to be in group

i

and predicted to be in group

j

.

Thus in binary classification, the count of true negatives is

C0,0

, false negatives is

C1,0

, true positives is

C1,1

and false positives is

C0,1

.

Read more in the User Guide.

Parameters
y_true*array-like of shape (n_samples,)*
Ground truth (correct) target values.

y_pred*array-like of shape (n_samples,)*
Estimated targets as returned by a classifier.

labels*array-like of shape (n_classes), default=None*
List of labels to index the matrix. This may be used to reorder or select a subset of labels. If None is given, those that appear at least once in y_true or y_pred are used in sorted order.

sample_weight*array-like of shape (n_samples,), default=None*
Sample weights.

New in version 0.18.

normalize*{'true', 'pred', 'all'}, default=None*
Normalizes confusion matrix over the true (rows), predicted (columns) conditions or all the population. If None, confusion matrix will not be normalized.

Returns
C*ndarray of shape (n_classes, n_classes)*
Confusion matrix whose i-th row and j-th column entry indicates the number of samples with true label being i-th class and prediced label being j-th class.

accuracy_score():
LINK

sklearn.metrics.**accuracy_score**(*y_true, y_pred, *, normalize=True, sample_weight=None*)[source]
Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the User Guide.

Parameters
y_true*1d array-like, or label indicator array / sparse matrix*
Ground truth (correct) labels.

y_pred*1d array-like, or label indicator array / sparse matrix*
Predicted labels, as returned by a classifier.

normalize*bool, optional (default=True)*
If False, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.

sample_weight*array-like of shape (n_samples,), default=None*
Sample weights.

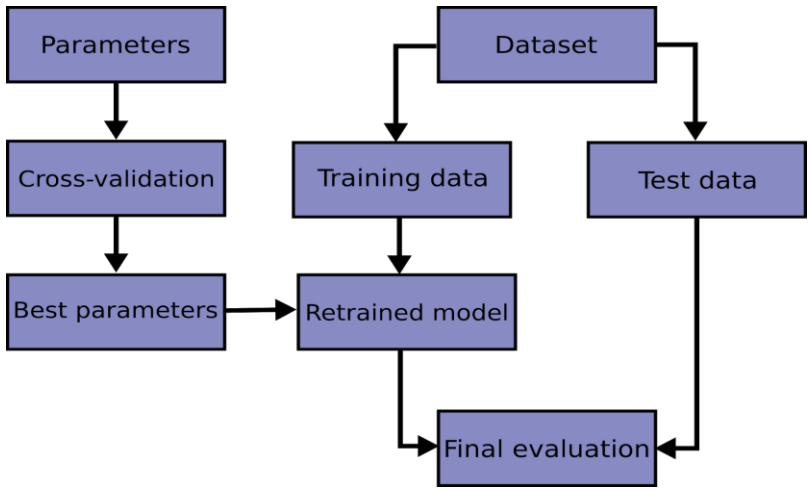
Returns
score*float*
If normalize == True, return the fraction of correctly classified samples (float), else returns the number of correctly classified samples (int).

The best performance is 1 with normalize == True and the number of samples with normalize == False.

from sklearn.model_selection import cross_val_score

cross_val_score():
LINK

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** X_test, y_test. Note that the word "experiment" is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally. Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques.



In scikit-learn a random split into training and test sets can be quickly computed with the **train_test_split** helper function. Let's load the iris data set to fit a linear support vector machine on it:

```
>>>
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
```

```
>>> from sklearn import svm

>>> X, y = datasets.load_iris(return_X_y=True)
>>> X.shape, y.shape
((150, 4), (150,))
```

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
>>>
>>> X_train, X_test, y_train, y_test = train_test_split(
...   X, y, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

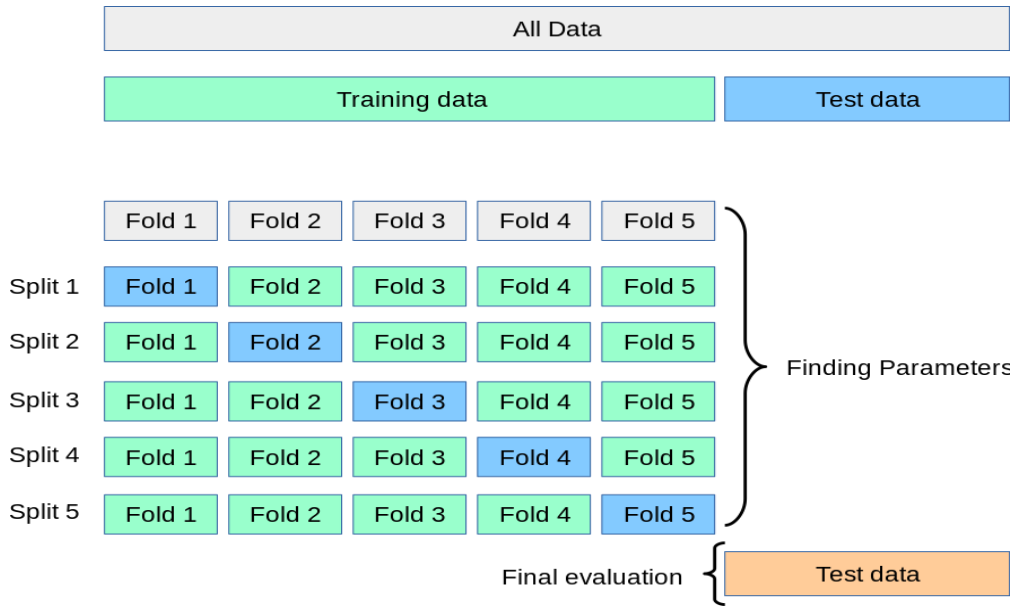
When evaluating different settings ("hyperparameters") for estimators, such as the C setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called *k*-fold CV, the training set is split into *k* smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the *k* "folds":

- A model is trained using
- k−1
- of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.



3.1.1. Computing cross-validated metrics

The simplest way to use cross-validation is to call the **cross_val_score** helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
>>>
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

The mean score and the 95% confidence interval of the score estimate are hence given by:

```
>>>
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

By default, the score computed at each CV iteration is the `score` method of the estimator. It is possible to change this by using the scoring parameter:

```
>>>
>>> from sklearn import metrics
>>> scores = cross_val_score(
...   clf, X, y, cv=5, scoring='f1_macro')
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

See The scoring parameter: defining model evaluation rules for details. In the case of the Iris dataset, the samples are balanced across target classes hence the accuracy and the F1-score are almost equal.

When the cv argument is an integer, **cross_val_score** uses the **KFold** or **StratifiedKFold** strategies by default, the latter being used if the estimator derives from **ClassifierMixin**.

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance:

```
>>>
>>> from sklearn.model_selection import ShuffleSplit
>>> n_samples = X.shape[0]
>>> cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
>>> cross_val_score(clf, X, y, cv=cv)
array([0.977..., 0.977..., 1.  ..., 0.955..., 1.        ])
```

Another option is to use an iterable yielding (train, test) splits as arrays of indices, for example:

```
>>>
>>> def custom_cv_2folds(X):
...   n = X.shape[0]
...   i = 1
...   while i <= 2:
...       idx = np.arange(n * (i - 1) / 2, n * i / 2, dtype=int)
...       yield idx, idx
...       i += 1
...
>>> custom_cv = custom_cv_2folds(X)
>>> cross_val_score(clf, X, y, cv=custom_cv)
array([1.        , 0.973...])
```

Data transformation with held out data

Just as it is important to test a predictor on data held-out from training, preprocessing (such as standardization, feature selection, etc.) and similar data transformations similarly should be learnt from a training set and applied to held-out data for prediction:

```
>>>
>>> from sklearn import preprocessing
>>> X_train, X_test, y_train, y_test = train_test_split(
...   X, y, test_size=0.4, random_state=0)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train_transformed = scaler.transform(X_train)
>>> clf = svm.SVC(C=1).fit(X_train_transformed, y_train)
>>> X_test_transformed = scaler.transform(X_test)
>>> clf.score(X_test_transformed, y_test)
0.9333...
```

A **Pipeline** makes it easier to compose estimators, providing this behavior under cross-validation:

```
>>>
>>> from sklearn.pipeline import make_pipeline
>>> clf = make_pipeline(preprocessing.StandardScaler(), svm.SVC(C=1))
>>> cross_val_score(clf, X, y, cv=cv)
array([0.977..., 0.933..., 0.955..., 0.933..., 0.977...])
```

See Pipelines and composite estimators.

3.1.1.1. The cross_validate function and multiple metric evaluation

The **cross_validate** function differs from **cross_val_score** in two ways:

- It allows specifying multiple metrics for evaluation.
- It returns a dict containing fit-times, score-times (and optionally training scores as well as fitted estimators) in addition to the test score.

For single metric evaluation, where the scoring parameter is a string, callable or None, the keys will be - ['test_score', 'fit_time', 'score_time']

And for multiple metric evaluation, the return value is a dict with the following keys - ['test_<scorer1_name>', 'test_<scorer2_name>', 'test_<scorer...>', 'fit_time', 'score_time']

return_train_score is set to False by default to save computation time. To evaluate the scores on the training set as well you need to be set to True.

You may also retain the estimator fitted on each training set by setting return_estimator=True.

The multiple metrics can be specified either as a list, tuple or set of predefined scorer names:

```
>>>
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro']
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, X, y, scoring=scoring)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro']
>>> scores['test_recall_macro']
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Or as a dict mapping scorer name to a predefined or custom scoring function:

```
>>>
>>> from sklearn.metrics import make_scorer
>>> scoring = {'prec_macro': 'precision_macro',
...            'rec_macro': make_scorer(recall_score, average='macro')}
>>> scores = cross_validate(clf, X, y, scoring=scoring,
...                         cv=5, return_train_score=True)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_prec_macro', 'test_rec_macro',
 'train_prec_macro', 'train_rec_macro']
>>> scores['train_rec_macro']
array([0.97..., 0.97..., 0.99..., 0.98..., 0.98...])
```

Here is an example of cross_validate using a single metric:

```
>>>
>>> scores = cross_validate(clf, X, y,
...                         scoring='precision_macro', cv=5,
...                         return_estimator=True)
>>> sorted(scores.keys())
['estimator', 'fit_time', 'score_time', 'test_score']
```

3.1.1.2. Obtaining predictions by cross-validation

The function **cross_val_predict** has a similar interface to **cross_val_score**, but returns, for each element in the input, the prediction that was obtained for that element when it was in the test set. Only cross-validation strategies that assign all elements to a test set exactly once can be used (otherwise, an exception is raised).

Warning Note on inappropriate usage of cross_val_predict
The result of **cross_val_predict** may be different from those obtained using **cross_val_score** as the elements are grouped in different ways. The function **cross_val_score** takes an average over cross-validation folds, whereas **cross_val_predict** simply returns the labels (or probabilities) from several distinct models undistinguished. Thus, **cross_val_predict** is not an appropriate measure of generalisation error.

The function cross_val_predict is appropriate for:

- Visualization of predictions obtained from different models.
- Model blending: When predictions of one supervised estimator are used to train another estimator in ensemble methods.

The available cross validation iterators are introduced in the following section.

Examples

- Receiver Operating Characteristic (ROC) with cross validation,
- Recursive feature elimination with cross-validation,
- Parameter estimation using grid search with cross-validation,
- Sample pipeline for text feature extraction and evaluation,
- Plotting Cross-Validated Predictions,
- Nested versus non-nested cross-validation.

3.1.2. Cross validation iterators

The following sections list utilities to generate indices that can be used to generate dataset splits according to different cross validation strategies.

3.1.2.1. Cross-validation iterators for i.i.d. data

Assuming that some data is Independent and Identically Distributed (i.i.d.) is making the assumption that all samples stem from the same generative process and that the generative process is assumed to have no memory of past generated samples.

The following cross-validators can be used in such cases.

NOTE

While i.i.d. data is a common assumption in machine learning theory, it rarely holds in practice. If one knows that the samples have been generated using a time-dependent process, it is safer to use a time-series aware cross-validation scheme. Similarly, if we know that the generative process has a group structure (samples collected from different subjects, experiments, measurement devices), it is safer to use group-wise cross-validation.

3.1.2.1.1. K-fold

KFold divides all the samples in

k

groups of samples, called folds (if

k=n

, this is equivalent to the *Leave One Out* strategy), of equal sizes (if possible). The prediction function is learned using

k−1

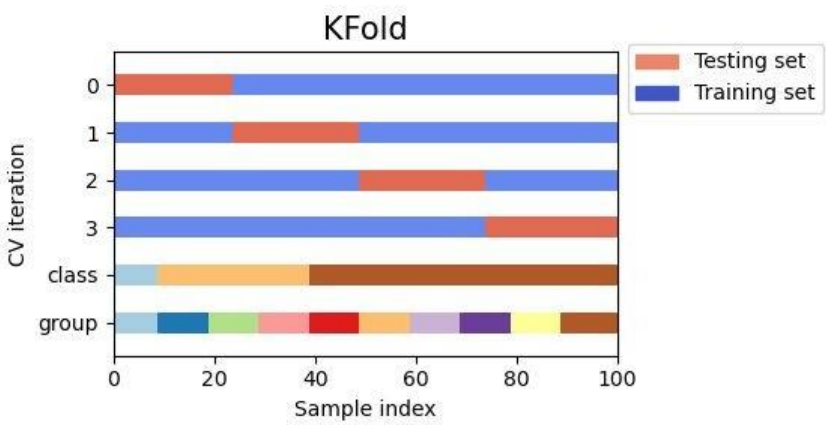
folds, and the fold left out is used for test.

Example of 2-fold cross-validation on a dataset with 4 samples:

```
>>>
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...  print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Here is a visualization of the cross-validation behavior. Note that **KFold** is not affected by classes or groups.



Each fold is constituted by two arrays: the first one is related to the *training set*, and the second one to the *test set*. Thus, one can create the training/test sets using numpy indexing:

```
>>>
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

3.1.2.1.2. Repeated K-Fold

RepeatedKFold repeats K-Fold n times. It can be used when one requires to run **KFold** n times, producing different splits in each repetition.

Example of 2-fold K-Fold repeated 2 times:

```
>>>
>>> import numpy as np
```

```
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...    print("%s %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

Similarly, **RepeatedStratifiedKFold** repeats Stratified K-Fold n times with different randomization in each repetition.

3.1.2.1.3. Leave One Out (LOO)

LeaveOneOut (or LOO) is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for

n

samples, we have

n

different training sets and

n

different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the training set:

```
>>>
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...  print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Potential users of LOO for model selection should weigh a few known caveats. When compared with

k

-fold cross validation, one builds

n

models from

n

samples instead of

k

models, where

$n>k$

. Moreover, each is trained on

$n-1$

samples rather than

$(k-1)n/k$

. In both ways, assuming

k

is not too large and

$k<n$

, LOO is more computationally expensive than

k

-fold cross validation.

In terms of accuracy, LOO often results in high variance as an estimator for the test error. Intuitively, since

$n-1$

of the

n

samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set.

However, if the learning curve is steep for the training size in question, then 5- or 10- fold cross validation can overestimate the generalization error.

As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

from sklearn.ensemble import RandomForestClassifier

RandomForestClassifier():
LINK

class sklearn.ensemble.**RandomForestClassifier**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*)[source]
A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

Read more in the User Guide.

Parameters
n_estimators*int, default=100*
The number of trees in the forest.

Changed in version 0.22: The default value of n_estimators changed from 10 to 100 in 0.22.
criterion*{"gini", "entropy"}, default="gini"*
The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth*int, default=None*
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split*int or float, default=2*
The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.
min_samples_leaf*int or float, default=1*
The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.
min_weight_fraction_leaf*float, default=0.0*
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features*{"auto", "sqrt", "log2"}, int or float, default="auto"*

The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features) (same as "auto").
- If "log2", then max_features=log2(n_features).
- If None, then max_features=n_features.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_leaf_nodes*int, default=None*
Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease*float, default=0.0*
A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

New in version 0.19.

min_impurity_split*float, default=None*
Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: min_impurity_split has been deprecated in favor of min_impurity_decrease in 0.19. The default value of min_impurity_split has changed from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use min_impurity_decrease instead.

bootstrap*bool, default=True*
Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score*bool, default=False*
Whether to use out-of-bag samples to estimate the generalization accuracy.

n_jobs*int, default=None*
The number of jobs to run in parallel. **fit**, **predict**, **decision_path** and **apply** are all parallelized over the trees. None means 1 unless in a **joblib.parallel_backend** context. -1 means using all processors. See Glossary for more details.

random_state*int or RandomState, default=None*
Controls both the randomness of the bootstrapping of the samples used when building trees (if bootstrap=True) and the sampling of the features to consider when looking for the best split at each node (if max_features < n_features). See Glossary for details.

verbose*int, default=0*
Controls the verbosity when fitting and predicting.

warm_start*bool, default=False*
When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See the Glossary.

class_weight*{"balanced", "balanced_subsample"}, dict or list of dicts, default=None*
Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha*non-negative float, default=0.0*
Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. See Minimal Cost-Complexity Pruning for details.

New in version 0.22.

max_samples*int or float, default=None*
If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw X.shape[0] samples.
- If int, then draw max_samples samples.
- If float, then draw max_samples * X.shape[0] samples. Thus, max_samples should be in the interval (0, 1).

New in version 0.22.

Attributes
base_estimator_*DecisionTreeClassifier*
The child estimator template used to create the collection of fitted sub-estimators.

estimators_*list of DecisionTreeClassifier*
The collection of fitted sub-estimators.

classes_*ndarray of shape (n_classes,) or a list of such arrays*
The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_*int or list*

The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

n_features_*int*
The number of features when fit is performed.

n_outputs_*int*
The number of outputs when fit is performed.

feature_importances_*ndarray of shape (n_features,)*
The impurity-based feature importances.

oob_score_*float*
Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when oob_score is True.

oob_decision_function_*ndarray of shape (n_samples, n_classes)*
Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, oob_decision_function_ might contain NaN. This attribute exists only when oob_score is True.

class sklearn.neighbors.**KNeighborsClassifier**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*)[source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters

n_neighbors*int, default=5*
Number of neighbors to use by default for **kneighbors** queries.

weights*{'uniform', 'distance'} or callable, default='uniform'*
weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm*{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*
Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use **BallTree**
- 'kd_tree' will use **KDTree**
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to **fit** method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size*int, default=30*
Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory

required to store the tree. The optimal value depends on the nature of the problem.

p*int, default=2*
Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric*str or callable, default='minkowski'*
the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of **DistanceMetric** for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

metric_params*dict, default=None*
Additional keyword arguments for the metric function.

n_jobs*int, default=None*
The number of parallel jobs to run for neighbors search. None means 1 unless in a **joblib.parallel_backend** context. -1 means using all processors. See Glossary for more details. Doesn't affect **fit** method.

Attributes

classes_*array of shape (n_classes,)*
Class labels known to the classifier

effective_metric_*str or callble*
The distance metric used. It will be same as the metric parameter or a synonym of it, e.g. 'euclidean' if the metric parameter set to 'minkowski' and p parameter set to 2.

effective_metric_params_*dict*
Additional keyword arguments for the metric function. For most metrics will be same with metric_params parameter, but may also contain the p parameter value if the effective_metric_ attribute is set to 'minkowski'.

n_samples_fit_*int*
Number of samples in the fitted data.

outputs_2d_*bool*
False when y's shape is (n_samples,) or (n_samples, 1) during fit otherwise True.

See also
RadiusNeighborsClassifier
KNeighborsRegressor
RadiusNeighborsRegressor
NearestNeighbors

Notes

See Nearest Neighbors in the online documentation for a discussion of the choice of algorithm and leaf_size.

Warning

Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor k+1 and k, have identical distances but different labels, the results will depend on the ordering of the training data.
https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>>
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
```

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

Methods

fit(X, y)	Fit the k-nearest neighbors classifier from the training dataset.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the class labels for the provided data.
predict_proba(X)	Return probability estimates for the test data X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

fit(*X*, *y*)[source]

Fit the k-nearest neighbors classifier from the training dataset.

Parameters

X{array-like, sparse matrix} of shape (n_samples, n_features) or (n_samples, n_samples) if metric='precomputed'

Training data.

y{array-like, sparse matrix} of shape (n_samples,) or (n_samples, n_outputs)

Target values.

Returns

self*KNeighborsClassifier*

The fitted k-nearest neighbors classifier.

get_params(*deep=True*)[source]

Get parameters for this estimator.

Parameters

deep*bool, default=True*

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params*dict*

Parameter names mapped to their values.

kneighbors(*X=None*, *n_neighbors=None*, *return_distance=True*)[source]

Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

Xarray-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None

The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors*int, default=None*
Number of neighbors required for each sample. The default is the value passed to the constructor.

return_distance*bool, default=True*
Whether or not to return the distances.

Returns
neigh_dist*ndarray of shape (n_queries, n_neighbors)*
Array representing the lengths to points, only present if return_distance=True

neigh_ind*ndarray of shape (n_queries, n_neighbors)*
Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NearestNeighbors class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors_graph(*X=None, n_neighbors=None, mode='connectivity'*)[source]
Computes the (weighted) graph of k-Neighbors for points in X

Parameters
X*array-like of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None*
The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor. For metric='precomputed' the shape should be (n_queries, n_indexed). Otherwise the shape should be (n_queries, n_features).

n_neighbors*int, default=None*
Number of neighbors for each sample. The default is the value passed to the constructor.

mode*{'connectivity', 'distance'}, default='connectivity'*
Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns
A*sparse-matrix of shape (n_queries, n_samples_fit)*
n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j. The matrix is of CSR format.

See also
NearestNeighbors.radius_neighbors_graph

Examples

```
>>>
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict(*X*)[source]

Predict the class labels for the provided data.

Parameters

X*array-like of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'*

Test samples.

Returns

y*ndarray of shape (n_queries,) or (n_queries, n_outputs)*

Class labels for each data sample.

predict_proba(*X*)[source]

Return probability estimates for the test data X.

Parameters

X*array-like of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'*

Test samples.

Returns

p*ndarray of shape (n_queries, n_classes), or a list of n_outputs*

of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

score(*X*, *y*, *sample_weight=None*)[source]

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X*array-like of shape (n_samples, n_features)*

Test samples.

y*array-like of shape (n_samples,) or (n_samples, n_outputs)*

True labels for X.

sample_weight*array-like of shape (n_samples,), default=None*

Sample weights.

Returns

score*float*

Mean accuracy of self.predict(X) wrt. y