K-nearest neigbhours(KNN)

In [1]:
```
1 import pandas as pd
```

In [2]:
```
1 data = pd.read_csv("https://raw.githubusercontent.com/AP-State-Skill-Develop
```

In [3]:
```
1 data
```

Out[3]:

	Height	Weight	Size
0	158	58	M
1	158	59	M
2	158	63	M
3	160	59	M
4	160	60	M
5	163	60	M
6	163	61	M
7	160	64	L
8	163	64	L
9	165	61	L
10	165	61	L
11	165	62	L
12	168	62	L
13	168	63	L
14	168	66	L
15	170	63	L
16	170	64	L
17	170	68	L

In [4]:
```
1 data.shape
```

Out[4]:
```
(18, 3)
```

In [5]:
```
1 data.columns
```

Out[5]:
```
Index(['Height', 'Weight', 'Size'], dtype='object')
```

In [6]:
```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 3 columns):
Height    18 non-null int64
Weight    18 non-null int64
Size      18 non-null object
dtypes: int64(2), object(1)
memory usage: 512.0+ bytes
```

In [7]:
```
1 data.isnull().sum()
```

Out[7]:
```
Height    0
Weight    0
Size      0
dtype: int64
```

In [8]:
```
1 data.isnull().sum().sum()
```

Out[8]:
```
0
```

In [9]:
```
1 data["Size"].value_counts()
```

Out[9]:
```
L    11
M     7
Name: Size, dtype: int64
```

In [10]:
```
1 # for this dataset we are not applying any preprocessing
```

In [11]:
```
1 # seperating features and targets
```

In [12]:
```
1 data.head()
```

Out[12]:

	Height	Weight	Size
0	158	58	M
1	158	59	M
2	158	63	M
3	160	59	M
4	160	60	M

In [13]:
```
inpu = data[["Height","Weight"]]
inpu
```

Out[13]:

	Height	Weight
0	158	58
1	158	59
2	158	63
3	160	59
4	160	60
5	163	60
6	163	61
7	160	64
8	163	64
9	165	61
10	165	61
11	165	62
12	168	62
13	168	63
14	168	66
15	170	63
16	170	64
17	170	68

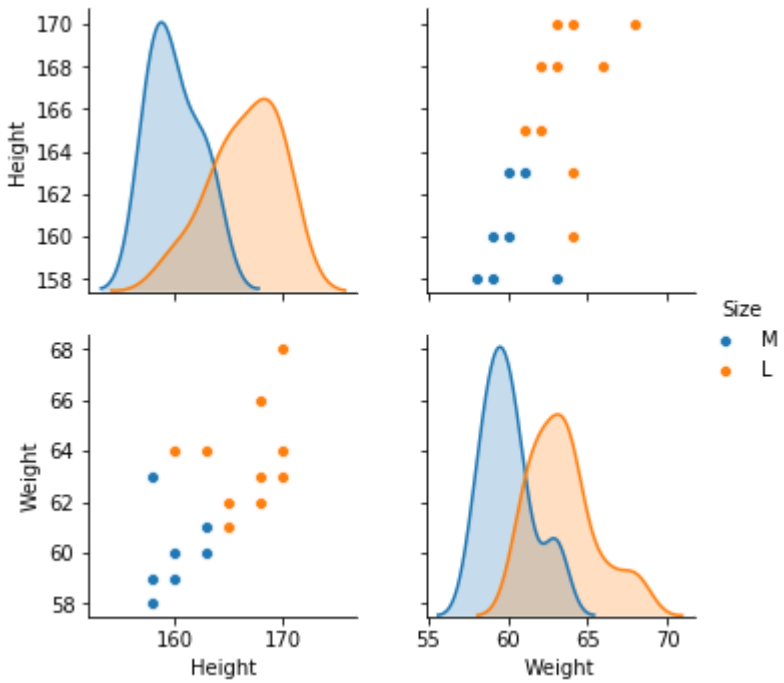
In [14]:
```
out = data["Size"]
out
```

Out[14]:
```
0     M
1     M
2     M
3     M
4     M
5     M
6     M
7     L
8     L
9     L
10    L
11    L
12    L
13    L
14    L
15    L
16    L
17    L
Name: Size, dtype: object
```

In [15]:
```
# visuvalize
import seaborn as sns
```

In [16]:
```
sns.pairplot(data,hue = "Size")
```

Out[16]:
```
<seaborn.axisgrid.PairGrid at 0x1c1eec3c978>
```



In [17]:
```
#get_dummies
#label Encoder
# one hot encoder
```

In [18]:
```
pd.get_dummies(out)
```

Out[18]:

	L	M
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1
5	0	1
6	0	1
7	1	0
8	1	0
9	1	0
10	1	0
11	1	0
12	1	0
13	1	0
14	1	0
15	1	0
16	1	0
17	1	0

In [19]:
```
from sklearn.preprocessing import LabelEncoder
```

In [20]:
```
label = LabelEncoder()
d = label.fit_transform(out)
d
```

Out[20]:
```
array([1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [25]:
```
data["Size"]=d
```

In [27]:
```
out = data["Size"]
```

In [28]:
```
# import algorithm
```

In [31]:
```
1 from sklearn.neighbors import KNeighborsClassifier
```

In [32]:
```
1 help(KNeighborsClassifier)
```

```
Help on class KNeighborsClassifier in module sklearn.neighbors.classificatio
n:

class KNeighborsClassifier(sklearn.neighbors.base.NeighborsBase, sklearn.neig
hbors.base.KNeighborsMixin, sklearn.neighbors.base.SupervisedIntegerMixin, sk
learn.base.ClassifierMixin)
 |  KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto',
leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwa
rgs)
 |
 |  Classifier implementing the k-nearest neighbors vote.
 |
 |  Read more in the :ref:`User Guide <classification>`.
 |
 |  Parameters
 |  ----------
 |  n_neighbors : int, optional (default = 5)
 |      Number of neighbors to use by default for :meth:`kneighbors` queries.
 |
 |  weights : str or callable, optional (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'}, optional
 |      Algorithm used to compute the nearest neighbors:
 |
 |      - 'ball_tree' will use :class:`BallTree`
 |      - 'kd_tree' will use :class:`KDTree`
 |      - 'brute' will use a brute-force search.
 |      - 'auto' will attempt to decide the most appropriate algorithm
 |        based on the values passed to :meth:`fit` method.
 |
 |      Note: fitting on sparse input will override the setting of
 |      this parameter, using brute force.
 |
 |  leaf_size : int, optional (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 : integer, optional (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 : string 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 the DistanceMetric class for a
|      list of available metrics.
|
|  metric_params : dict, optional (default = None)
|      Additional keyword arguments for the metric function.
|
|  n_jobs : int or None, optional (default=None)
|      The number of parallel jobs to run for neighbors search.
|      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
|      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
|      for more details.
|      Doesn't affect :meth:`fit` method.
|
|  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) # doctest: +ELLIPSIS
|  KNeighborsClassifier(...)
|  >>> print(neigh.predict([[1.1]]))
|  [0]
|  >>> print(neigh.predict_proba([[0.9]]))
|  [[0.66666667 0.33333333]]
|
|  See also
|  --------
|  RadiusNeighborsClassifier
|  KNeighborsRegressor
|  RadiusNeighborsRegressor
|  NearestNeighbors
|
|  Notes
|  -----
|  See :ref:`Nearest Neighbors <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 (https://en.wi
kipedia.org/wiki/K-nearest_neighbor_algorithm)
|
|  Method resolution order:
|      KNeighborsClassifier
|      sklearn.neighbors.base.NeighborsBase
|      abc.NewBase
|      sklearn.base.BaseEstimator
|      sklearn.neighbors.base.KNeighborsMixin
```

```
 |      sklearn.neighbors.base.SupervisedIntegerMixin
 |      sklearn.base.ClassifierMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_neighbors=5, weights='uniform', algorithm='auto', leaf_s
ize=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  predict(self, X)
 |      Predict the class labels for the provided data
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_query, n_features),                 or (n_qu
ery, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      y : array of shape [n_samples] or [n_samples, n_outputs]
 |          Class labels for each data sample.
 |
 |  predict_proba(self, X)
 |      Return probability estimates for the test data X.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_query, n_features),                 or (n_qu
ery, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      p : array of shape = [n_samples, 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.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
```

```
|
|      Parameters
|      ----------
|      deep : boolean, optional
|          If True, will return the parameters for this estimator and
|          contained subobjects that are estimators.
|
|      Returns
|      -------
|      params : mapping of string to any
|          Parameter names mapped to their values.
|
|  set_params(self, **params)
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
|      component of a nested object.
|
|      Returns
|      -------
|      self
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.BaseEstimator:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.neighbors.base.KNeighborsMixin:
|
|  kneighbors(self, X=None, n_neighbors=None, return_distance=True)
|      Finds the K-neighbors of a point.
|      Returns indices of and distances to the neighbors of each point.
|
|      Parameters
|      ----------
|      X : array-like, shape (n_query, n_features),                or (n_qu
ery, n_indexed) if metric == 'precomputed'
|          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
|          Number of neighbors to get (default is the value
|          passed to the constructor).
|
|      return_distance : boolean, optional. Defaults to True.
|          If False, distances will not be returned
|
|      Returns
|      -------
```

```
|      dist : array
|          Array representing the lengths to points, only present if
|          return_distance=True
|
|      ind : array
|          Indices of the nearest points in the population matrix.
|
|      Examples
|      --------
|      In the following example, we construct a NeighborsClassifier
|      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) # doctest: +ELLIPSIS
|      NearestNeighbors(algorithm='auto', leaf_size=30, ...)
|      >>> print(neigh.kneighbors([[1., 1., 1.]])) # doctest: +ELLIPSIS
|      (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) # doctest: +ELLIPSIS
|      array([[1],
|             [2]]...)
|
|  kneighbors_graph(self, X=None, n_neighbors=None, mode='connectivity')
|      Computes the (weighted) graph of k-Neighbors for points in X
|
|      Parameters
|      ----------
|      X : array-like, shape (n_query, n_features),                or (n_qu
ery, n_indexed) if metric == 'precomputed'
|          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
|          Number of neighbors for each sample.
|          (default is value passed to the constructor).
|
|      mode : {'connectivity', 'distance'}, optional
|          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 in CSR format, shape = [n_samples, 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.
|
|      Examples
```

```
|      --------
|      >>> X = [[0], [3], [1]]
|      >>> from sklearn.neighbors import NearestNeighbors
|      >>> neigh = NearestNeighbors(n_neighbors=2)
|      >>> neigh.fit(X) # doctest: +ELLIPSIS
|      NearestNeighbors(algorithm='auto', leaf_size=30, ...)
|      >>> A = neigh.kneighbors_graph(X)
|      >>> A.toarray()
|      array([[1., 0., 1.],
|             [0., 1., 1.],
|             [1., 0., 1.]])
|
|      See also
|      --------
|      NearestNeighbors.radius_neighbors_graph
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.neighbors.base.SupervisedIntegerMixin:
|
|  fit(self, X, y)
|      Fit the model using X as training data and y as target values
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix, BallTree, KDTree}
|          Training data. If array or matrix, shape [n_samples, n_features],
|          or [n_samples, n_samples] if metric='precomputed'.
|
|      y : {array-like, sparse matrix}
|          Target values of shape = [n_samples] or [n_samples, n_outputs]
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      Returns 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, shape = (n_samples, n_features)
|          Test samples.
|
|      y : array-like, shape = (n_samples) or (n_samples, n_outputs)
|          True labels for X.
|
|      sample_weight : array-like, shape = [n_samples], optional
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          Mean accuracy of self.predict(X) wrt. y.
```

In [77]:
```
knn = KNeighborsClassifier(n_neighbors=5)
```

In [78]:
```
knn.fit(inpu,out)
```

Out[78]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

In [79]:
```
# predict the model
pred = knn.predict(inpu)
pred
```

Out[79]:
```
array([1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [80]:
```
from sklearn import metrics
```

In [81]:
```
print(dir(metrics))
```

```
['SCORERS', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__
loader__', '__name__', '__package__', '__path__', '__spec__', 'accuracy_score',
'adjusted_mutual_info_score', 'adjusted_rand_score', 'auc', 'average_precision_
score', 'balanced_accuracy_score', 'base', 'brier_score_loss', 'calinski_haraba
z_score', 'check_scoring', 'classification', 'classification_report', 'cluste
r', 'cohen_kappa_score', 'completeness_score', 'confusion_matrix', 'consensus_s
core', 'coverage_error', 'davies_bouldin_score', 'euclidean_distances', 'explai
ned_variance_score', 'f1_score', 'fbeta_score', 'fowlkes_mallows_score', 'get_s
corer', 'hamming_loss', 'hinge_loss', 'homogeneity_completeness_v_measure', 'ho
mogeneity_score', 'jaccard_similarity_score', 'label_ranking_average_precision_
score', 'label_ranking_loss', 'log_loss', 'make_scorer', 'matthews_corrcoef',
'mean_absolute_error', 'mean_squared_error', 'mean_squared_log_error', 'median_
absolute_error', 'mutual_info_score', 'normalized_mutual_info_score', 'pairwis
e', 'pairwise_distances', 'pairwise_distances_argmin', 'pairwise_distances_argm
in_min', 'pairwise_distances_chunked', 'pairwise_fast', 'pairwise_kernels', 'pr
ecision_recall_curve', 'precision_recall_fscore_support', 'precision_score', 'r
2_score', 'ranking', 'recall_score', 'regression', 'roc_auc_score', 'roc_curv
e', 'scorer', 'silhouette_samples', 'silhouette_score', 'v_measure_score', 'zer
o_one_loss']
```

In [82]:
```
from sklearn.metrics import accuracy_score,confusion_matrix
```

In [83]:
```
accuracy_score(out,pred)*100
```

Out[83]:
```
83.33333333333334
```

In [84]:
```
confusion_matrix(out,pred)
```

Out[84]:
```
array([[10,  1],
       [ 2,  5]], dtype=int64)
```

In [85]:
```
1 data["Size"].value_counts()
```

Out[85]:
```
0    11
1     7
Name: Size, dtype: int64
```

Multi Class Classification

In [93]:
```
data1 = pd.read_excel("winequality-red.xls")
data1
```

Out[93]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alc
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	
5	11.6	0.580	0.66	2.2	0.074	10.0	47.0	1.0008	3.25	0.57	
6	7.4	0.660	0.00	1.8	0.075	13.0	40.0	0.9978	3.51	0.56	
7	7.9	0.600	0.06	1.6	0.069	15.0	59.0	0.9964	3.30	0.46	
8	7.3	0.650	0.00	1.2	0.065	15.0	21.0	0.9946	3.39	0.47	
9	7.8	0.580	0.02	2.0	0.073	9.0	18.0	0.9968	3.36	0.57	
10	7.5	0.500	0.36	6.1	0.071	17.0	102.0	0.9978	3.35	0.80	
11	6.7	0.580	0.08	1.8	0.097	15.0	65.0	0.9959	3.28	0.54	
12	7.5	0.500	0.36	6.1	0.071	17.0	102.0	0.9978	3.35	0.80	
13	5.6	0.615	0.00	1.6	0.089	16.0	59.0	0.9943	3.58	0.52	
14	11.6	0.580	0.66	2.2	0.074	10.0	47.0	1.0008	3.25	0.57	
15	7.8	0.610	0.29	1.6	0.114	9.0	29.0	0.9974	3.26	1.56	
16	8.9	0.620	0.18	3.8	0.176	52.0	145.0	0.9986	3.16	0.88	
17	7.3	0.650	0.00	1.2	0.065	15.0	21.0	0.9946	3.39	0.47	
18	8.9	0.620	0.19	3.9	0.170	51.0	148.0	0.9986	3.17	0.93	
19	8.5	0.280	0.56	1.8	0.092	35.0	103.0	0.9969	3.30	0.75	
20	8.1	0.560	0.28	1.7	0.368	16.0	56.0	0.9968	3.11	1.28	
21	7.4	0.590	0.08	4.4	0.086	6.0	29.0	0.9974	3.38	0.50	
22	7.9	0.320	0.51	1.8	0.341	17.0	56.0	0.9969	3.04	1.08	
23	11.6	0.580	0.66	2.2	0.074	10.0	47.0	1.0008	3.25	0.57	
24	8.9	0.220	0.48	1.8	0.077	29.0	60.0	0.9968	3.39	0.53	
25	7.6	0.390	0.31	2.3	0.082	23.0	71.0	0.9982	3.52	0.65	
26	7.9	0.430	0.21	1.6	0.106	10.0	37.0	0.9966	3.17	0.91	
27	8.5	0.490	0.11	2.3	0.084	9.0	67.0	0.9968	3.17	0.53	
28	6.9	0.400	0.14	2.4	0.085	21.0	40.0	0.9968	3.43	0.63	
29	6.3	0.390	0.16	1.4	0.080	11.0	23.0	0.9955	3.34	0.56	
...	...	...	...	...	...	...	...	...	...	...	
1714	6.9	0.400	0.14	2.4	0.085	21.0	40.0	0.9968	3.43	0.63	

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alc
1715	6.3	0.390	0.16	1.4	0.080	11.0	23.0	0.9955	3.34	0.56	
1716	7.6	0.410	0.24	1.8	0.080	4.0	11.0	0.9962	3.28	0.59	
1717	7.9	0.430	0.21	1.6	0.106	10.0	37.0	0.9966	3.17	0.91	
1718	7.1	0.710	0.00	1.9	0.080	14.0	35.0	0.9972	3.47	0.55	
1719	7.8	0.645	0.00	2.0	0.082	8.0	16.0	0.9964	3.38	0.59	
1720	6.7	0.675	0.07	2.4	0.089	17.0	82.0	0.9958	3.35	0.54	
1721	6.9	0.685	0.00	2.5	0.105	22.0	37.0	0.9966	3.46	0.57	
1722	8.3	0.655	0.12	2.3	0.083	15.0	113.0	0.9966	3.17	0.66	
1723	5.2	0.320	0.25	1.8	0.103	13.0	50.0	0.9957	3.38	0.55	
1724	7.8	0.600	0.14	2.4	0.086	3.0	15.0	0.9975	3.42	0.60	
1725	8.1	0.380	0.28	2.1	0.066	13.0	30.0	0.9968	3.23	0.73	
1726	5.7	1.130	0.09	1.5	0.172	7.0	19.0	0.9940	3.50	0.48	
1727	7.3	0.450	0.36	5.9	0.074	12.0	87.0	0.9978	3.33	0.83	
1728	7.3	0.450	0.36	5.9	0.074	12.0	87.0	0.9978	3.33	0.83	
1729	8.8	0.610	0.30	2.8	0.088	17.0	46.0	0.9976	3.26	0.51	
1730	7.5	0.490	0.20	2.6	0.332	8.0	14.0	0.9968	3.21	0.90	
1731	8.1	0.660	0.22	2.2	0.069	9.0	23.0	0.9968	3.30	1.20	
1732	6.8	0.670	0.02	1.8	0.050	5.0	11.0	0.9962	3.48	0.52	
1733	4.6	0.520	0.15	2.1	0.054	8.0	65.0	0.9934	3.90	0.56	
1734	7.7	0.935	0.43	2.2	0.114	22.0	114.0	0.9970	3.25	0.73	
1735	8.7	0.290	0.52	1.6	0.113	12.0	37.0	0.9969	3.25	0.58	
1736	6.4	0.400	0.23	1.6	0.066	5.0	12.0	0.9958	3.34	0.56	
1737	5.6	0.310	0.37	1.4	0.074	12.0	96.0	0.9954	3.32	0.58	
1738	8.8	0.660	0.26	1.7	0.074	4.0	23.0	0.9971	3.15	0.74	
1739	6.6	0.520	0.04	2.2	0.069	8.0	15.0	0.9956	3.40	0.63	
1740	6.6	0.500	0.04	2.1	0.068	6.0	14.0	0.9955	3.39	0.64	
1741	8.6	0.380	0.36	3.0	0.081	30.0	119.0	0.9970	3.20	0.56	
1742	7.6	0.510	0.15	2.8	0.110	33.0	73.0	0.9955	3.17	0.63	
1743	7.7	0.620	0.04	3.8	0.084	25.0	45.0	0.9978	3.34	0.53	

1744 rows × 12 columns

In [88]:
```
1 data1.shape
```

Out[88]: (1744, 12)

In [ ]:
```
1
```

In [89]:
```
1 data1.columns
```

Out[89]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [91]:
```
1 data1["quality"].value_counts()
```

Out[91]:
```
5    716
6    647
7    224
4     90
8     46
3     21
Name: quality, dtype: int64
```

In [92]:
```
1 data1["alcohol"].value_counts()
```

Out[92]:
```
9.500000     146
9.400000     112
9.800000      89
10.000000     88
9.200000      81
10.900000     77
10.500000     73
9.300000      62
11.000000     59
9.600000      58
9.700000      57
9.900000      50
10.100000     48
10.200000     47
10.800000     43
10.400000     41
9.000000      38
11.200000     36
10.300000     34
11.400000     32
11.300000     32
11.500000     30
10.600000     29
11.800000     29
10.700000     27
11.100000     27
12.800000     25
9.100000      24
11.700000     23
12.000000     21
              ...
12.400000     13
12.100000     13
12.300000     12
12.200000     12
12.700000     10
12.600000      6
8.400000       6
13.000000      6
13.600000      4
13.300000      3
13.100000      3
13.400000      3
9.550000       2
10.550000      2
10.033333      2
8.700000       2
8.800000       2
9.050000       1
9.950000       1
9.233333       1
11.950000      1
13.500000      1
8.500000       1
```

```
9.250000       1
10.750000      1
11.066667      1
13.200000      1
13.566667      1
14.900000      1
9.566667       1
Name: alcohol, Length: 65, dtype: int64
```

In [94]:
```
1 data1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1744 entries, 0 to 1743
Data columns (total 12 columns):
fixed acidity           1744 non-null float64
volatile acidity        1744 non-null float64
citric acid             1744 non-null float64
residual sugar          1744 non-null float64
chlorides               1744 non-null float64
free sulfur dioxide     1744 non-null float64
total sulfur dioxide    1744 non-null float64
density                 1744 non-null float64
pH                      1744 non-null float64
sulphates               1744 non-null float64
alcohol                 1744 non-null float64
quality                 1744 non-null int64
dtypes: float64(11), int64(1)
memory usage: 163.6 KB
```

In [95]:
```
1 data1.isnull().sum()
```

Out[95]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

In [96]:
```
1 featu = data1.drop(["quality"],axis=1)
```

In [97]:
```
1 featu.head()
```

Out[97]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

In [98]:
```
1 data1.head()
```

Out[98]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

In [99]:
```
1 out1 = data1["quality"]
2 out1.head()
```

Out[99]:
```
0    5
1    5
2    5
3    6
4    5
Name: quality, dtype: int64
```

In [100]:
```
1 ou = label.fit_transform(out1)
2 ou
```

Out[100]:
```
array([2, 2, 2, ..., 2, 3, 2], dtype=int64)
```

In [102]:
```
1 data1["quality"] = ou
```

In [103]:
```
1 out1 = data1["quality"]
```

In [104]:

```
# splitting the data for training and testing
from sklearn.model_selection import train_test_split
```

In [105]:
```
1 help(train_test_split)
```

```
Help on function train_test_split in module sklearn.model_selection._split:

train_test_split(*arrays, **options)
    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 :ref:`User Guide <cross_validation>`.

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

    test_size : float, int or None, optional (default=0.25)
        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. By default, the value is set to 0.25.
        The default will change in version 0.21. It will remain 0.25 only
        if ``train_size`` is unspecified, otherwise it will complement
        the specified ``train_size``.

    train_size : float, int, or None, (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, RandomState instance or None, optional (default=None)
        If int, random_state is the seed used by the random number generator;
        If RandomState instance, random_state is the random number generator;
        If None, the random number generator is the RandomState instance used
        by `np.random`.

    shuffle : boolean, optional (default=True)
        Whether or not to shuffle the data before splitting. If shuffle=False
        then stratify must be None.

    stratify : array-like or None (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.

        .. versionadded:: 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]]
```

In [145]:
```
x_train,x_test,y_train,y_test = train_test_split(featu,out1,test_size=0.3,ra
```

In [146]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [147]:
```
knn1 = KNeighborsClassifier(n_neighbors=4)
```

In [148]:
```
knn1.fit(x_train,y_train)
```

Out[148]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=4, p=2,
           weights='uniform')
```

In [149]:
```
pred1 = knn1.predict(x_train)
pred1
```

Out[149]:
```
array([3, 3, 3, ..., 2, 1, 4], dtype=int64)
```

In [150]:
```
1 accuracy_score(y_train,pred1)
```

Out[150]:
```
0.7065573770491803
```

In [151]:
```
1 confusion_matrix(y_train,pred1)
```

Out[151]:
```
array([[ 13,   1,   0,   1,   1,   0],
       [  0,  34,  16,  11,   3,   0],
       [  1,   8, 421,  55,   7,   0],
       [  0,   8, 148, 293,  19,   1],
       [  0,   4,  16,  49,  78,   1],
       [  0,   0,   3,   3,   2,  23]], dtype=int64)
```

In [ ]:
```
1
```

In [ ]:
```
1
```