HACKATHON TASK

In [1]:
```
# import library
import pandas as pd
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [2]:
```
import scipy
```

In [3]:
```
# download dataset
data = pd.read_csv("Downloads/data_1.csv")
```

In [4]:
```
# show a few rows of data
data.head()
```

Out[4]:

	x1	x2	y
0	-119.366669	115.000000	1
1	-101.108045	97.777159	1
2	-130.278658	106.767654	1
3	-114.703415	101.195477	1
4	-119.366669	115.000000	1

In [5]:
```
data.shape
```

Out[5]:
```
(2227, 3)
```

In [6]:
```
data.value_counts().shape
```

Out[6]:
```
(2203,)
```

In [7]:
```
data.columns
```

Out[7]:
```
Index(['x1', 'x2', 'y'], dtype='object')
```

Handle Missing Values

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

Out[8]:
```
x1    5
x2    3
y     0
dtype: int64
```

In [9]:
```
# from above result, it is clear that output/label has not contain any NULL va
lue.
```

In [10]:
```
data[['x1','x2']].isnull().sum()
```

Out[10]:
```
x1    5
x2    3
dtype: int64
```

In [11]:
```
data['x1'].index
```

Out[11]:
```
RangeIndex(start=0, stop=2227, step=1)
```

In [12]:
```
# find rows with missing data
null_data_rows = data[data.isnull().any(axis=1)]
null_data_rows
```

Out[12]:

	x1	x2	y
36	NaN	116.138522	1
44	-99.627522	NaN	1
98	NaN	36.905402	1
268	NaN	-116.385719	1
1084	NaN	34.714328	0
1092	55.162258	NaN	0
1318	NaN	53.182191	0
1430	-74.126054	NaN	0

In [13]:
```
# it is clear, not any row has both the missing values ['x1' and 'x2']
```

It is convinient, that remove these 8 rows [inspite of filling it with 0 or max_value or min_value or Average _value]

In [14]:
```
data[data['x1'].isnull()]
```

Out[14]:

	x1	x2	y
36	NaN	116.138522	1
98	NaN	36.905402	1
268	NaN	-116.385719	1
1084	NaN	34.714328	0
1318	NaN	53.182191	0

In [15]:
```
# remove null rows
final_data = data.dropna(axis=0, how='any')
final_data.shape
```

Out[15]:
```
(2219, 3)
```

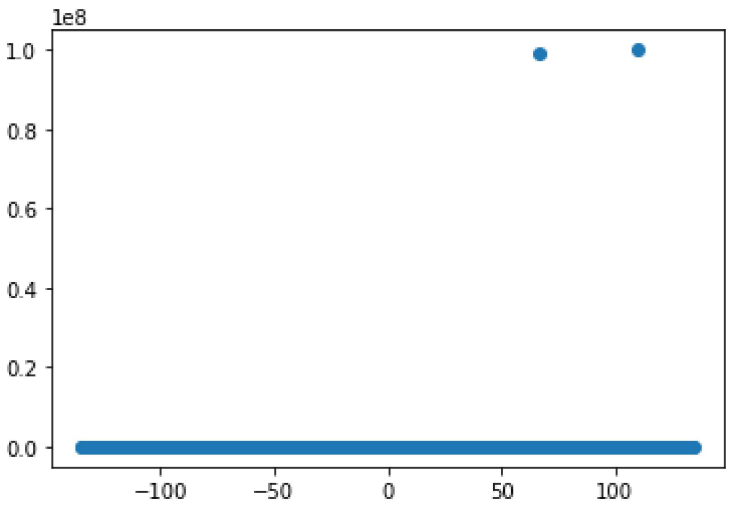
In [16]:
```
final_data.isnull().sum()
```

Out[16]:
```
x1    0
x2    0
y     0
dtype: int64
```

Final_data has no NULL value.

In [17]:
```
#Visualize final data
plt.scatter(final_data['x1'],final_data['x2'])
```

Out[17]:
```
<matplotlib.collections.PathCollection at 0x1402769e3a0>
```



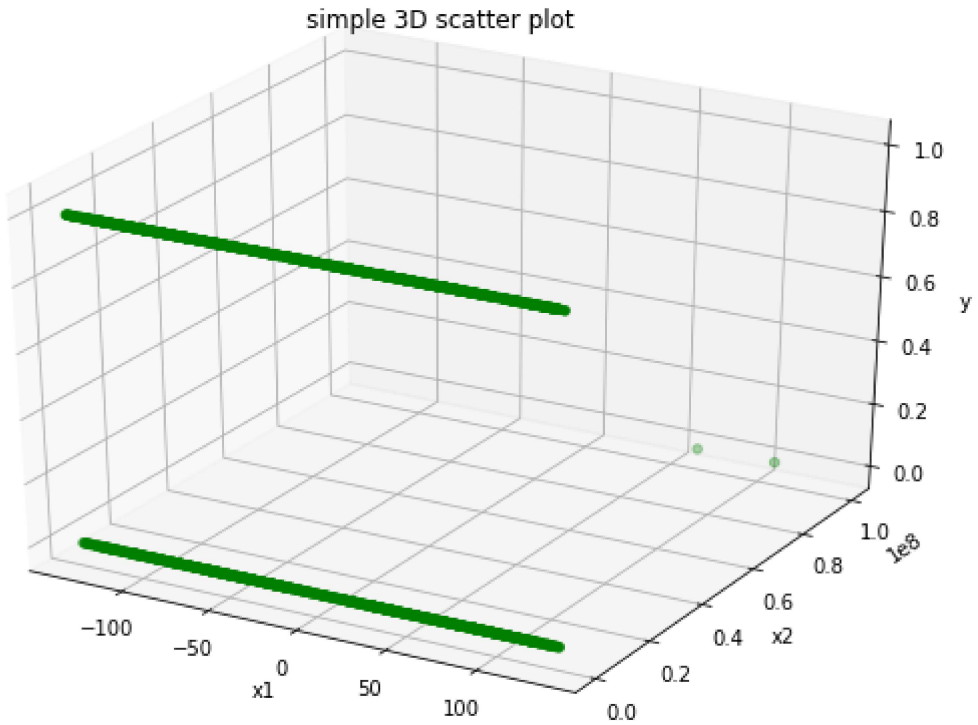
In [18]:
```
# Creating dataset
z = final_data['y']
x = final_data['x1']
y = final_data['x2']

# Creating figure
fig = plt.figure(figsize = (10, 7))
ax = plt.axes(projection ="3d")

# Creating plot
ax.scatter3D(x, y, z, color = "green")
plt.title("simple 3D scatter plot")

# set axis-label
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

# show plot
plt.show()
```



From above graph , it is clear that it has 2 outliers

In [19]:
```
final_data['x2'].max()
```

Out[19]: 99999999.0

In [20]:
```
final_data['x2'].min()
```

Out[20]: -134.8761321

In [21]:
```
final_data.describe()
```

Out[21]:

	x1	x2	y
count	2219.000000	2.219000e+03	2219.000000
mean	-4.771837	8.967782e+04	0.442091
std	74.938264	2.986680e+06	0.496747
min	-134.369160	-1.348761e+02	0.000000
25%	-71.581453	-8.025966e+01	0.000000
50%	-9.750840	-1.566092e+00	0.000000
75%	59.963056	6.390448e+01	1.000000
max	134.508687	1.000000e+08	1.000000

In [22]:
```
from scipy import stats
```

Ouliers detection and removal

keep only the ones that are within +3 to -3 standard deviations in the column 'Data'.

In [23]:
```
final_data =final_data[(np.abs(stats.zscore(final_data)) < 3).all(axis=1)]
final_data.shape
```

Out[23]: (2217, 3)

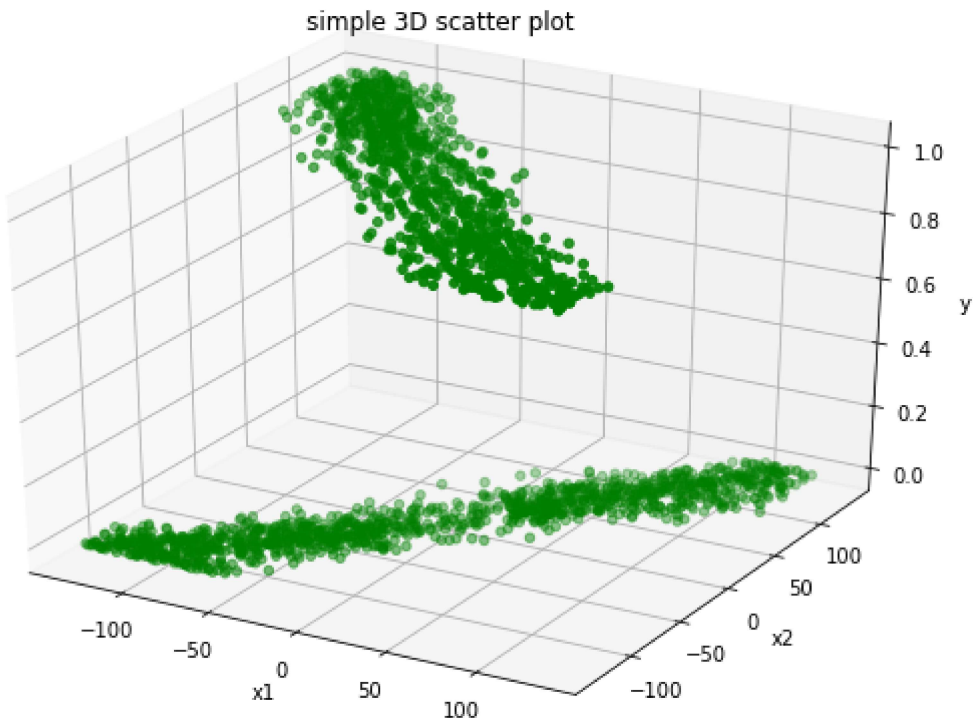
In [24]:
```
# Creating dataset
z = final_data['y']
x = final_data['x1']
y = final_data['x2']

# Creating figure
fig = plt.figure(figsize = (10, 7))
ax = plt.axes(projection ="3d")

# Creating plot
ax.scatter3D(x, y, z, color = "green")
plt.title("simple 3D scatter plot")

# set axis-label
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

# show plot
plt.show()
```



Now, this dataset has no outliers, as seen from above graph.

In [25]:
```
final_data.describe()
```

Out[25]:

	x1	x2	y
count	2217.000000	2217.000000	2217.000000
mean	-4.855645	-6.730568	0.442490
std	74.917277	79.843938	0.496794
min	-134.369160	-134.876132	0.000000
25%	-71.626564	-80.519315	0.000000
50%	-9.922271	-1.827336	0.000000
75%	59.633331	63.622498	1.000000
max	134.508687	134.929748	1.000000

In [26]:
```
final_data.dtypes
```

Out[26]:
```
x1    float64
x2    float64
y       int64
dtype: object
```

In [27]:
```
# see summary of dataset
final_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2217 entries, 0 to 2225
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   x1      2217 non-null   float64
 1   x2      2217 non-null   float64
 2   y       2217 non-null   int64  
dtypes: float64(2), int64(1)
memory usage: 69.3 KB
```

In [28]:
```
# Let's define X and y for our datatset as input_variable and output_variable.
```

In [29]:
```
# Converting pandas dataset into Numpy array
```

In [30]:
```
X = np.asarray(final_data[['x1','x2']])
X[0:5]
```

Out[30]:
```
array([[-119.3666687 ,  115.        ],
       [-101.1080445 ,   97.77715859],
       [-130.2786583 ,  106.767654  ],
       [-114.7034152 ,  101.1954767 ],
       [-119.3666687 ,  115.        ]])
```

In [31]:
```
y = np.asarray(final_data['y'])
y [0:5]
```

Out[31]:
```
array([1, 1, 1, 1, 1], dtype=int64)
```

also, normalize the dataset

In [32]:
```
from sklearn import preprocessing
X = preprocessing.StandardScaler().fit(X).transform(X)
X[0:5]
```

Out[32]:
```
array([[-1.52884438,  1.52495022],
       [-1.28507226,  1.30919525],
       [-1.67453109,  1.4218215 ],
       [-1.46658497,  1.3520174 ],
       [-1.52884438,  1.52495022]])
```

Train/Test dataset

Split our dataset into train and test set:

In [33]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, rand
om_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (1773, 2) (1773,)
Test set: (444, 2) (444,)
```

In [34]:
```
(444/(1773+443))
```

Out[34]:
```
0.2003610108303249
```

In [ ]:
```

```

Step - 2: Train different Machine Learning models:

a. Logistic Regression

In [35]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
```

In [36]:
```
LR = LogisticRegression().fit(X_train,y_train)
LR
```

Out[36]:
```
LogisticRegression()
```

In [37]:
```
help(LR)
```

```
Help on LogisticRegression in module sklearn.linear_model._logistic object:

class LogisticRegression(sklearn.base.BaseEstimator, sklearn.linear_model._ba
se.LinearClassifierMixin, sklearn.linear_model._base.SparseCoefMixin)
 |  LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_in
tercept=True, intercept_scaling=1, class_weight=None, random_state=None, solv
er='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_
jobs=None, l1_ratio=None)
 |
 |  Logistic Regression (aka logit, MaxEnt) classifier.
 |
 |  In the multiclass case, the training algorithm uses the one-vs-rest (OvR)
 |  scheme if the 'multi_class' option is set to 'ovr', and uses the
 |  cross-entropy loss if the 'multi_class' option is set to 'multinomial'.
 |  (Currently the 'multinomial' option is supported only by the 'lbfgs',
 |  'sag', 'saga' and 'newton-cg' solvers.)
 |
 |  This class implements regularized logistic regression using the
 |  'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **No
te
 |  that regularization is applied by default**. It can handle both dense
 |  and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit
 |  floats for optimal performance; any other input format will be converted
 |  (and copied).
 |
 |  The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularizatio
n
 |  with primal formulation, or no regularization. The 'liblinear' solver
 |  supports both L1 and L2 regularization, with a dual formulation only for
 |  the L2 penalty. The Elastic-Net regularization is only supported by the
 |  'saga' solver.
 |
 |  Read more in the :ref:`User Guide <logistic_regression>`.
 |
 |  Parameters
 |  ----------
 |  penalty : {'l1', 'l2', 'elasticnet', 'none'}, default='l2'
 |      Used to specify the norm used in the penalization. The 'newton-cg',
 |      'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is
 |      only supported by the 'saga' solver. If 'none' (not supported by the
 |      liblinear solver), no regularization is applied.
 |
 |      .. versionadded:: 0.19
 |         l1 penalty with SAGA solver (allowing 'multinomial' + L1)
 |
 |  dual : bool, default=False
 |      Dual or primal formulation. Dual formulation is only implemented for
 |      l2 penalty with liblinear solver. Prefer dual=False when
 |      n_samples > n_features.
 |
 |  tol : float, default=1e-4
 |      Tolerance for stopping criteria.
 |
 |  C : float, default=1.0
 |      Inverse of regularization strength; must be a positive float.
 |      Like in support vector machines, smaller values specify stronger
 |      regularization.
```

```
    |
    |  fit_intercept : bool, default=True
    |      Specifies if a constant (a.k.a. bias or intercept) should be
    |      added to the decision function.
    |
    |  intercept_scaling : float, default=1
    |      Useful only when the solver 'liblinear' is used
    |      and self.fit_intercept is set to True. In this case, x becomes
    |      [x, self.intercept_scaling],
    |      i.e. a "synthetic" feature with constant value equal to
    |      intercept_scaling is appended to the instance vector.
    |      The intercept becomes ``intercept_scaling * synthetic_feature_weight`
`.
    |
    |      Note! the synthetic feature weight is subject to l1/l2 regularization
    |      as all other features.
    |      To lessen the effect of regularization on synthetic feature weight
    |      (and therefore on the intercept) intercept_scaling has to be increase
d.
    |
    |  class_weight : dict or 'balanced', default=None
    |      Weights associated with classes in the form ``{class_label: weight}`
`.
    |      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))``.
    |
    |      Note that these weights will be multiplied with sample_weight (passed
    |      through the fit method) if sample_weight is specified.
    |
    |      .. versionadded:: 0.17
    |         *class_weight='balanced'*
    |
    |  random_state : int, RandomState instance, default=None
    |      Used when ``solver`` == 'sag', 'saga' or 'liblinear' to shuffle the
    |      data. See :term:`Glossary <random_state>` for details.
    |
    |  solver : {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'},
default='lbfgs'
    |
    |      Algorithm to use in the optimization problem.
    |
    |      - For small datasets, 'liblinear' is a good choice, whereas 'sag' and
    |        'saga' are faster for large ones.
    |      - For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfg
s'
    |        handle multinomial loss; 'liblinear' is limited to one-versus-rest
    |        schemes.
    |      - 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty
    |      - 'liblinear' and 'saga' also handle L1 penalty
    |      - 'saga' also supports 'elasticnet' penalty
    |      - 'liblinear' does not support setting ``penalty='none'``
    |
    |      Note that 'sag' and 'saga' fast convergence is only guaranteed on
    |      features with approximately the same scale. You can
```

```
    |      preprocess the data with a scaler from sklearn.preprocessing.
    |
    |      .. versionadded:: 0.17
    |         Stochastic Average Gradient descent solver.
    |      .. versionadded:: 0.19
    |         SAGA solver.
    |      .. versionchanged:: 0.22
    |          The default solver changed from 'liblinear' to 'lbfgs' in 0.22.
    |
    |  max_iter : int, default=100
    |      Maximum number of iterations taken for the solvers to converge.
    |
    |  multi_class : {'auto', 'ovr', 'multinomial'}, default='auto'
    |      If the option chosen is 'ovr', then a binary problem is fit for each
    |      label. For 'multinomial' the loss minimised is the multinomial loss f
it
    |      across the entire probability distribution, *even when the data is
    |      binary*. 'multinomial' is unavailable when solver='liblinear'.
    |      'auto' selects 'ovr' if the data is binary, or if solver='liblinear',
    |      and otherwise selects 'multinomial'.
    |
    |      .. versionadded:: 0.18
    |         Stochastic Average Gradient descent solver for 'multinomial' case.
    |      .. versionchanged:: 0.22
    |          Default changed from 'ovr' to 'auto' in 0.22.
    |
    |  verbose : int, default=0
    |      For the liblinear and lbfgs solvers set verbose to any positive
    |      number for verbosity.
    |
    |  warm_start : bool, default=False
    |      When set to True, reuse the solution of the previous call to fit as
    |      initialization, otherwise, just erase the previous solution.
    |      Useless for liblinear solver. See :term:`the Glossary <warm_start>`.
    |
    |      .. versionadded:: 0.17
    |         *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solver
s.
    |
    |  n_jobs : int, default=None
    |      Number of CPU cores used when parallelizing over classes if
    |      multi_class='ovr'". This parameter is ignored when the ``solver`` is
    |      set to 'liblinear' regardless of whether 'multi_class' is specified o
r
    |      not. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
    |      context. ``-1`` means using all processors.
    |      See :term:`Glossary <n_jobs>` for more details.
    |
    |  l1_ratio : float, default=None
    |      The Elastic-Net mixing parameter, with ``0 <= l1_ratio <= 1``. Only
    |      used if ``penalty='elasticnet'``. Setting ``l1_ratio=0`` is equivalen
t
    |      to using ``penalty='l2'``, while setting ``l1_ratio=1`` is equivalent
    |      to using ``penalty='l1'``. For ``0 < l1_ratio <1``, the penalty is a
    |      combination of L1 and L2.
    |
    |  Attributes
```

```
|  ----------
|
|  classes_ : ndarray of shape (n_classes, )
|      A list of class labels known to the classifier.
|
|  coef_ : ndarray of shape (1, n_features) or (n_classes, n_features)
|      Coefficient of the features in the decision function.
|
|      `coef_` is of shape (1, n_features) when the given problem is binary.
|      In particular, when `multi_class='multinomial'`, `coef_` corresponds
|      to outcome 1 (True) and `-coef_` corresponds to outcome 0 (False).
|
|  intercept_ : ndarray of shape (1,) or (n_classes,)
|      Intercept (a.k.a. bias) added to the decision function.
|
|      If `fit_intercept` is set to False, the intercept is set to zero.
|      `intercept_` is of shape (1,) when the given problem is binary.
|      In particular, when `multi_class='multinomial'`, `intercept_`
|      corresponds to outcome 1 (True) and `-intercept_` corresponds to
|      outcome 0 (False).
|
|  n_iter_ : ndarray of shape (n_classes,) or (1, )
|      Actual number of iterations for all classes. If binary or multinomia
l,
|      it returns only 1 element. For liblinear solver, only the maximum
|      number of iteration across all classes is given.
|
|      .. versionchanged:: 0.20
|
|          In SciPy <= 1.0.0 the number of lbfgs iterations may exceed
|          ``max_iter``. ``n_iter_`` will now report at most ``max_iter``.
|
|  See Also
|  --------
|  SGDClassifier : Incrementally trained logistic regression (when given
|      the parameter ``loss="log"``).
|  LogisticRegressionCV : Logistic regression with built-in cross validatio
n.
|
|  Notes
|  -----
|  The underlying C implementation uses a random number generator to
|  select features when fitting the model. It is thus not uncommon,
|  to have slightly different results for the same input data. If
|  that happens, try with a smaller tol parameter.
|
|  Predict output may not match that of standalone liblinear in certain
|  cases. See :ref:`differences from liblinear <liblinear_differences>`
|  in the narrative documentation.
|
|  References
|  ----------
|
|  L-BFGS-B -- Software for Large-scale Bound-constrained Optimization
|      Ciyou Zhu, Richard Byrd, Jorge Nocedal and Jose Luis Morales.
|      http://users.iems.northwestern.edu/~nocedal/lbfgsb.html
|
```

```
 |  LIBLINEAR -- A Library for Large Linear Classification
 |      https://www.csie.ntu.edu.tw/~cjlin/liblinear/
 |
 |  SAG -- Mark Schmidt, Nicolas Le Roux, and Francis Bach
 |      Minimizing Finite Sums with the Stochastic Average Gradient
 |      https://hal.inria.fr/hal-00860051/document
 |
 |  SAGA -- Defazio, A., Bach F. & Lacoste-Julien S. (2014).
 |      SAGA: A Fast Incremental Gradient Method With Support
 |      for Non-Strongly Convex Composite Objectives
 |      https://arxiv.org/abs/1407.0202
 |
 |  Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descen
t
 |      methods for logistic regression and maximum entropy models.
 |      Machine Learning 85(1-2):41-75.
 |      https://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf
 |
 |  Examples
 |  --------
 |  >>> from sklearn.datasets import load_iris
 |  >>> from sklearn.linear_model import LogisticRegression
 |  >>> X, y = load_iris(return_X_y=True)
 |  >>> clf = LogisticRegression(random_state=0).fit(X, y)
 |  >>> clf.predict(X[:2, :])
 |  array([0, 0])
 |  >>> clf.predict_proba(X[:2, :])
 |  array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
 |         [9.7...e-01, 2.8...e-02, ...e-08]])
 |  >>> clf.score(X, y)
 |  0.97...
 |
 |  Method resolution order:
 |      LogisticRegression
 |      sklearn.base.BaseEstimator
 |      sklearn.linear_model._base.LinearClassifierMixin
 |      sklearn.base.ClassifierMixin
 |      sklearn.linear_model._base.SparseCoefMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_interc
ept=True, intercept_scaling=1, class_weight=None, random_state=None, solver
='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jo
bs=None, l1_ratio=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit the model according to the given training data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          Training vector, where n_samples is the number of samples and
 |          n_features is the number of features.
 |
```

```
|      y : array-like of shape (n_samples,)
|          Target vector relative to X.
|
|      sample_weight : array-like of shape (n_samples,) default=None
|          Array of weights that are assigned to individual samples.
|          If not provided, then each sample is given unit weight.
|
|          .. versionadded:: 0.17
|             *sample_weight* support to LogisticRegression.
|
|      Returns
|      -------
|      self
|          Fitted estimator.
|
|      Notes
|      -----
|      The SAGA solver supports both float64 and float32 bit arrays.
|
|  predict_log_proba(self, X)
|      Predict logarithm of probability estimates.
|
|      The returned estimates for all classes are ordered by the
|      label of classes.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Vector to be scored, where `n_samples` is the number of samples a
nd
|          `n_features` is the number of features.
|
|      Returns
|      -------
|      T : array-like of shape (n_samples, n_classes)
|          Returns the log-probability of the sample for each class in the
|          model, where classes are ordered as they are in ``self.classes_`
`.
|
|  predict_proba(self, X)
|      Probability estimates.
|
|      The returned estimates for all classes are ordered by the
|      label of classes.
|
|      For a multi_class problem, if multi_class is set to be "multinomial"
|      the softmax function is used to find the predicted probability of
|      each class.
|      Else use a one-vs-rest approach, i.e calculate the probability
|      of each class assuming it to be positive using the logistic function.
|      and normalize these values across all the classes.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Vector to be scored, where `n_samples` is the number of samples a
nd
```

```
|          `n_features` is the number of features.
|
|      Returns
|      -------
|      T : array-like of shape (n_samples, n_classes)
|          Returns the probability of the sample for each class in the mode
l,
|          where classes are ordered as they are in ``self.classes_``.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      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 : 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.
|
|      Parameters
|      ----------
|      **params : dict
|          Estimator parameters.
|
|      Returns
|      -------
|      self : object
|          Estimator instance.
|
|  ----------------------------------------------------------------------
|  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.linear_model._base.LinearClassifierMixin:
|
|  decision_function(self, X)
|      Predict confidence scores for samples.
|
|      The confidence score for a sample is the signed distance of that
|      sample to the hyperplane.
|
|      Parameters
|      ----------
|      X : array_like or sparse matrix, shape (n_samples, n_features)
|          Samples.
|
|      Returns
|      -------
|      array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classe
s)
|          Confidence scores per (sample, class) combination. In the binary
|          case, confidence score for self.classes_[1] where >0 means this
|          class would be predicted.
|
|  predict(self, X)
|      Predict class labels for samples in X.
|
|      Parameters
|      ----------
|      X : array_like or sparse matrix, shape (n_samples, n_features)
|          Samples.
|
|      Returns
|      -------
|      C : array, shape [n_samples]
|          Predicted class label per sample.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      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.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.linear_model._base.SparseCoefMixin:
|
|  densify(self)
|      Convert coefficient matrix to dense array format.
|
|      Converts the ``coef_`` member (back) to a numpy.ndarray. This is the
|      default format of ``coef_`` and is required for fitting, so calling
|      this method is only required on models that have previously been
|      sparsified; otherwise, it is a no-op.
|
|      Returns
|      -------
|      self
|          Fitted estimator.
|
|  sparsify(self)
|      Convert coefficient matrix to sparse format.
|
|      Converts the ``coef_`` member to a scipy.sparse matrix, which for
|      L1-regularized models can be much more memory- and storage-efficient
|      than the usual numpy.ndarray representation.
|
|      The ``intercept_`` member is not converted.
|
|      Returns
|      -------
|      self
|          Fitted estimator.
|
|      Notes
|      -----
|      For non-sparse models, i.e. when there are not many zeros in ``coef_``
,
|      this may actually *increase* memory usage, so use this method with
|      care. A rule of thumb is that the number of zero elements, which can
|      be computed with ``(coef_ == 0).sum()``, must be more than 50% for th
is
|      to provide significant benefits.
|
|      After calling this method, further fitting with the partial_fit
|      method (if any) will not work until you call densify.
```

Note:

This class implements regularized logistic regression using the | 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note | that regularization is applied by default**. It can handle both dense | and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit | floats for optimal performance; any other input format will be converted | (and copied).

Now we can predict using our test set:

In [38]:
```
yhat = LR.predict(X_test)
yhat[:20]
```

Out[38]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
      dtype=int64)
```

predict_proba returns estimates for all classes, ordered by the label of classes. So, the first column is the probability of class 0, P(Y=0|X), and second column is probability of class 1, P(Y=1|X):

In [39]:
```
yhat_prob = LR.predict_proba(X_test)
yhat_prob[:5]
```

Out[39]:
```
array([[0.57589113, 0.42410887],
       [0.56695529, 0.43304471],
       [0.59984188, 0.40015812],
       [0.65646321, 0.34353679],
       [0.61968343, 0.38031657]])
```

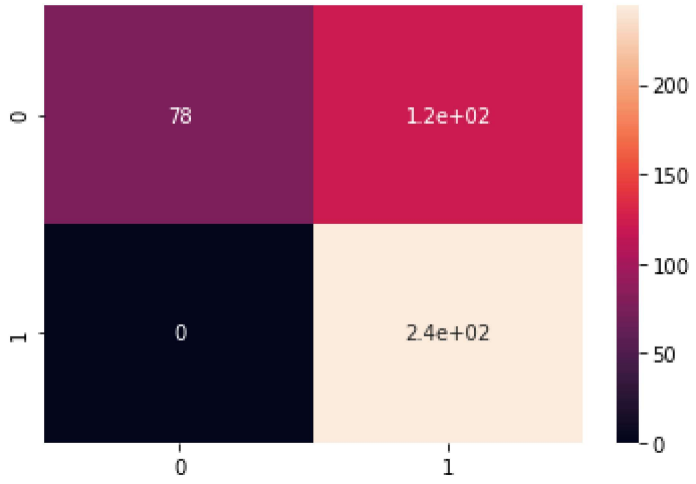
Accuracy Calculation

In [40]:
```
confusion_matrix(y_test, yhat, labels=[1,0])
```

Out[40]:
```
array([[ 78, 122],
       [  0, 244]], dtype=int64)
```

In [41]:
```
sns.heatmap(confusion_matrix(y_test, yhat, labels=[1,0]),annot=True)
```

Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x14027ada490>



In [42]:
```
from sklearn.metrics import accuracy_score, classification_report
```

In [43]:
```
print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       0.67      1.00      0.80       244
           1       1.00      0.39      0.56       200

    accuracy                           0.73       444
   macro avg       0.83      0.70      0.68       444
weighted avg       0.82      0.73      0.69       444
```

In [44]:
```
accuracy_score(y_test, yhat)
```

Out[44]: 0.7252252252252253

for hyperparrameter tunning, we can change the values inside LogisticRegression model

LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)

Accuracy of Logistic Regression Model : 72.5%

In [ ]:
```

```

b. SVM with Linear Kernel

In [45]:
```
from sklearn import svm
```

The SVM algorithm has a number of kernel functions for performing its processing.

Kernel : mapping data into a higher dimensional space is called kernelling

The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:

1.Linear 2.Polynomial 3.Radial basis function (RBF) 4.Sigmoid

Linear kernel

kernel : {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'

In [46]:
```
svm_lin = svm.SVC(kernel='linear')

#fitting data into model
svm_lin.fit(X_train, y_train)
```

Out[46]:
```
SVC(kernel='linear')
```

In [47]:
```
# model is used to predict value
yhat = svm_lin.predict(X_test)
yhat [0:12]
```

Out[47]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0], dtype=int64)
```

Accuracy Calculation

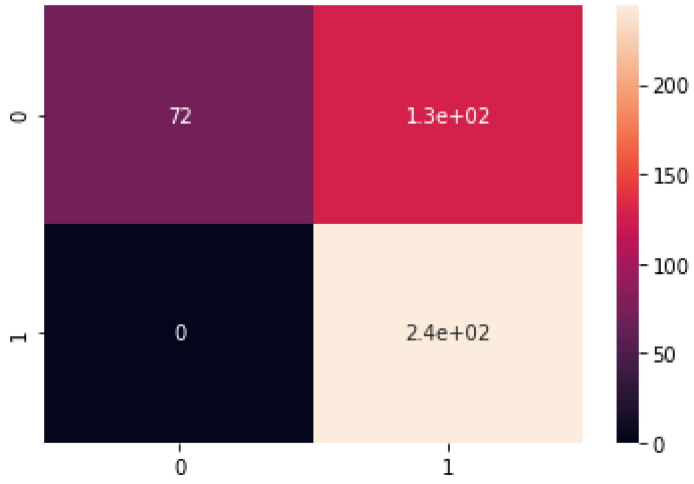
In [48]:
```
from sklearn.metrics import classification_report, confusion_matrix, f1_score,
accuracy_score,  jaccard_score
```

In [49]:
```
confusion_matrix(y_test, yhat, labels=[1,0])
```

Out[49]:
```
array([[ 72, 128],
       [  0, 244]], dtype=int64)
```

In [50]:
```
sns.heatmap(confusion_matrix(y_test, yhat, labels=[1,0]),annot=True)
```

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x14027bb4f10>



In [51]:
```
f1_score(y_test, yhat, average='weighted')
```

Out[51]: 0.673831203242968

In [52]:
```
jaccard_score(y_test, yhat,pos_label=0)
```

Out[52]: 0.6559139784946236

In [53]:
```
print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       0.66      1.00      0.79       244
           1       1.00      0.36      0.53       200

    accuracy                           0.71       444
   macro avg       0.83      0.68      0.66       444
weighted avg       0.81      0.71      0.67       444
```

In [54]:
```
accuracy_score(y_test, yhat)
```

Out[54]: 0.7117117117117117

SVM also has some hyper-parameters (like what C or gamma values to use)

Accuracy of SVM (Linear-Kernel) Model : 71.2%

In [ ]:
```

```

c. SVM with RBF Kernel

In [55]:
```
svm_rbf = svm.SVC(kernel='rbf')

#fitting data into model
svm_rbf.fit(X_train, y_train)
```

Out[55]:
```
SVC()
```

In [56]:
```
# model is used to predict value
yhat = svm_rbf.predict(X_test)
yhat [0:12]
```

Out[56]:
```
array([0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0], dtype=int64)
```

Accuracy Calculation

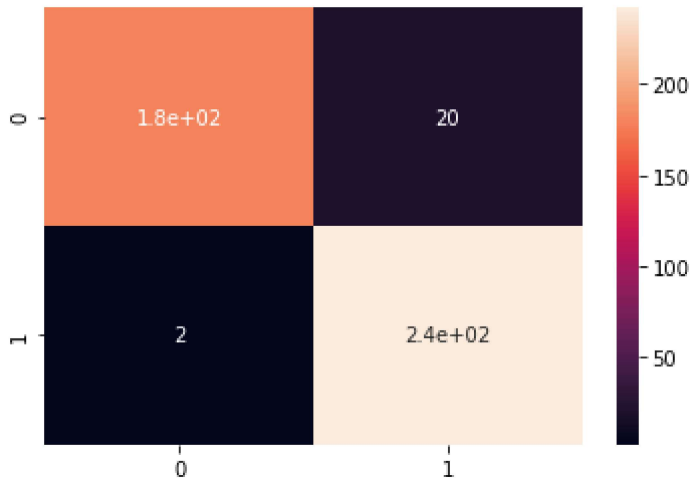
In [57]:
```
from sklearn.metrics import classification_report, confusion_matrix, f1_score,
accuracy_score,  jaccard_score
```

In [58]:
```
confusion_matrix(y_test, yhat, labels=[1,0])
```

Out[58]:
```
array([[180,  20],
       [  2, 242]], dtype=int64)
```

In [59]:
```
sns.heatmap(confusion_matrix(y_test, yhat, labels=[1,0]),annot=True)
```

Out[59]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x14028c14f10>
```



In [60]:
```
f1_score(y_test, yhat, average='weighted')
```

Out[60]:
```
0.9501643687849015
```

In [61]:
```
jaccard_score(y_test, yhat,pos_label=0)
```

Out[61]:
```
0.9166666666666666
```

In [62]:
```
print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       0.92      0.99      0.96       244
           1       0.99      0.90      0.94       200

    accuracy                           0.95       444
   macro avg       0.96      0.95      0.95       444
weighted avg       0.95      0.95      0.95       444
```

In [63]:
```
accuracy_score(y_test, yhat)
```

Out[63]:
```
0.9504504504504504
```

Accuracy of SVM (rbf-Kernel) Model : 95%

In [ ]:
```

```

Decision Tree Model

In [64]:
```
from sklearn.tree import DecisionTreeClassifier
```

In [65]:
```
# Creating tree model
tree_model = DecisionTreeClassifier()
tree_model # it shows the default parameters
```

Out[65]:
```
DecisionTreeClassifier()
```

In [66]:
```
# fitting tree model
tree_model.fit(X_train, y_train)
```

Out[66]:
```
DecisionTreeClassifier()
```

Prediction

In [67]:
```
y_pred = tree_model.predict(X_test)
```

In [68]:
```
y_pred[:12]
```

Out[68]:
```
array([0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0], dtype=int64)
```

Calculating Accuracy

In [69]:
```
accuracy_score(y_test, y_pred)
```

Out[69]:
```
0.9414414414414415
```

In [70]:
```
from sklearn.tree import plot_tree
```

In [71]:
```
plot_tree(tree_model)
```

Out[71]:
```
[Text(222.6299278846154, 212.26285714285714, 'X[0] <= -0.802\ngini = 0.493\ns
amples = 1773\nvalue = [992, 781]'),
 Text(200.33962912087912, 201.90857142857143, 'X[1] <= -0.243\ngini = 0.494\n
samples = 512\nvalue = [229, 283]'),
 Text(196.66050824175827, 191.5542857142857, 'gini = 0.0\nsamples = 229\nvalu
e = [229, 0]'),
 Text(204.01875, 191.5542857142857, 'gini = 0.0\nsamples = 283\nvalue = [0, 2
83]'),
 Text(244.92022664835167, 201.90857142857143, 'X[1] <= 0.738\ngini = 0.478\ns
amples = 1261\nvalue = [763, 498]'),
 Text(211.3769917582418, 191.5542857142857, 'X[0] <= 0.885\ngini = 0.5\nsampl
es = 963\nvalue = [480, 483]'),
 Text(155.32788461538462, 181.2, 'X[0] <= -0.114\ngini = 0.486\nsamples = 777
\nvalue = [454, 323]'),
 Text(54.26703296703297, 170.84571428571428, 'X[1] <= -0.276\ngini = 0.451\ns
amples = 350\nvalue = [230, 120]'),
 Text(20.235164835164838, 160.49142857142857, 'X[1] <= -0.418\ngini = 0.052\n
samples = 223\nvalue = [217, 6]'),
 Text(7.358241758241759, 150.13714285714286, 'X[0] <= -0.216\ngini = 0.019\ns
amples = 204\nvalue = [202, 2]'),
 Text(3.6791208791208794, 139.78285714285715, 'gini = 0.0\nsamples = 180\nval
ue = [180, 0]'),
 Text(11.037362637362637, 139.78285714285715, 'X[0] <= -0.215\ngini = 0.153\n
samples = 24\nvalue = [22, 2]'),
 Text(7.358241758241759, 129.42857142857144, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(14.716483516483517, 129.42857142857144, 'X[0] <= -0.192\ngini = 0.083\n
samples = 23\nvalue = [22, 1]'),
 Text(11.037362637362637, 119.07428571428571, 'X[0] <= -0.194\ngini = 0.278\n
samples = 6\nvalue = [5, 1]'),
 Text(7.358241758241759, 108.72, 'gini = 0.0\nsamples = 5\nvalue = [5, 0]'),
 Text(14.716483516483517, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(18.395604395604398, 119.07428571428571, 'gini = 0.0\nsamples = 17\nvalu
e = [17, 0]'),
 Text(33.112087912087915, 150.13714285714286, 'X[1] <= -0.393\ngini = 0.332\n
samples = 19\nvalue = [15, 4]'),
 Text(25.753846153846155, 139.78285714285715, 'X[0] <= -0.395\ngini = 0.444\n
samples = 3\nvalue = [1, 2]'),
 Text(22.074725274725274, 129.42857142857144, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(29.432967032967035, 129.42857142857144, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(40.470329670329676, 139.78285714285715, 'X[0] <= -0.244\ngini = 0.219\n
samples = 16\nvalue = [14, 2]'),
 Text(36.791208791208796, 129.42857142857144, 'X[0] <= -0.254\ngini = 0.375\n
samples = 8\nvalue = [6, 2]'),
 Text(33.112087912087915, 119.07428571428571, 'X[0] <= -0.299\ngini = 0.245\n
samples = 7\nvalue = [6, 1]'),
 Text(29.432967032967035, 108.72, 'gini = 0.0\nsamples = 4\nvalue = [4, 0]'),
 Text(36.791208791208796, 108.72, 'X[0] <= -0.275\ngini = 0.444\nsamples = 3
\nvalue = [2, 1]'),
 Text(33.112087912087915, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(40.470329670329676, 98.36571428571429, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(40.470329670329676, 119.07428571428571, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
```

```
 Text(44.14945054945055, 129.42857142857144, 'gini = 0.0\nsamples = 8\nvalue
= [8, 0]'),
 Text(88.2989010989011, 160.49142857142857, 'X[1] <= 0.044\ngini = 0.184\nsam
ples = 127\nvalue = [13, 114]'),
 Text(73.58241758241759, 150.13714285714286, 'X[0] <= -0.185\ngini = 0.457\ns
amples = 34\nvalue = [12, 22]'),
 Text(62.54505494505495, 139.78285714285715, 'X[1] <= -0.179\ngini = 0.384\ns
amples = 27\nvalue = [7, 20]'),
 Text(51.50769230769231, 129.42857142857144, 'X[1] <= -0.239\ngini = 0.5\nsam
ples = 8\nvalue = [4, 4]'),
 Text(47.82857142857143, 119.07428571428571, 'gini = 0.0\nsamples = 3\nvalue
= [0, 3]'),
 Text(55.18681318681319, 119.07428571428571, 'X[0] <= -0.219\ngini = 0.32\nsa
mples = 5\nvalue = [4, 1]'),
 Text(51.50769230769231, 108.72, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(58.86593406593407, 108.72, 'X[0] <= -0.2\ngini = 0.5\nsamples = 2\nvalu
e = [1, 1]'),
 Text(55.18681318681319, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
 Text(62.54505494505495, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(73.58241758241759, 129.42857142857144, 'X[1] <= 0.034\ngini = 0.266\nsa
mples = 19\nvalue = [3, 16]'),
 Text(69.9032967032967, 119.07428571428571, 'X[1] <= -0.097\ngini = 0.198\nsa
mples = 18\nvalue = [2, 16]'),
 Text(66.22417582417583, 108.72, 'gini = 0.0\nsamples = 7\nvalue = [0, 7]'),
 Text(73.58241758241759, 108.72, 'X[1] <= -0.087\ngini = 0.298\nsamples = 11
\nvalue = [2, 9]'),
 Text(69.9032967032967, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(77.26153846153846, 98.36571428571429, 'X[0] <= -0.336\ngini = 0.18\nsam
ples = 10\nvalue = [1, 9]'),
 Text(73.58241758241759, 88.01142857142858, 'gini = 0.0\nsamples = 6\nvalue =
[0, 6]'),
 Text(80.94065934065935, 88.01142857142858, 'X[0] <= -0.274\ngini = 0.375\nsa
mples = 4\nvalue = [1, 3]'),
 Text(77.26153846153846, 77.65714285714284, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(84.61978021978022, 77.65714285714284, 'gini = 0.0\nsamples = 3\nvalue =
[0, 3]'),
 Text(77.26153846153846, 119.07428571428571, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(84.61978021978022, 139.78285714285715, 'X[1] <= -0.196\ngini = 0.408\ns
amples = 7\nvalue = [5, 2]'),
 Text(80.94065934065935, 129.42857142857144, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(88.2989010989011, 129.42857142857144, 'X[1] <= -0.085\ngini = 0.278\nsa
mples = 6\nvalue = [5, 1]'),
 Text(84.61978021978022, 119.07428571428571, 'gini = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(91.97802197802199, 119.07428571428571, 'X[0] <= -0.17\ngini = 0.444\nsa
mples = 3\nvalue = [2, 1]'),
 Text(88.2989010989011, 108.72, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
 Text(95.65714285714286, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(103.01538461538462, 150.13714285714286, 'X[1] <= 0.181\ngini = 0.021\ns
amples = 93\nvalue = [1, 92]'),
 Text(99.33626373626375, 139.78285714285715, 'X[1] <= 0.164\ngini = 0.245\nsa
```

```
mples = 7\nvalue = [1, 6]'),
 Text(95.65714285714286, 129.42857142857144, 'gini = 0.0\nsamples = 6\nvalue
= [0, 6]'),
 Text(103.01538461538462, 129.42857142857144, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(106.6945054945055, 139.78285714285715, 'gini = 0.0\nsamples = 86\nvalue
= [0, 86]'),
 Text(256.3887362637363, 170.84571428571428, 'X[1] <= 0.123\ngini = 0.499\nsa
mples = 427\nvalue = [224, 203]'),
 Text(212.9291208791209, 160.49142857142857, 'X[0] <= 0.347\ngini = 0.406\nsa
mples = 258\nvalue = [73, 185]'),
 Text(160.96153846153848, 150.13714285714286, 'X[1] <= 0.072\ngini = 0.5\nsam
ples = 123\nvalue = [62, 61]'),
 Text(157.2824175824176, 139.78285714285715, 'X[0] <= -0.026\ngini = 0.498\ns
amples = 117\nvalue = [62, 55]'),
 Text(110.37362637362638, 129.42857142857144, 'X[1] <= -0.557\ngini = 0.393\n
samples = 26\nvalue = [19, 7]'),
 Text(106.6945054945055, 119.07428571428571, 'gini = 0.0\nsamples = 5\nvalue
= [5, 0]'),
 Text(114.05274725274727, 119.07428571428571, 'X[1] <= -0.481\ngini = 0.444\n
samples = 21\nvalue = [14, 7]'),
 Text(110.37362637362638, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(117.73186813186814, 108.72, 'X[0] <= -0.083\ngini = 0.42\nsamples = 20
\nvalue = [14, 6]'),
 Text(108.53406593406594, 98.36571428571429, 'X[1] <= -0.17\ngini = 0.48\nsam
ples = 5\nvalue = [2, 3]'),
 Text(104.85494505494506, 88.01142857142858, 'X[0] <= -0.106\ngini = 0.444\ns
amples = 3\nvalue = [2, 1]'),
 Text(101.17582417582419, 77.65714285714284, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(108.53406593406594, 77.65714285714284, 'X[1] <= -0.274\ngini = 0.5\nsam
ples = 2\nvalue = [1, 1]'),
 Text(104.85494505494506, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(112.21318681318682, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(112.21318681318682, 88.01142857142858, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(126.92967032967034, 98.36571428571429, 'X[0] <= -0.045\ngini = 0.32\nsa
mples = 15\nvalue = [12, 3]'),
 Text(119.57142857142858, 88.01142857142858, 'X[1] <= -0.061\ngini = 0.18\nsa
mples = 10\nvalue = [9, 1]'),
 Text(115.8923076923077, 77.65714285714284, 'gini = 0.0\nsamples = 7\nvalue =
[7, 0]'),
 Text(123.25054945054946, 77.65714285714284, 'X[0] <= -0.072\ngini = 0.444\ns
amples = 3\nvalue = [2, 1]'),
 Text(119.57142857142858, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(126.92967032967034, 67.30285714285714, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(134.2879120879121, 88.01142857142858, 'X[1] <= -0.41\ngini = 0.48\nsamp
les = 5\nvalue = [3, 2]'),
 Text(130.60879120879122, 77.65714285714284, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(137.96703296703296, 77.65714285714284, 'X[0] <= -0.043\ngini = 0.375\ns
amples = 4\nvalue = [3, 1]'),
 Text(134.2879120879121, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue =
```

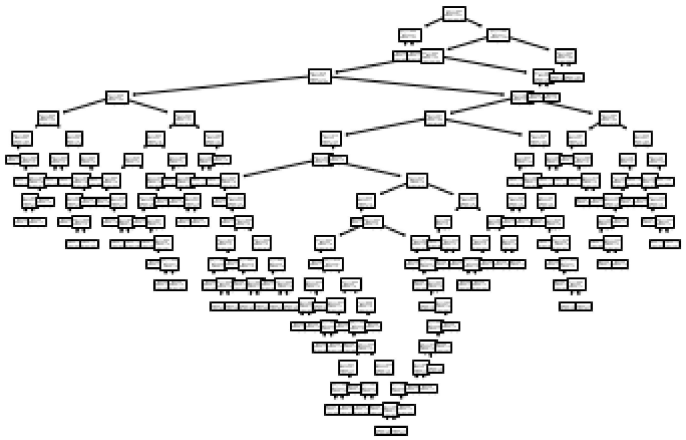
```
[0, 1]'),
 Text(141.64615384615385, 67.30285714285714, 'gini = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(204.1912087912088, 129.42857142857144, 'X[1] <= -0.112\ngini = 0.498\ns
amples = 91\nvalue = [43, 48]'),
 Text(178.43736263736264, 119.07428571428571, 'X[1] <= -0.751\ngini = 0.478\n
samples = 71\nvalue = [28, 43]'),
 Text(174.75824175824178, 108.72, 'gini = 0.0\nsamples = 6\nvalue = [0, 6]'),
 Text(182.11648351648353, 108.72, 'X[0] <= 0.232\ngini = 0.49\nsamples = 65\n
value = [28, 37]'),
 Text(158.20219780219782, 98.36571428571429, 'X[1] <= -0.662\ngini = 0.5\nsam
ples = 46\nvalue = [23, 23]'),
 Text(154.52307692307693, 88.01142857142858, 'gini = 0.0\nsamples = 5\nvalue
= [5, 0]'),
 Text(161.8813186813187, 88.01142857142858, 'X[0] <= 0.013\ngini = 0.493\nsam
ples = 41\nvalue = [18, 23]'),
 Text(152.68351648351648, 77.65714285714284, 'X[1] <= -0.364\ngini = 0.346\ns
amples = 9\nvalue = [2, 7]'),
 Text(149.00439560439563, 67.30285714285714, 'X[0] <= -0.015\ngini = 0.444\ns
amples = 3\nvalue = [2, 1]'),
 Text(145.32527472527474, 56.94857142857143, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(152.68351648351648, 56.94857142857143, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(156.36263736263737, 67.30285714285714, 'gini = 0.0\nsamples = 6\nvalue
= [0, 6]'),
 Text(171.0791208791209, 77.65714285714284, 'X[1] <= -0.486\ngini = 0.5\nsamp
les = 32\nvalue = [16, 16]'),
 Text(163.72087912087912, 67.30285714285714, 'X[1] <= -0.601\ngini = 0.245\ns
amples = 7\nvalue = [1, 6]'),
 Text(160.04175824175826, 56.94857142857143, 'X[1] <= -0.628\ngini = 0.444\ns
amples = 3\nvalue = [1, 2]'),
 Text(156.36263736263737, 46.59428571428572, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(163.72087912087912, 46.59428571428572, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(167.4, 56.94857142857143, 'gini = 0.0\nsamples = 4\nvalue = [0, 4]'),
 Text(178.43736263736264, 67.30285714285714, 'X[1] <= -0.128\ngini = 0.48\nsa
mples = 25\nvalue = [15, 10]'),
 Text(174.75824175824178, 56.94857142857143, 'X[0] <= 0.059\ngini = 0.454\nsa
mples = 23\nvalue = [15, 8]'),
 Text(171.0791208791209, 46.59428571428572, 'gini = 0.0\nsamples = 5\nvalue =
[5, 0]'),
 Text(178.43736263736264, 46.59428571428572, 'X[0] <= 0.095\ngini = 0.494\nsa
mples = 18\nvalue = [10, 8]'),
 Text(169.23956043956045, 36.24000000000001, 'X[1] <= -0.361\ngini = 0.408\ns
amples = 7\nvalue = [2, 5]'),
 Text(165.56043956043956, 25.8857142857143, 'X[0] <= 0.081\ngini = 0.444\nsam
ples = 3\nvalue = [2, 1]'),
 Text(161.8813186813187, 15.531428571428563, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(169.23956043956045, 15.531428571428563, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(172.91868131868134, 25.8857142857143, 'gini = 0.0\nsamples = 4\nvalue =
[0, 4]'),
 Text(187.63516483516486, 36.24000000000001, 'X[1] <= -0.354\ngini = 0.397\ns
amples = 11\nvalue = [8, 3]'),
```

```
 Text(180.27692307692308, 25.8857142857143, 'X[0] <= 0.197\ngini = 0.444\nsam
ples = 3\nvalue = [1, 2]'),
 Text(176.5978021978022, 15.531428571428563, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(183.95604395604397, 15.531428571428563, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(194.9934065934066, 25.8857142857143, 'X[0] <= 0.121\ngini = 0.219\nsamp
les = 8\nvalue = [7, 1]'),
 Text(191.31428571428572, 15.531428571428563, 'X[0] <= 0.114\ngini = 0.375\ns
amples = 4\nvalue = [3, 1]'),
 Text(187.63516483516486, 5.177142857142854, 'gini = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(194.9934065934066, 5.177142857142854, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
 Text(198.6725274725275, 15.531428571428563, 'gini = 0.0\nsamples = 4\nvalue
= [4, 0]'),
 Text(182.11648351648353, 56.94857142857143, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(206.03076923076924, 98.36571428571429, 'X[1] <= -0.426\ngini = 0.388\ns
amples = 19\nvalue = [5, 14]'),
 Text(202.35164835164838, 88.01142857142858, 'gini = 0.0\nsamples = 8\nvalue
= [0, 8]'),
 Text(209.70989010989013, 88.01142857142858, 'X[1] <= -0.334\ngini = 0.496\ns
amples = 11\nvalue = [5, 6]'),
 Text(206.03076923076924, 77.65714285714284, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(213.389010989011, 77.65714285714284, 'X[0] <= 0.257\ngini = 0.444\nsamp
les = 9\nvalue = [3, 6]'),
 Text(209.70989010989013, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(217.06813186813187, 67.30285714285714, 'X[0] <= 0.302\ngini = 0.375\nsa
mples = 8\nvalue = [2, 6]'),
 Text(213.389010989011, 56.94857142857143, 'X[0] <= 0.295\ngini = 0.48\nsampl
es = 5\nvalue = [2, 3]'),
 Text(209.70989010989013, 46.59428571428572, 'X[0] <= 0.279\ngini = 0.375\nsa
mples = 4\nvalue = [1, 3]'),
 Text(206.03076923076924, 36.24000000000001, 'X[0] <= 0.267\ngini = 0.5\nsamp
les = 2\nvalue = [1, 1]'),
 Text(202.35164835164838, 25.8857142857143, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
 Text(209.70989010989013, 25.8857142857143, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(213.389010989011, 36.24000000000001, 'gini = 0.0\nsamples = 2\nvalue =
[0, 2]'),
 Text(217.06813186813187, 46.59428571428572, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(220.74725274725276, 56.94857142857143, 'gini = 0.0\nsamples = 3\nvalue
= [0, 3]'),
 Text(229.94505494505495, 119.07428571428571, 'X[1] <= -0.033\ngini = 0.375\n
samples = 20\nvalue = [15, 5]'),
 Text(217.06813186813187, 108.72, 'X[0] <= 0.182\ngini = 0.18\nsamples = 10\n
value = [9, 1]'),
 Text(213.389010989011, 98.36571428571429, 'gini = 0.0\nsamples = 6\nvalue =
[6, 0]'),
 Text(220.74725274725276, 98.36571428571429, 'X[0] <= 0.244\ngini = 0.375\nsa
mples = 4\nvalue = [3, 1]'),
 Text(217.06813186813187, 88.01142857142858, 'gini = 0.0\nsamples = 1\nvalue
```

```
= [0, 1]'),
 Text(224.42637362637365, 88.01142857142858, 'gini = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(242.82197802197803, 108.72, 'X[1] <= 0.055\ngini = 0.48\nsamples = 10\n
value = [6, 4]'),
 Text(235.46373626373628, 98.36571428571429, 'X[1] <= 0.033\ngini = 0.48\nsam
ples = 5\nvalue = [2, 3]'),
 Text(231.7846153846154, 88.01142857142858, 'X[1] <= -0.013\ngini = 0.444\nsa
mples = 3\nvalue = [2, 1]'),
 Text(228.10549450549453, 77.65714285714284, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(235.46373626373628, 77.65714285714284, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(239.14285714285717, 88.01142857142858, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(250.1802197802198, 98.36571428571429, 'X[0] <= 0.293\ngini = 0.32\nsamp
les = 5\nvalue = [4, 1]'),
 Text(246.5010989010989, 88.01142857142858, 'gini = 0.0\nsamples = 4\nvalue =
[4, 0]'),
 Text(253.8593406593407, 88.01142857142858, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
 Text(164.64065934065934, 139.78285714285715, 'gini = 0.0\nsamples = 6\nvalue
= [0, 6]'),
 Text(264.8967032967033, 150.13714285714286, 'X[1] <= 0.025\ngini = 0.15\nsam
ples = 135\nvalue = [11, 124]'),
 Text(257.53846153846155, 139.78285714285715, 'X[1] <= -0.326\ngini = 0.076\n
samples = 127\nvalue = [5, 122]'),
 Text(253.8593406593407, 129.42857142857144, 'gini = 0.0\nsamples = 105\nvalu
e = [0, 105]'),
 Text(261.21758241758243, 129.42857142857144, 'X[0] <= 0.501\ngini = 0.351\ns
amples = 22\nvalue = [5, 17]'),
 Text(253.8593406593407, 119.07428571428571, 'X[1] <= -0.308\ngini = 0.142\ns
amples = 13\nvalue = [1, 12]'),
 Text(250.1802197802198, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(257.53846153846155, 108.72, 'gini = 0.0\nsamples = 12\nvalue = [0, 1
2]'),
 Text(268.5758241758242, 119.07428571428571, 'X[0] <= 0.52\ngini = 0.494\nsam
ples = 9\nvalue = [4, 5]'),
 Text(264.8967032967033, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(272.2549450549451, 108.72, 'X[1] <= -0.175\ngini = 0.469\nsamples = 8\n
value = [3, 5]'),
 Text(268.5758241758242, 98.36571428571429, 'gini = 0.0\nsamples = 3\nvalue =
[0, 3]'),
 Text(275.9340659340659, 98.36571428571429, 'X[1] <= -0.117\ngini = 0.48\nsam
ples = 5\nvalue = [3, 2]'),
 Text(272.2549450549451, 88.01142857142858, 'gini = 0.0\nsamples = 2\nvalue =
[2, 0]'),
 Text(279.6131868131868, 88.01142857142858, 'X[1] <= -0.085\ngini = 0.444\nsa
mples = 3\nvalue = [1, 2]'),
 Text(275.9340659340659, 77.65714285714284, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
 Text(283.2923076923077, 77.65714285714284, 'X[1] <= -0.051\ngini = 0.5\nsamp
les = 2\nvalue = [1, 1]'),
 Text(279.6131868131868, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(286.9714285714286, 67.30285714285714, 'gini = 0.0\nsamples = 1\nvalue =
[0, 1]'),
```

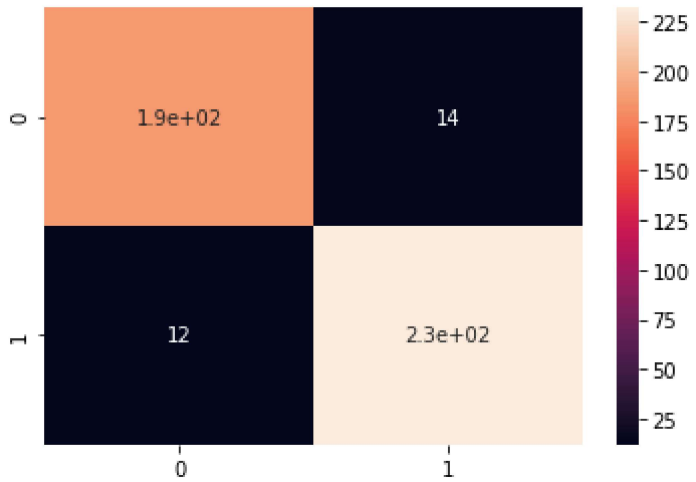
```
 Text(272.2549450549451, 139.78285714285715, 'X[0] <= 0.461\ngini = 0.375\nsa
mples = 8\nvalue = [6, 2]'),
 Text(268.5758241758242, 129.42857142857144, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(275.9340659340659, 129.42857142857144, 'gini = 0.0\nsamples = 6\nvalue
= [6, 0]'),
 Text(299.8483516483517, 160.49142857142857, 'X[0] <= 0.089\ngini = 0.19\nsam
ples = 169\nvalue = [151, 18]'),
 Text(283.2923076923077, 150.13714285714286, 'X[1] <= 0.134\ngini = 0.245\nsa
mples = 14\nvalue = [2, 12]'),
 Text(279.6131868131868, 139.78285714285715, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(286.9714285714286, 139.78285714285715, 'X[1] <= 0.504\ngini = 0.142\nsa
mples = 13\nvalue = [1, 12]'),
 Text(283.2923076923077, 129.42857142857144, 'gini = 0.0\nsamples = 10\nvalue
= [0, 10]'),
 Text(290.6505494505495, 129.42857142857144, 'X[1] <= 0.614\ngini = 0.444\nsa
mples = 3\nvalue = [1, 2]'),
 Text(286.9714285714286, 119.07428571428571, 'gini = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(294.32967032967036, 119.07428571428571, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(316.40439560439563, 150.13714285714286, 'X[0] <= 0.246\ngini = 0.074\ns
amples = 155\nvalue = [149, 6]'),
 Text(309.04615384615386, 139.78285714285715, 'X[1] <= 0.38\ngini = 0.463\nsa
mples = 11\nvalue = [7, 4]'),
 Text(305.36703296703297, 129.42857142857144, 'X[0] <= 0.186\ngini = 0.5\nsam
ples = 8\nvalue = [4, 4]'),
 Text(301.6879120879121, 119.07428571428571, 'X[1] <= 0.366\ngini = 0.444\nsa
mples = 6\nvalue = [4, 2]'),
 Text(298.00879120879125, 108.72, 'X[1] <= 0.348\ngini = 0.32\nsamples = 5\nv
alue = [4, 1]'),
 Text(294.32967032967036, 98.36571428571429, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(301.6879120879121, 98.36571428571429, 'X[1] <= 0.352\ngini = 0.444\nsam
ples = 3\nvalue = [2, 1]'),
 Text(298.00879120879125, 88.01142857142858, 'gini = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(305.36703296703297, 88.01142857142858, 'gini = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(305.36703296703297, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(309.04615384615386, 119.07428571428571, 'gini = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(312.72527472527474, 129.42857142857144, 'gini = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(323.7626373626374, 139.78285714285715, 'X[0] <= 0.35\ngini = 0.027\nsam
ples = 144\nvalue = [142, 2]'),
 Text(320.0835164835165, 129.42857142857144, 'X[0] <= 0.337\ngini = 0.219\nsa
mples = 16\nvalue = [14, 2]'),
 Text(316.40439560439563, 119.07428571428571, 'gini = 0.0\nsamples = 13\nvalu
e = [13, 0]'),
 Text(323.7626373626374, 119.07428571428571, 'X[1] <= 0.396\ngini = 0.444\nsa
mples = 3\nvalue = [1, 2]'),
 Text(320.0835164835165, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(327.44175824175824, 108.72, 'X[0] <= 0.342\ngini = 0.5\nsamples = 2\nva
lue = [1, 1]'),
 Text(323.7626373626374, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue =
```

```
[0, 1]'),
 Text(331.1208791208791, 98.36571428571429, 'gini = 0.0\nsamples = 1\nvalue =
[1, 0]'),
 Text(327.44175824175824, 129.42857142857144, 'gini = 0.0\nsamples = 128\nval
ue = [128, 0]'),
 Text(267.4260989010989, 181.2, 'X[1] <= -0.066\ngini = 0.24\nsamples = 186\n
value = [26, 160]'),
 Text(263.74697802197807, 170.84571428571428, 'gini = 0.0\nsamples = 160\nval
ue = [0, 160]'),
 Text(271.1052197802198, 170.84571428571428, 'gini = 0.0\nsamples = 26\nvalue
= [26, 0]'),
 Text(278.46346153846156, 191.5542857142857, 'X[0] <= -0.054\ngini = 0.096\ns
amples = 298\nvalue = [283, 15]'),
 Text(274.78434065934067, 181.2, 'gini = 0.0\nsamples = 15\nvalue = [0, 1
5]'),
 Text(282.14258241758245, 181.2, 'gini = 0.0\nsamples = 283\nvalue = [283,
0]')]
```



In [72]:
```
sns.heatmap(confusion_matrix(y_test, y_pred, labels=[1,0]),annot=True)
```

Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x14028eecf10>



Accuracy of Decision tree : 94.4%

hyperparameter tunning

import GridSearchCV

In [73]:
```
from sklearn.model_selection import GridSearchCV
```

In [74]:
```
param_grid = {"max_depth": [2,3,4,5,6],
              "min_samples_split": [20,15,10,5],
              "min_samples_leaf": [10,8,6,4,2]}
```

In [75]:
```
param_grid
```

Out[75]:
```
{'max_depth': [2, 3, 4, 5, 6],
 'min_samples_split': [20, 15, 10, 5],
 'min_samples_leaf': [10, 8, 6, 4, 2]}
```

In [76]:
```
[param_grid]
```

Out[76]:
```
[{'max_depth': [2, 3, 4, 5, 6],
  'min_samples_split': [20, 15, 10, 5],
  'min_samples_leaf': [10, 8, 6, 4, 2]}]
```

In [77]:
```
model = DecisionTreeClassifier()
```

In [78]:
```
clf = GridSearchCV(model, param_grid)
```

In [79]:
```
clf
```

Out[79]:
```
GridSearchCV(estimator=DecisionTreeClassifier(),
             param_grid={'max_depth': [2, 3, 4, 5, 6],
                         'min_samples_leaf': [10, 8, 6, 4, 2],
                         'min_samples_split': [20, 15, 10, 5]})
```

In [80]:
```
clf.fit(X_train,y_train)
```

Out[80]:
```
GridSearchCV(estimator=DecisionTreeClassifier(),
             param_grid={'max_depth': [2, 3, 4, 5, 6],
                         'min_samples_leaf': [10, 8, 6, 4, 2],
                         'min_samples_split': [20, 15, 10, 5]})
```

In [81]:
```
clf.best_estimator_
```

Out[81]:
```
DecisionTreeClassifier(max_depth=6, min_samples_leaf=10, min_samples_split=2
0)
```

In [82]:
```
# make the final model based on best estimated condition
```

In [83]:
```
final_model = DecisionTreeClassifier(max_depth=6, min_samples_leaf=10, min_sam
ples_split=20)
```

In [84]:
```
final_model
```

Out[84]:
```
DecisionTreeClassifier(max_depth=6, min_samples_leaf=10, min_samples_split=2
0)
```

In [85]:
```
final_model.fit(X_train,y_train)
```

Out[85]:
```
DecisionTreeClassifier(max_depth=6, min_samples_leaf=10, min_samples_split=2
0)
```

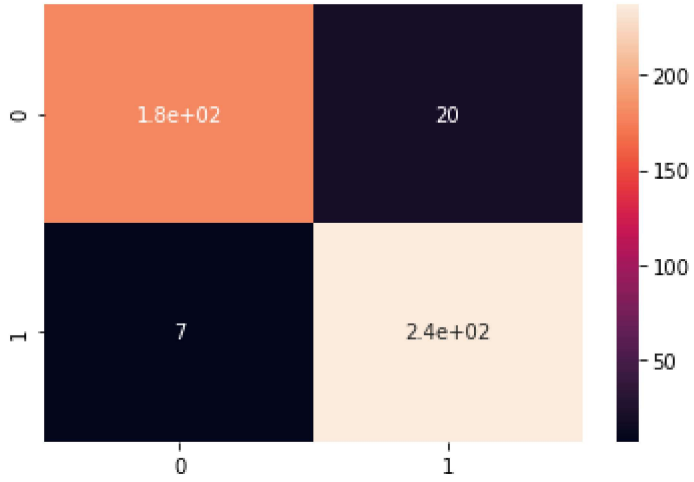
In [86]:
```
y_pred = final_model.predict(X_test)
```

In [87]:
```
accuracy_score(y_test,y_pred)
```

Out[87]:
```
0.9391891891891891
```

In [88]:
```
sns.heatmap(confusion_matrix(y_test, y_pred, labels=[1,0]),annot=True)
```

Out[88]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x14028faaf40>
```



Accuracy : 94%

In [ ]:
```
```

KNN Model

import library

In [89]:
```
from sklearn.neighbors import KNeighborsClassifier
```

Training

Let's start the algorithm with k=5 for now:

In [90]:
```
k = 5
#Train Model
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

Out[90]:
```
KNeighborsClassifier()
```

Predict Model

In [91]:
```
yhat = neigh.predict(X_test)
yhat[0:5]
```

Out[91]:
```
array([0, 0, 0, 1, 0], dtype=int64)
```

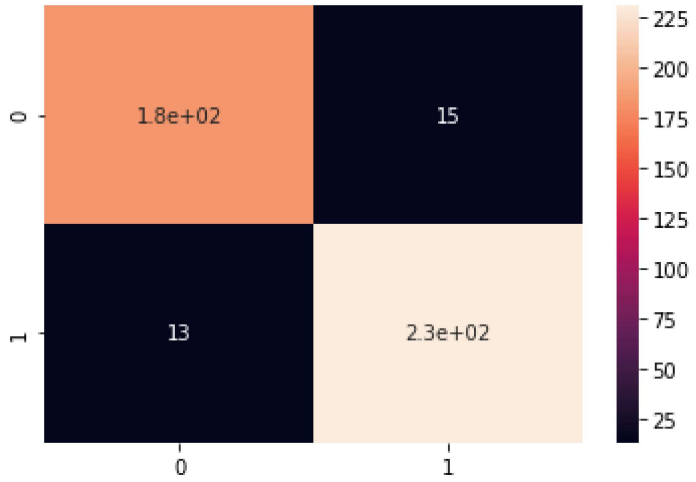
Accuracy

In [92]:
```
accuracy_score(y_test, yhat)
```

Out[92]:
```
0.9369369369369369
```

In [93]:
```
sns.heatmap(confusion_matrix(y_test, yhat, labels=[1,0]),annot=True)
```

Out[93]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x14028ff6220>
```



Again, Build the model with K=6

In [94]:
```
k = 9
#Train Model
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

Out[94]:
```
KNeighborsClassifier(n_neighbors=9)
```

Predict Model

In [95]:
```
yhat = neigh.predict(X_test)
yhat[0:5]
```

Out[95]:
```
array([0, 0, 0, 1, 0], dtype=int64)
```

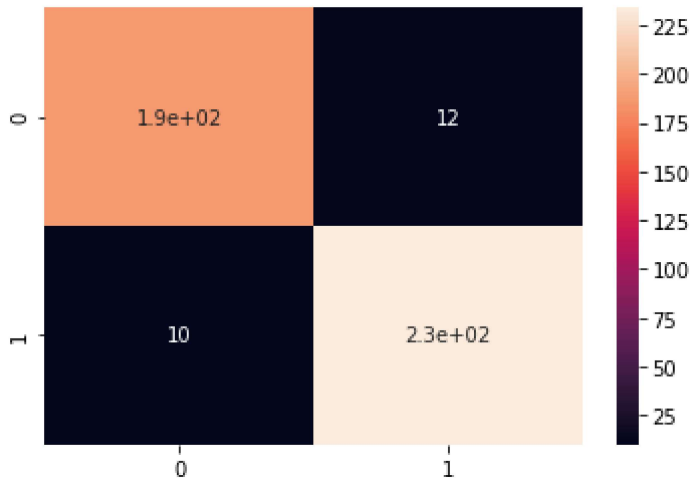
Accuracy

In [96]:
```
accuracy_score(y_test, yhat)
```

Out[96]:
```
0.9504504504504504
```

In [97]:
```
sns.heatmap(confusion_matrix(y_test, yhat, labels=[1,0]),annot=True)
```

Out[97]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x14029082be0>
```



Hyperparameter tunning

import library

In [98]:
```
from sklearn.model_selection import GridSearchCV
```

In [99]:
```
param_grid = {'n_neighbors':[1,3,5,7,9,11,13,15,17,19]}
```

In [100]:
```
model = KNeighborsClassifier()
model
```

Out[100]:
```
KNeighborsClassifier()
```

In [101]:
```
clf = GridSearchCV(model, param_grid)
```

In [102]:
```
clf
```

Out[102]:
```
GridSearchCV(estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]})
```

In [103]:
```
clf.fit(X_train,y_train)
```

Out[103]:
```
GridSearchCV(estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]})
```

In [104]:
```
clf.best_estimator_
```

Out[104]:
```
KNeighborsClassifier(n_neighbors=19)
```

In [105]:
```
# make the final model based on best estimated condition
```

In [106]:
```
final_model = KNeighborsClassifier(n_neighbors=19)
```

In [107]:
```
final_model
```

Out[107]:
```
KNeighborsClassifier(n_neighbors=19)
```

In [108]:
```
final_model.fit(X_train,y_train)
```

Out[108]:
```
KNeighborsClassifier(n_neighbors=19)
```

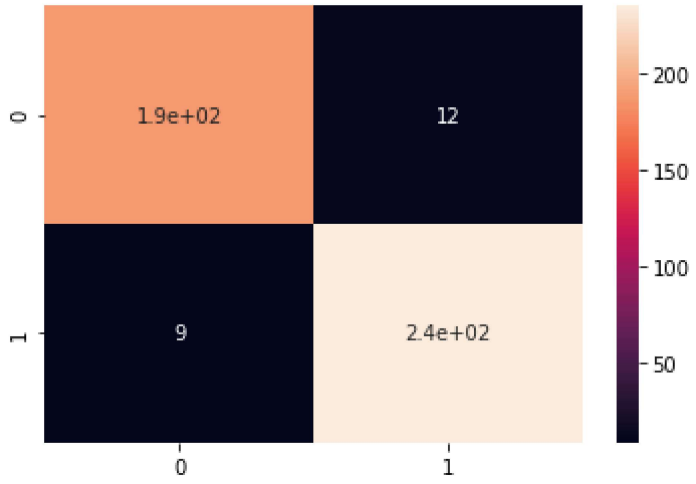
In [109]:
```
y_pred = final_model.predict(X_test)
```

In [110]:
```
accuracy_score(y_test,y_pred)
```

Out[110]:
```
0.9527027027027027
```

In [111]:
```
sns.heatmap(confusion_matrix(y_test, y_pred, labels=[1,0]),annot=True)
```

Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0x1402906c7c0>



Accuracy of KNN with n_neighbors=19 is : 95.3%

In [ ]:

ACCURACY OF MODELS

Logisic Regression	SVM_LINEAR	SVM_rbf	DECISION TREE	KNN
72.5%	71.2 %	95%	94.4%	95.3

For this dataset, SVM(kernel='rbf'), Decision Tree and KNN Model give higher accuracy.

END

In [ ]: