CS 634 FINAL TERM PROJECT

OPTION1: SUPERVISED DATA MINING (CLASSIFICATION)

SUBMITTED BY: Kavitha Kannanunny
NJIT UCID: kk46
Email:kk46@njit.edu

INSTRUCTOR: Dr. Jason Wang

Category

- **Category 3**: Decision Trees (CART)
- **Category 5**: Naive bayes (Multinomial)
- **Category 10**: Scikit learn

Programming Language

Python

Hardware

Laptop

Software

Operating System : Windows

Libraries used: Scikit-learn, Pandas, Graphviz, Matplotlib, pydotplus, numpy
Libraries need to be first installed using `pip install library_name` .
Once installed they can be imported and used for our purposes.

Dataset

Title: Estimation of obesity levels based on eating habits and physical condition

http://archive.ics.uci.edu/ml/datasets/EstimationOfObesity

This dataset includes data for the estimation of obesity levels in individuals from the countries of Mexico, Peru and Colombia, based on their eating habits and physical condition. The data contains 17 attributes and 2111 records, the records are labeled with the class variable `NObeyesdad` (Obesity Level), that allows classification of the data using the values of Insufficient Weight, Normal Weight, Overweight Level I, Overweight Level II, Obesity Type I, Obesity Type II and Obesity Type III. 77% of the data was generated synthetically using the Weka tool and the SMOTE filter, 23% of the data was collected directly from users through a web platform.

- **Dataset Characteristic** Multivariate
- **Number of Instances** 2111
- **Number of Attributes** 17
- **Attribute Characteristic** Integer and categorical

Data Description

`Gender` - Male/ Female
`Age` - Age of the individual
`Height` - Height of the individual
`Family_history_with_overweight` - Whether overweight exists in family
`FAVC` - Frequent consumption of high caloric food
`FCVC` - frequency of consumption of vegetables
`NCP` - Number of main meals
`CAEC` - Consumption of food between meals
`SMOKE` - Whether person smokes or not
`CH2O` - Consumption of water daily
`SCC` - Calories consumption monitoring
`FAF` - Physical activity frequency
`TUE` - Time using technology devices
`CALC` - Consumption of alcohol
`MTRANS` - Transportation used

`NObeyesdad` - Obesity level (Target)

Importing required libraries

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import graphviz
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.naive_bayes import MultinomialNB
from sklearn import model_selection
import pydotplus
```

Loading Dataset

- Load the dataset from the given file location and store it as a dataframe
- head() is used to print rows of the dataframe. Default number of rows is 5.

In [3]:
```
df=pd.read_csv('C:\\Users\\Archer\\Desktop\\dataset.csv')
df.head()
```

Out[3]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC
0	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes
1	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes
2	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes
3	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes
4	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes

Data Cleaning and Pre-processing

Our dataset might contain null or missing values due to data collection errors. This has to be removed before we train the model inorder to prevent noisy data and misclassification. In addition, some columns need to be converted to a form that is advisable for training the model. For example, categorical values are converted to numerical values by label encoding. This is called pre-processing. Here, we go through some common data cleanup and pre-processing.

Checking for Missing values

In [4]:
```
missing_values=df.isnull().sum()/len(df)*100
missing_values
```

Out[4]:
```
Gender                            0.0
Age                               0.0
Height                            0.0
Weight                            0.0
family_history_with_overweight    0.0
FAVC                              0.0
FCVC                              0.0
NCP                               0.0
CAEC                              0.0
SMOKE                             0.0
CH2O                              0.0
SCC                               0.0
FAF                               0.0
TUE                               0.0
CALC                              0.0
MTRANS                            0.0
NObeyesdad                        0.0
dtype: float64
```

Encoding categorical data

`LabelEncoder` is used to encode the categorical data. For example `gender` which has values *male* and *female* are encoded to 1 and 2. This allows machine learning models are trained on numerical values.

In [5]:
```
labelencoder = LabelEncoder()
df['Gender'] = labelencoder.fit_transform(df['Gender'])
df['family_history_with_overweight'] = labelencoder.fit_transform(df['family_h
istory_with_overweight'])
df['FAVC'] = labelencoder.fit_transform(df['FAVC'])
df['CAEC'] = labelencoder.fit_transform(df['CAEC'])
df['SMOKE'] = labelencoder.fit_transform(df['SMOKE'])
df['SCC'] = labelencoder.fit_transform(df['SCC'])
df['CALC'] = labelencoder.fit_transform(df['CALC'])
df['MTRANS'] = labelencoder.fit_transform(df['MTRANS'])
df['NObeyesdad'] = labelencoder.fit_transform(df['NObeyesdad'])
df.head()
```

Out[5]:

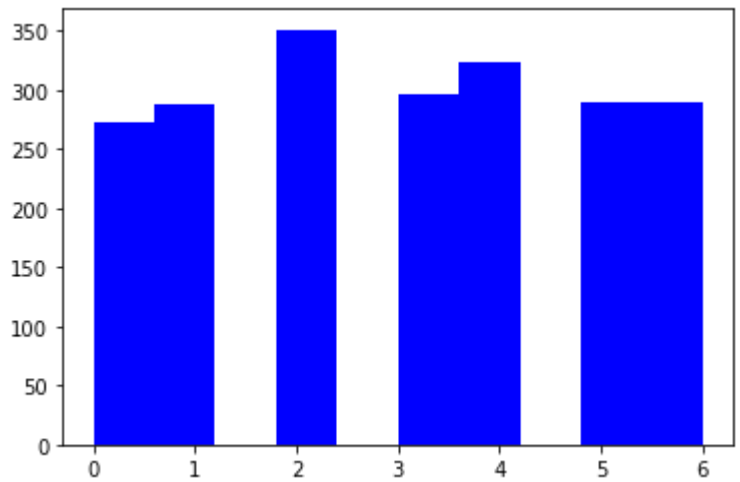
	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SM
0	0	21.0	1.62	64.0	1	0	2.0	3.0	2	
1	0	21.0	1.52	56.0	1	0	3.0	3.0	2	
2	1	23.0	1.80	77.0	1	0	2.0	3.0	2	
3	1	27.0	1.80	87.0	0	0	3.0	3.0	2	
4	1	22.0	1.78	89.8	0	0	2.0	1.0	2	

Checking the Skewness of data

Data is said to be skewed when plotting the values shows a asymmetry as compared to the normal distribution or bell curve distribution of data. Skewed data doesn't train the model well and affect the classification/regression ability of the model.

In [6]:
```
print ("Skew is:", df.NObeyesdad.skew())
plt.hist(df.NObeyesdad, color='blue')
plt.show()
```

```
Skew is: 0.006754449086780782
```



Since skewness of the dataset is closer to 0, it shows the data is not skewed and good to proceed. If its skewed we remove the skew by taking log of the values.

Splitting data into features and target

Now, our dataframe does not have any null values or categorical values and now we can split it into features and target to perform supervised learning.

In [7]:
```
features=df.iloc[:,:-1]
target=df.iloc[:,-1]
```

In [8]:
```
features.head()
```

Out[8]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SM
0	0	21.0	1.62	64.0	1	0	2.0	3.0	2	
1	0	21.0	1.52	56.0	1	0	3.0	3.0	2	
2	1	23.0	1.80	77.0	1	0	2.0	3.0	2	
3	1	27.0	1.80	87.0	0	0	3.0	3.0	2	
4	1	22.0	1.78	89.8	0	0	2.0	1.0	2	

In [9]:
```
target.head()
```

Out[9]:
```
0    1
1    1
2    1
3    5
4    6
Name: NObeyesdad, dtype: int32
```

Splitting the data to train and test sets

Feature data is now split into testing and training sets. We train the model using the training set and test the model using the test set.
X represents the feature set and Y represents the target set.

In [10]:
```
trainX,testX,trainY,testY = train_test_split(features, target, random_state=1)
```

Decision Tree Classifier

- Decision Tree is a supervised machine learning algorithm.
- DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset
- Its simple and efficient especially when number of classes are less and dataset is not too big.

Decsion trees consist of: [1]

- **Nodes** : test for value of certain attribute
- **Branches/edges**:intermediate outcome of a test and connect to next node
- **Leaf Nodes**: Predict the outcome(class/label)

Key Concepts:

- Entropy is the impurity of a set of examples. Entropy is 0 for homogeneous dataset
- Information Gain is the expected reduction in entropy caused by partitioning.We choose the branch with highest gain to split the tree.
- Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest information gain (IG)
- In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the samples at each leaf node all belong to the same class. [3]

CART Algorithm [3]

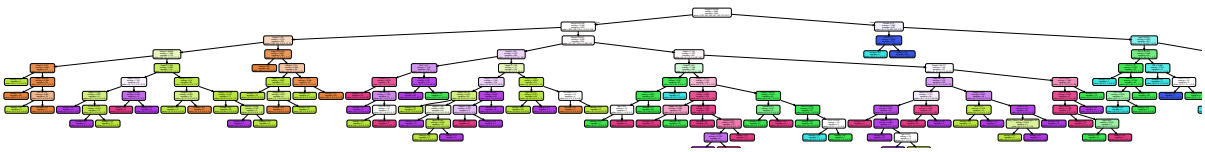
- Scikit-learn utilises optimized version of CART algorithm.
- CART stands for Classification And Regression Tree.
- CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.

In [11]:
```
tree = DecisionTreeClassifier(criterion = 'entropy').fit(trainX,trainY)
```

In [12]:
```
dot_data=export_graphviz(tree, out_file=None, filled=True, rounded=True,
                                special_characters=True,
                                feature_names=features.columns)
pydot_graph = pydotplus.graph_from_dot_data(dot_data)
pydot_graph.set_size('"10,8"')
```

In [13]:
```
gvz_graph = graphviz.Source(pydot_graph.to_string())
gvz_graph
```

Out[13]:



Predicting the values

In [14]:
```
predictions=tree.predict(testX)
```

In [15]:
```
print(accuracy_score(predictions, testY))
```

```
0.9450757575757576
```

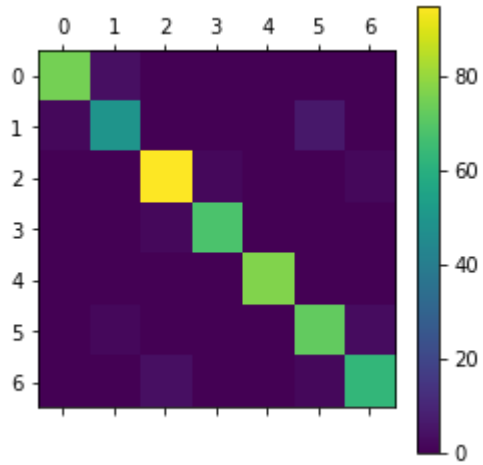
Confusion Matrix

Confusion matrices are useful to inform what kinds of errors your models tend to make.

They work for binary classification and multi-class classification.

In [16]:
```
print(metrics.confusion_matrix(testY,predictions))
plt.matshow(metrics.confusion_matrix(testY,predictions))
plt.colorbar()
plt.show()
```

```
[[75  4  0  0  0  0  0]
 [ 2 49  0  0  0  6  0]
 [ 0  0 95  2  0  0  2]
 [ 0  0  2 68  0  0  0]
 [ 0  0  0  0 77  0  0]
 [ 0  2  0  0  0 72  3]
 [ 0  0  4  0  0  2 63]]
```



Classification report[7]

- **Recall**: Out of all the positive classes, how much we predicted correctly. It should be high as possible.

$$Recall = \frac{TP}{TP + FN}$$

- **Precision**: Out of all the positive classes we have predicted correctly, how many are actually positive.

$$Precision = \frac{TP}{TP + FP}$$

- **Accuracy**: Out of all the classes, how much we predicted correctly

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **F-measure**: F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more. Its values are between 0 and 1. Models with values closer to 1 are better than the one which are closer to 0.

$$F\text{-}measure = \frac{2*Recall*Precision}{Recall + Precision}$$

```
In [17]: print(classification_report(testY,predictions))
```

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96        79
           1       0.89      0.86      0.88        57
           2       0.94      0.96      0.95        99
           3       0.97      0.97      0.97        70
           4       1.00      1.00      1.00        77
           5       0.90      0.94      0.92        77
           6       0.93      0.91      0.92        69

    accuracy                           0.95       528
   macro avg       0.94      0.94      0.94       528
weighted avg       0.95      0.95      0.95       528
```

10 Fold cross validation [4]

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample.[5]
It splits the dataset into k consecutive folds (without shuffling by default). Each fold is then used once as a test set while the k - 1 remaining folds form the training set.
The procedure has a parameter called `n_splits` that refers to the number of groups that a given data sample is to be split into. When `n_splits` =10, its 10 fold cross validation.
`shuffle` determines whether to shuffle the data before splitting into branches. When `shuffle` is True, `random_state` affects the ordering of the indices, which controls the randomness of each fold. By default `shuffle` is False.
10-fold cross validation would perform the fitting procedure a total of ten times, with each fit being performed on a training set consisting of 90% of the total training set selected at random, with the remaining 10% used as a hold out set for validation
Finally, accuracy is calculated by taking the mean of cross validation score of all splits.

In [18]:
```
fold10= model_selection.KFold(n_splits=10,shuffle=True,random_state=20)
result_kfold=model_selection.cross_val_score(tree,features,target,cv=fold10)
accuracy=result_kfold.mean()*100
print("Accuracy: "+"{:.2f}".format(accuracy))
```

```
Accuracy: 95.41
```

Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. **Bayes theorem**:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Multinomial Naive Bayes classifier is a specific instance of a Naive Bayes classifier which uses a multinomial distribution for each of the features. [2] **Parameters**:[6]
`alpha` - smoothing parameter (0 for no smoothing).
`fit_prior` -Whether to learn class prior probabilities or not. If false, a uniform prior will be used
`class_prior` - Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

In [19]:
```
model = MultinomialNB(alpha=1.0,fit_prior=True)
model.fit(trainX, trainY)
```

Out[19]:
```
MultinomialNB()
```

In [20]:
```
y_model = model.predict(testX)
```

In [21]:
```
accuracy_score(testY, y_model)
```

Out[21]:
```
0.6174242424242424
```

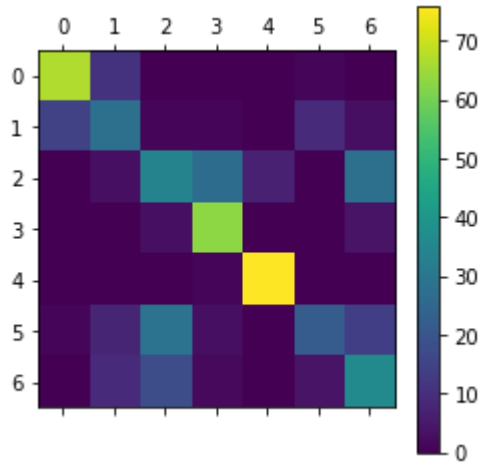
Confusion Matrix

Confusion matrices are useful to inform what kinds of errors your models tend to make.

They work for binary classification and multi-class classification.

In [22]:
```
print(metrics.confusion_matrix(testY,y_model))
plt.matshow(metrics.confusion_matrix(testY,y_model))
plt.colorbar()
plt.show()
```

```
[[67 11  0  0  0  1  0]
 [15 28  1  1  0  9  3]
 [ 0  3 34 27  7  0 28]
 [ 0  0  3 63  0  0  4]
 [ 0  0  0  1 76  0  0]
 [ 1  8 29  3  0 22 14]
 [ 0  9 18  2  0  4 36]]
```



Classification Report

```
In [23]: print(classification_report(testY,y_model))
```

```
              precision    recall  f1-score   support

           0       0.81      0.85      0.83        79
           1       0.47      0.49      0.48        57
           2       0.40      0.34      0.37        99
           3       0.65      0.90      0.75        70
           4       0.92      0.99      0.95        77
           5       0.61      0.29      0.39        77
           6       0.42      0.52      0.47        69

    accuracy                           0.62       528
   macro avg       0.61      0.63      0.61       528
weighted avg       0.61      0.62      0.60       528
```

10 Fold cross validation

```
In [24]: fold10= model_selection.KFold(n_splits=10,shuffle=True,random_state=20)
         result_kfold=model_selection.cross_val_score(model,features,target,cv=fold10)
         accuracy=result_kfold.mean()*100
         print("Accuracy: "+"{:.2f}".format(accuracy))
```

```
Accuracy: 59.02
```

Comparison between Decision Tree and Naive Bayes

- Comparing the results of 10 fold cross validation and classification report we find that Decision Tree performs better than naive Bayes for this dataset
- Naive Bayes gives an accuracy 59.02% by 10 fold cross validation and f1 score of 0.62. Whereas, Decision Tree gives accuracy of 95.07% by 10 fold cross validation and f1 score of 0.95.
- Naive Bayes works better on values that are discrete and that have independent features.

Scikit-learn DecisionTreeClassifier Implementation

Decision Tree Classifier source code URL https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/tree/_tree.pyx

In [ ]:

```
"""
This module gathers tree-based methods, including decision, regression and
randomized trees. Single and multi-output problems are both handled.
"""

# Authors: Gilles Louppe <g.louppe@gmail.com>
#          Peter Prettenhofer <peter.prettenhofer@gmail.com>
#          Brian Holt <bdholt1@gmail.com>
#          Noel Dawe <noel@dawe.me>
#          Satrajit Gosh <satrajit.ghosh@gmail.com>
#          Joly Arnaud <arnaud.v.joly@gmail.com>
#          Fares Hedayati <fares.hedayati@gmail.com>
#          Nelson Liu <nelson@nelsonliu.me>
#
# License: BSD 3 clause

import numbers
import warnings
import copy
from abc import ABCMeta
from abc import abstractmethod
from math import ceil

import numpy as np
from scipy.sparse import issparse

from ..base import BaseEstimator
from ..base import ClassifierMixin
from ..base import clone
from ..base import RegressorMixin
from ..base import is_classifier
from ..base import MultiOutputMixin
from ..utils import Bunch
from ..utils import check_random_state
from ..utils.validation import _check_sample_weight
from ..utils import compute_sample_weight
from ..utils.multiclass import check_classification_targets
from ..utils.validation import check_is_fitted
from ..utils.validation import _deprecate_positional_args

from ._criterion import Criterion
from ._splitter import Splitter
from ._tree import DepthFirstTreeBuilder
from ._tree import BestFirstTreeBuilder
from ._tree import Tree
from ._tree import _build_pruned_tree_ccp
from ._tree import ccp_pruning_path
from . import _tree, _splitter, _criterion

__all__ = ["DecisionTreeClassifier",
           "DecisionTreeRegressor",
           "ExtraTreeClassifier",
           "ExtraTreeRegressor"]


# =============================================================================
```

```
=
# Types and constants
# =============================================================================
=

DTYPE = _tree.DTYPE
DOUBLE = _tree.DOUBLE

CRITERIA_CLF = {"gini": _criterion.Gini,
                "entropy": _criterion.Entropy}
# TODO: Remove "mse" in version 1.2.
CRITERIA_REG = {"squared_error": _criterion.MSE,
                "mse": _criterion.MSE,
                "friedman_mse": _criterion.FriedmanMSE,
                "mae": _criterion.MAE,
                "poisson": _criterion.Poisson}

DENSE_SPLITTERS = {"best": _splitter.BestSplitter,
                   "random": _splitter.RandomSplitter}

SPARSE_SPLITTERS = {"best": _splitter.BestSparseSplitter,
                    "random": _splitter.RandomSparseSplitter}

# =============================================================================
=
# Base decision tree
# =============================================================================
=


class BaseDecisionTree(MultiOutputMixin, BaseEstimator, metaclass=ABCMeta):
    """Base class for decision trees.
    Warning: This class should not be used directly.
    Use derived classes instead.
    """

    @abstractmethod
    @_deprecate_positional_args
    def __init__(self, *,
                 criterion,
                 splitter,
                 max_depth,
                 min_samples_split,
                 min_samples_leaf,
                 min_weight_fraction_leaf,
                 max_features,
                 max_leaf_nodes,
                 random_state,
                 min_impurity_decrease,
                 min_impurity_split,
                 class_weight=None,
                 ccp_alpha=0.0):
        self.criterion = criterion
        self.splitter = splitter
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
```

```
        self.min_weight_fraction_leaf = min_weight_fraction_leaf
        self.max_features = max_features
        self.max_leaf_nodes = max_leaf_nodes
        self.random_state = random_state
        self.min_impurity_decrease = min_impurity_decrease
        self.min_impurity_split = min_impurity_split
        self.class_weight = class_weight
        self.ccp_alpha = ccp_alpha

    def get_depth(self):
        """Return the depth of the decision tree.
        The depth of a tree is the maximum distance between the root
        and any leaf.
        Returns
        -------
        self.tree_.max_depth : int
            The maximum depth of the tree.
        """
        check_is_fitted(self)
        return self.tree_.max_depth

    def get_n_leaves(self):
        """Return the number of leaves of the decision tree.
        Returns
        -------
        self.tree_.n_leaves : int
            Number of leaves.
        """
        check_is_fitted(self)
        return self.tree_.n_leaves

    def fit(self, X, y, sample_weight=None, check_input=True,
            X_idx_sorted="deprecated"):

        random_state = check_random_state(self.random_state)

        if self.ccp_alpha < 0.0:
            raise ValueError("ccp_alpha must be greater than or equal to 0")

        if check_input:
            # Need to validate separately here.
            # We can't pass multi_ouput=True because that would allow y to be
            # csr.
            check_X_params = dict(dtype=DTYPE, accept_sparse="csc")
            check_y_params = dict(ensure_2d=False, dtype=None)
            X, y = self._validate_data(X, y,
                                       validate_separately=(check_X_params,
                                                            check_y_params))
            if issparse(X):
                X.sort_indices()

                if X.indices.dtype != np.intc or X.indptr.dtype != np.intc:
                    raise ValueError("No support for np.int64 index based "
                                     "sparse matrices")

            if self.criterion == "poisson":
                if np.any(y < 0):
```

```
                    raise ValueError("Some value(s) of y are negative which i
s"
                                     " not allowed for Poisson regression.")
                if np.sum(y) <= 0:
                    raise ValueError("Sum of y is not positive which is "
                                     "necessary for Poisson regression.")

        # Determine output settings
        n_samples, self.n_features_ = X.shape
        self.n_features_in_ = self.n_features_
        is_classification = is_classifier(self)

        y = np.atleast_1d(y)
        expanded_class_weight = None

        if y.ndim == 1:
            # reshape is necessary to preserve the data contiguity against vs
            # [:, np.newaxis] that does not.
            y = np.reshape(y, (-1, 1))

        self.n_outputs_ = y.shape[1]

        if is_classification:
            check_classification_targets(y)
            y = np.copy(y)

            self.classes_ = []
            self.n_classes_ = []

            if self.class_weight is not None:
                y_original = np.copy(y)

            y_encoded = np.zeros(y.shape, dtype=int)
            for k in range(self.n_outputs_):
                classes_k, y_encoded[:, k] = np.unique(y[:, k],
                                                       return_inverse=True)
                self.classes_.append(classes_k)
                self.n_classes_.append(classes_k.shape[0])
            y = y_encoded

            if self.class_weight is not None:
                expanded_class_weight = compute_sample_weight(
                    self.class_weight, y_original)

            self.n_classes_ = np.array(self.n_classes_, dtype=np.intp)

        if getattr(y, "dtype", None) != DOUBLE or not y.flags.contiguous:
            y = np.ascontiguousarray(y, dtype=DOUBLE)

        # Check parameters
        max_depth = (np.iinfo(np.int32).max if self.max_depth is None
                     else self.max_depth)
        max_leaf_nodes = (-1 if self.max_leaf_nodes is None
                          else self.max_leaf_nodes)

        if isinstance(self.min_samples_leaf, numbers.Integral):
            if not 1 <= self.min_samples_leaf:
```

```
                raise ValueError("min_samples_leaf must be at least 1 "
                                 "or in (0, 0.5], got %s"
                                 % self.min_samples_leaf)
            min_samples_leaf = self.min_samples_leaf
        else:  # float
            if not 0. < self.min_samples_leaf <= 0.5:
                raise ValueError("min_samples_leaf must be at least 1 "
                                 "or in (0, 0.5], got %s"
                                 % self.min_samples_leaf)
            min_samples_leaf = int(ceil(self.min_samples_leaf * n_samples))

        if isinstance(self.min_samples_split, numbers.Integral):
            if not 2 <= self.min_samples_split:
                raise ValueError("min_samples_split must be an integer "
                                 "greater than 1 or a float in (0.0, 1.0]; "
                                 "got the integer %s"
                                 % self.min_samples_split)
            min_samples_split = self.min_samples_split
        else:  # float
            if not 0. < self.min_samples_split <= 1.:
                raise ValueError("min_samples_split must be an integer "
                                 "greater than 1 or a float in (0.0, 1.0]; "
                                 "got the float %s"
                                 % self.min_samples_split)
            min_samples_split = int(ceil(self.min_samples_split * n_samples))
            min_samples_split = max(2, min_samples_split)

        min_samples_split = max(min_samples_split, 2 * min_samples_leaf)

        if isinstance(self.max_features, str):
            if self.max_features == "auto":
                if is_classification:
                    max_features = max(1, int(np.sqrt(self.n_features_)))
                else:
                    max_features = self.n_features_
            elif self.max_features == "sqrt":
                max_features = max(1, int(np.sqrt(self.n_features_)))
            elif self.max_features == "log2":
                max_features = max(1, int(np.log2(self.n_features_)))
            else:
                raise ValueError("Invalid value for max_features. "
                                 "Allowed string values are 'auto', "
                                 "'sqrt' or 'log2'.")
        elif self.max_features is None:
            max_features = self.n_features_
        elif isinstance(self.max_features, numbers.Integral):
            max_features = self.max_features
        else:  # float
            if self.max_features > 0.0:
                max_features = max(1,
                                   int(self.max_features * self.n_features_))
            else:
                max_features = 0

        self.max_features_ = max_features

        if len(y) != n_samples:
```

```
            raise ValueError("Number of labels=%d does not match "
                             "number of samples=%d" % (len(y), n_samples))
        if not 0 <= self.min_weight_fraction_leaf <= 0.5:
            raise ValueError("min_weight_fraction_leaf must in [0, 0.5]")
        if max_depth <= 0:
            raise ValueError("max_depth must be greater than zero. ")
        if not (0 < max_features <= self.n_features_):
            raise ValueError("max_features must be in (0, n_features]")
        if not isinstance(max_leaf_nodes, numbers.Integral):
            raise ValueError("max_leaf_nodes must be integral number but was "
                             "%r" % max_leaf_nodes)
        if -1 < max_leaf_nodes < 2:
            raise ValueError(("max_leaf_nodes {0} must be either None "
                              "or larger than 1").format(max_leaf_nodes))

        if sample_weight is not None:
            sample_weight = _check_sample_weight(sample_weight, X, DOUBLE)

        if expanded_class_weight is not None:
            if sample_weight is not None:
                sample_weight = sample_weight * expanded_class_weight
            else:
                sample_weight = expanded_class_weight

        # Set min_weight_leaf from min_weight_fraction_leaf
        if sample_weight is None:
            min_weight_leaf = (self.min_weight_fraction_leaf *
                               n_samples)
        else:
            min_weight_leaf = (self.min_weight_fraction_leaf *
                               np.sum(sample_weight))

        min_impurity_split = self.min_impurity_split
        if min_impurity_split is not None:
            warnings.warn(
                "The min_impurity_split parameter is deprecated. Its default "
                "value has changed from 1e-7 to 0 in version 0.23, and it "
                "will be removed in 1.0 (renaming of 0.25). Use the "
                "min_impurity_decrease parameter instead.",
                FutureWarning
            )

            if min_impurity_split < 0.:
                raise ValueError("min_impurity_split must be greater than "
                                 "or equal to 0")
        else:
            min_impurity_split = 0

        if self.min_impurity_decrease < 0.:
            raise ValueError("min_impurity_decrease must be greater than "
                             "or equal to 0")

        # TODO: Remove in 1.1
        if X_idx_sorted != "deprecated":
            warnings.warn(
                "The parameter 'X_idx_sorted' is deprecated and has no "
                "effect. It will be removed in 1.1 (renaming of 0.26). You "
```

```
                "can suppress this warning by not passing any value to the "
                "'X_idx_sorted' parameter.",
                FutureWarning
            )

        # Build tree
        criterion = self.criterion
        if not isinstance(criterion, Criterion):
            if is_classification:
                criterion = CRITERIA_CLF[self.criterion](self.n_outputs_,
                                                         self.n_classes_)
            else:
                criterion = CRITERIA_REG[self.criterion](self.n_outputs_,
                                                         n_samples)
            # TODO: Remove in v1.2
            if self.criterion == "mse":
                warnings.warn(
                    "Criterion 'mse' was deprecated in v1.0 and will be "
                    "removed in version 1.2. Use `criterion='squared_error'` "
                    "which is equivalent.",
                    FutureWarning
                )
        else:
            # Make a deepcopy in case the criterion has mutable attributes tha
t
            # might be shared and modified concurrently during parallel fittin
g
            criterion = copy.deepcopy(criterion)

        SPLITTERS = SPARSE_SPLITTERS if issparse(X) else DENSE_SPLITTERS

        splitter = self.splitter
        if not isinstance(self.splitter, Splitter):
            splitter = SPLITTERS[self.splitter](criterion,
                                                self.max_features_,
                                                min_samples_leaf,
                                                min_weight_leaf,
                                                random_state)

        if is_classifier(self):
            self.tree_ = Tree(self.n_features_,
                              self.n_classes_, self.n_outputs_)
        else:
            self.tree_ = Tree(self.n_features_,
                              # TODO: tree should't need this in this case
                              np.array([1] * self.n_outputs_, dtype=np.intp),
                              self.n_outputs_)

        # Use BestFirst if max_leaf_nodes given; use DepthFirst otherwise
        if max_leaf_nodes < 0:
            builder = DepthFirstTreeBuilder(splitter, min_samples_split,
                                            min_samples_leaf,
                                            min_weight_leaf,
                                            max_depth,
                                            self.min_impurity_decrease,
                                            min_impurity_split)
        else:
```

```
            builder = BestFirstTreeBuilder(splitter, min_samples_split,
                                           min_samples_leaf,
                                           min_weight_leaf,
                                           max_depth,
                                           max_leaf_nodes,
                                           self.min_impurity_decrease,
                                           min_impurity_split)

        builder.build(self.tree_, X, y, sample_weight)

        if self.n_outputs_ == 1 and is_classifier(self):
            self.n_classes_ = self.n_classes_[0]
            self.classes_ = self.classes_[0]

        self._prune_tree()

        return self

    def _validate_X_predict(self, X, check_input):
        """Validate the training data on predict (probabilities)."""
        if check_input:
            X = self._validate_data(X, dtype=DTYPE, accept_sparse="csr",
                                    reset=False)
            if issparse(X) and (X.indices.dtype != np.intc or
                                X.indptr.dtype != np.intc):
                raise ValueError("No support for np.int64 index based "
                                 "sparse matrices")
        else:
            # The number of features is checked regardless of `check_input`
            self._check_n_features(X, reset=False)
        return X

    def predict(self, X, check_input=True):
        """Predict class or regression value for X.
        For a classification model, the predicted class for each sample in X i
s
        returned. For a regression model, the predicted value based on X is
        returned.
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csr_matrix``.
        check_input : bool, default=True
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.
        Returns
        -------
        y : array-like of shape (n_samples,) or (n_samples, n_outputs)
            The predicted classes, or the predict values.
        """
        check_is_fitted(self)
        X = self._validate_X_predict(X, check_input)
        proba = self.tree_.predict(X)
        n_samples = X.shape[0]
```

```
        # Classification
        if is_classifier(self):
            if self.n_outputs_ == 1:
                return self.classes_.take(np.argmax(proba, axis=1), axis=0)

            else:
                class_type = self.classes_[0].dtype
                predictions = np.zeros((n_samples, self.n_outputs_),
                                       dtype=class_type)
                for k in range(self.n_outputs_):
                    predictions[:, k] = self.classes_[k].take(
                        np.argmax(proba[:, k], axis=1),
                        axis=0)

                return predictions

        # Regression
        else:
            if self.n_outputs_ == 1:
                return proba[:, 0]

            else:
                return proba[:, :, 0]

    def apply(self, X, check_input=True):
        """Return the index of the leaf that each sample is predicted as.
        .. versionadded:: 0.17
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csr_matrix``.
        check_input : bool, default=True
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.
        Returns
        -------
        X_leaves : array-like of shape (n_samples,)
            For each datapoint x in X, return the index of the leaf x
            ends up in. Leaves are numbered within
            ``[0; self.tree_.node_count)``, possibly with gaps in the
            numbering.
        """
        check_is_fitted(self)
        X = self._validate_X_predict(X, check_input)
        return self.tree_.apply(X)

    def decision_path(self, X, check_input=True):
        """Return the decision path in the tree.
        .. versionadded:: 0.18
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csr_matrix``.
```

```
        check_input : bool, default=True
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.
        Returns
        -------
        indicator : sparse matrix of shape (n_samples, n_nodes)
            Return a node indicator CSR matrix where non zero elements
            indicates that the samples goes through the nodes.
        """
        X = self._validate_X_predict(X, check_input)
        return self.tree_.decision_path(X)

    def _prune_tree(self):
        """Prune tree using Minimal Cost-Complexity Pruning."""
        check_is_fitted(self)

        if self.ccp_alpha < 0.0:
            raise ValueError("ccp_alpha must be greater than or equal to 0")

        if self.ccp_alpha == 0.0:
            return

        # build pruned tree
        if is_classifier(self):
            n_classes = np.atleast_1d(self.n_classes_)
            pruned_tree = Tree(self.n_features_, n_classes, self.n_outputs_)
        else:
            pruned_tree = Tree(self.n_features_,
                               # TODO: the tree shouldn't need this param
                               np.array([1] * self.n_outputs_, dtype=np.intp),
                               self.n_outputs_)
        _build_pruned_tree_ccp(pruned_tree, self.tree_, self.ccp_alpha)

        self.tree_ = pruned_tree

    def cost_complexity_pruning_path(self, X, y, sample_weight=None):
        """Compute the pruning path during Minimal Cost-Complexity Pruning.
        See :ref:`minimal_cost_complexity_pruning` for details on the pruning
        process.
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The training input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csc_matrix``.
        y : array-like of shape (n_samples,) or (n_samples, n_outputs)
            The target values (class labels) as integers or strings.
        sample_weight : array-like of shape (n_samples,), default=None
            Sample weights. If None, then samples are equally weighted. Splits
            that would create child nodes with net zero or negative weight are
            ignored while searching for a split in each node. Splits are also
            ignored if they would result in any single class carrying a
            negative weight in either child node.
        Returns
        -------
        ccp_path : :class:`~sklearn.utils.Bunch`
            Dictionary-like object, with the following attributes.
```

```
            ccp_alphas : ndarray
                Effective alphas of subtree during pruning.
            impurities : ndarray
                Sum of the impurities of the subtree leaves for the
                corresponding alpha value in ``ccp_alphas``.
        """
        est = clone(self).set_params(ccp_alpha=0.0)
        est.fit(X, y, sample_weight=sample_weight)
        return Bunch(**ccp_pruning_path(est.tree_))

    @property
    def feature_importances_(self):
        """Return the feature importances.
        The importance of a feature is computed as the (normalized) total
        reduction of the criterion brought by that feature.
        It is also known as the Gini importance.
        Warning: impurity-based feature importances can be misleading for
        high cardinality features (many unique values). See
        :func:`sklearn.inspection.permutation_importance` as an alternative.
        Returns
        -------
        feature_importances_ : ndarray of shape (n_features,)
            Normalized total reduction of criteria by feature
            (Gini importance).
        """
        check_is_fitted(self)

        return self.tree_.compute_feature_importances()


# =============================================================================
=
# Public estimators
# =============================================================================
=

class DecisionTreeClassifier(ClassifierMixin, BaseDecisionTree):
    """A decision tree classifier.
    Read more in the :ref:`User Guide <tree>`.
    Parameters
    ----------
    criterion : {"gini", "entropy"}, default="gini"
        The function to measure the quality of a split. Supported criteria are
        "gini" for the Gini impurity and "entropy" for the information gain.
    splitter : {"best", "random"}, default="best"
        The strategy used to choose the split at each node. Supported
        strategies are "best" to choose the best split and "random" to choose
        the best random split.
    max_depth : int, default=None
        The maximum depth of the tree. If None, then nodes are expanded until
        all leaves are pure or until all leaves contain less than
        min_samples_split samples.
    min_samples_split : int or float, default=2
        The minimum number of samples required to split an internal node:
        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
```

```
          number of samples for each split.
       .. versionchanged:: 0.18
          Added float values for fractions.
   min_samples_leaf : int or float, default=1
       The minimum number of samples required to be at a leaf node.
       A split point at any depth will only be considered if it leaves at
       least ``min_samples_leaf`` training samples in each of the left and
       right branches.  This may have the effect of smoothing the model,
       especially in regression.
       - If int, then consider `min_samples_leaf` as the minimum number.
       - If float, then `min_samples_leaf` is a fraction and
         `ceil(min_samples_leaf * n_samples)` are the minimum
         number of samples for each node.
       .. versionchanged:: 0.18
          Added float values for fractions.
   min_weight_fraction_leaf : float, default=0.0
       The minimum weighted fraction of the sum total of weights (of all
       the input samples) required to be at a leaf node. Samples have
       equal weight when sample_weight is not provided.
   max_features : int, float or {"auto", "sqrt", "log2"}, default=None
       The number of features to consider when looking for the best split:
           - If int, then consider `max_features` features at each split.
           - If float, then `max_features` is a fraction and
             `int(max_features * n_features)` features are considered at each
             split.
           - If "auto", then `max_features=sqrt(n_features)`.
           - If "sqrt", then `max_features=sqrt(n_features)`.
           - If "log2", then `max_features=log2(n_features)`.
           - If None, then `max_features=n_features`.
       Note: the search for a split does not stop until at least one
       valid partition of the node samples is found, even if it requires to
       effectively inspect more than ``max_features`` features.
   random_state : int, RandomState instance or None, default=None
       Controls the randomness of the estimator. The features are always
       randomly permuted at each split, even if ``splitter`` is set to
       ``"best"``. When ``max_features < n_features``, the algorithm will
       select ``max_features`` at random at each split before finding the bes
t
       split among them. But the best found split may vary across different
       runs, even if ``max_features=n_features``. That is the case, if the
       improvement of the criterion is identical for several splits and one
       split has to be selected at random. To obtain a deterministic behaviou
r
       during fitting, ``random_state`` has to be fixed to an integer.
       See :term:`Glossary <random_state>` for details.
   max_leaf_nodes : int, default=None
       Grow a tree with ``max_leaf_nodes`` in best-first fashion.
       Best nodes are defined as relative reduction in impurity.
       If None then unlimited number of leaf nodes.
   min_impurity_decrease : float, default=0.0
       A node will be split if this split induces a decrease of the impurity
       greater than or equal to this value.
       The weighted impurity decrease equation is the following::
           N_t / N * (impurity - N_t_R / N_t * right_impurity
                               - N_t_L / N_t * left_impurity)
       where ``N`` is the total number of samples, ``N_t`` is the number of
       samples at the current node, ``N_t_L`` is the number of samples in the
```

```
    left child, and ``N_t_R`` is the number of samples in the right child.
    ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
    if ``sample_weight`` is passed.
    .. versionadded:: 0.19
min_impurity_split : float, default=0
    Threshold for early stopping in tree growth. A node will split
    if its impurity is above the threshold, otherwise it is a leaf.
    .. deprecated:: 0.19
       ``min_impurity_split`` has been deprecated in favor of
       ``min_impurity_decrease`` in 0.19. The default value of
       ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
       will be removed in 1.0 (renaming of 0.25).
       Use ``min_impurity_decrease`` instead.
class_weight : dict, list of dict or "balanced", default=None
    Weights associated with classes in the form ``{class_label: weight}``.
    If None, all classes are supposed to have weight one. For
    multi-output problems, a list of dicts can be provided in the same
    order as the columns of y.
    Note that for multioutput (including multilabel) weights should be
    defined for each class of every column in its own dict. For example,
    for four-class multilabel classification weights should be
    [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
    [{1:1}, {2:5}, {3:1}, {4:1}].
    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as ``n_samples / (n_classes * np.bincount(y))``
    For multi-output, the weights of each column of y will be multiplied.
    Note that these weights will be multiplied with sample_weight (passed
    through the fit method) if sample_weight is specified.
ccp_alpha : non-negative float, default=0.0
    Complexity parameter used for Minimal Cost-Complexity Pruning. The
    subtree with the largest cost complexity that is smaller than
    ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
    :ref:`minimal_cost_complexity_pruning` for details.
    .. versionadded:: 0.22
Attributes
----------
classes_ : ndarray of shape (n_classes,) or list of ndarray
    The classes labels (single output problem),
    or a list of arrays of class labels (multi-output problem).
feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature.  It is also
    known as the Gini importance [4]_.
    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.
max_features_ : int
    The inferred value of max_features.
n_classes_ : int or list of int
    The number of classes (for single output problems),
    or a list containing the number of classes for each
    output (for multi-output problems).
n_features_ : int
    The number of features when ``fit`` is performed.
```

```
    n_outputs_ : int
        The number of outputs when ``fit`` is performed.
    tree_ : Tree instance
        The underlying Tree object. Please refer to
        ``help(sklearn.tree._tree.Tree)`` for attributes of Tree object and
        :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py`
        for basic usage of these attributes.
    See Also
    --------
    DecisionTreeRegressor : A decision tree regressor.
    Notes
    -----
    The default values for the parameters controlling the size of the trees
    (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
    unpruned trees which can potentially be very large on some data sets. To
    reduce memory consumption, the complexity and size of the trees should be
    controlled by setting those parameter values.
    The :meth:`predict` method operates using the :func:`numpy.argmax`
    function on the outputs of :meth:`predict_proba`. This means that in
    case the highest predicted probabilities are tied, the classifier will
    predict the tied class with the lowest index in :term:`classes_`.
    References
    ----------
    .. [1] https://en.wikipedia.org/wiki/Decision_tree_learning
    .. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification
           and Regression Trees", Wadsworth, Belmont, CA, 1984.
    .. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical
           Learning", Springer, 2009.
    .. [4] L. Breiman, and A. Cutler, "Random Forests",
           https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm
    Examples
    --------
    >>> from sklearn.datasets import load_iris
    >>> from sklearn.model_selection import cross_val_score
    >>> from sklearn.tree import DecisionTreeClassifier
    >>> clf = DecisionTreeClassifier(random_state=0)
    >>> iris = load_iris()
    >>> cross_val_score(clf, iris.data, iris.target, cv=10)
    ...                             # doctest: +SKIP
    ...
    array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
            0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
    """
    @_deprecate_positional_args
    def __init__(self, *,
                 criterion="gini",
                 splitter="best",
                 max_depth=None,
                 min_samples_split=2,
                 min_samples_leaf=1,
                 min_weight_fraction_leaf=0.,
                 max_features=None,
                 random_state=None,
                 max_leaf_nodes=None,
                 min_impurity_decrease=0.,
                 min_impurity_split=None,
                 class_weight=None,
```

```
                 ccp_alpha=0.0):
        super().__init__(
            criterion=criterion,
            splitter=splitter,
            max_depth=max_depth,
            min_samples_split=min_samples_split,
            min_samples_leaf=min_samples_leaf,
            min_weight_fraction_leaf=min_weight_fraction_leaf,
            max_features=max_features,
            max_leaf_nodes=max_leaf_nodes,
            class_weight=class_weight,
            random_state=random_state,
            min_impurity_decrease=min_impurity_decrease,
            min_impurity_split=min_impurity_split,
            ccp_alpha=ccp_alpha)

    def fit(self, X, y, sample_weight=None, check_input=True,
            X_idx_sorted="deprecated"):
        """Build a decision tree classifier from the training set (X, y).
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The training input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csc_matrix``.
        y : array-like of shape (n_samples,) or (n_samples, n_outputs)
            The target values (class labels) as integers or strings.
        sample_weight : array-like of shape (n_samples,), default=None
            Sample weights. If None, then samples are equally weighted. Splits
            that would create child nodes with net zero or negative weight are
            ignored while searching for a split in each node. Splits are also
            ignored if they would result in any single class carrying a
            negative weight in either child node.
        check_input : bool, default=True
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.
        X_idx_sorted : deprecated, default="deprecated"
            This parameter is deprecated and has no effect.
            It will be removed in 1.1 (renaming of 0.26).
            .. deprecated :: 0.24
        Returns
        -------
        self : DecisionTreeClassifier
            Fitted estimator.
        """

        super().fit(
            X, y,
            sample_weight=sample_weight,
            check_input=check_input,
            X_idx_sorted=X_idx_sorted)
        return self

    def predict_proba(self, X, check_input=True):
        """Predict class probabilities of the input samples X.
        The predicted class probability is the fraction of samples of the same
        class in a leaf.
```

```
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csr_matrix``.
        check_input : bool, default=True
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.
        Returns
        -------
        proba : ndarray of shape (n_samples, n_classes) or list of n_outputs \
            such arrays if n_outputs > 1
            The class probabilities of the input samples. The order of the
            classes corresponds to that in the attribute :term:`classes_`.
        """
        check_is_fitted(self)
        X = self._validate_X_predict(X, check_input)
        proba = self.tree_.predict(X)

        if self.n_outputs_ == 1:
            proba = proba[:, :self.n_classes_]
            normalizer = proba.sum(axis=1)[:, np.newaxis]
            normalizer[normalizer == 0.0] = 1.0
            proba /= normalizer

            return proba

        else:
            all_proba = []

            for k in range(self.n_outputs_):
                proba_k = proba[:, k, :self.n_classes_[k]]
                normalizer = proba_k.sum(axis=1)[:, np.newaxis]
                normalizer[normalizer == 0.0] = 1.0
                proba_k /= normalizer
                all_proba.append(proba_k)

            return all_proba

    def predict_log_proba(self, X):
        """Predict class log-probabilities of the input samples X.
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, it will be converted to
            ``dtype=np.float32`` and if a sparse matrix is provided
            to a sparse ``csr_matrix``.
        Returns
        -------
        proba : ndarray of shape (n_samples, n_classes) or list of n_outputs \
            such arrays if n_outputs > 1
            The class log-probabilities of the input samples. The order of the
            classes corresponds to that in the attribute :term:`classes_`.
        """
        proba = self.predict_proba(X)
```

```
        if self.n_outputs_ == 1:
            return np.log(proba)

        else:
            for k in range(self.n_outputs_):
                proba[k] = np.log(proba[k])

            return proba
```

Scikit-learn Naive Bayes Classification

Naive Bayes source code URL https://github.com/scikit-learn/scikit-learn/blob/114616d9f6ce9eba7c1aacd3d4a254f868010e25/sklearn/naive_bayes.py

In [ ]:
```
# -*- coding: utf-8 -*-

"""
The :mod:`sklearn.naive_bayes` module implements Naive Bayes algorithms. These
are supervised learning methods based on applying Bayes' theorem with strong
(naive) feature independence assumptions.
"""

# Author: Vincent Michel <vincent.michel@inria.fr>
#         Minor fixes by Fabian Pedregosa
#         Amit Aides <amitibo@tx.technion.ac.il>
#         Yehuda Finkelstein <yehudaf@tx.technion.ac.il>
#         Lars Buitinck
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#         (parts based on earlier work by Mathieu Blondel)
#
# License: BSD 3 clause
import warnings

from abc import ABCMeta, abstractmethod


import numpy as np
from scipy.special import logsumexp

from .base import BaseEstimator, ClassifierMixin
from .preprocessing import binarize
from .preprocessing import LabelBinarizer
from .preprocessing import label_binarize
from .utils import deprecated
from .utils.extmath import safe_sparse_dot
from .utils.multiclass import _check_partial_fit_first_call
from .utils.validation import check_is_fitted, check_non_negative
from .utils.validation import _check_sample_weight
from .utils.validation import _deprecate_positional_args
```

In [ ]:

```
__all__ = ['BernoulliNB', 'GaussianNB', 'MultinomialNB', 'ComplementNB',
           'CategoricalNB']


class _BaseNB(ClassifierMixin, BaseEstimator, metaclass=ABCMeta):
    """Abstract base class for naive Bayes estimators"""

    @abstractmethod
    def _joint_log_likelihood(self, X):
        """Compute the unnormalized posterior log probability of X
        I.e. ``log P(c) + log P(x|c)`` for all rows x of X, as an array-like o
f
        shape (n_classes, n_samples).
        Input is passed to _joint_log_likelihood as-is by predict,
        predict_proba and predict_log_proba.
        """

    @abstractmethod
    def _check_X(self, X):
        """To be overridden in subclasses with the actual checks.
        Only used in predict* methods.
        """

    def predict(self, X):
        """
        Perform classification on an array of test vectors X.
        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
        Returns
        -------
        C : ndarray of shape (n_samples,)
            Predicted target values for X
        """
        check_is_fitted(self)
        X = self._check_X(X)
        jll = self._joint_log_likelihood(X)
        return self.classes_[np.argmax(jll, axis=1)]

    def predict_log_proba(self, X):
        """
        Return log-probability estimates for the test vector X.
        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
        Returns
        -------
        C : array-like of shape (n_samples, n_classes)
            Returns the log-probability of the samples for each class in
            the model. The columns correspond to the classes in sorted
            order, as they appear in the attribute :term:`classes_`.
        """
        check_is_fitted(self)
        X = self._check_X(X)
        jll = self._joint_log_likelihood(X)
        # normalize by P(x) = P(f_1, ..., f_n)
```

```
        log_prob_x = logsumexp(jll, axis=1)
        return jll - np.atleast_2d(log_prob_x).T

    def predict_proba(self, X):
        """
        Return probability estimates for the test vector X.
        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
        Returns
        -------
        C : array-like of shape (n_samples, n_classes)
            Returns the probability of the samples for each class in
            the model. The columns correspond to the classes in sorted
            order, as they appear in the attribute :term:`classes_`.
        """
        return np.exp(self.predict_log_proba(X))




_ALPHA_MIN = 1e-10


class _BaseDiscreteNB(_BaseNB):
    """Abstract base class for naive Bayes on discrete/categorical data
    Any estimator based on this class should provide:
    __init__
    _joint_log_likelihood(X) as per _BaseNB
    """

    def _check_X(self, X):
        """Validate X, used only in predict* methods."""
        return self._validate_data(X, accept_sparse='csr', reset=False)

    def _check_X_y(self, X, y, reset=True):
        """Validate X and y in fit methods."""
        return self._validate_data(X, y, accept_sparse='csr', reset=reset)

    def _update_class_log_prior(self, class_prior=None):
        n_classes = len(self.classes_)
        if class_prior is not None:
            if len(class_prior) != n_classes:
                raise ValueError("Number of priors must match number of"
                                 " classes.")
            self.class_log_prior_ = np.log(class_prior)
        elif self.fit_prior:
            with warnings.catch_warnings():
                # silence the warning when count is 0 because class was not ye
t
                # observed
                warnings.simplefilter("ignore", RuntimeWarning)
                log_class_count = np.log(self.class_count_)

            # empirical prior, with sample_weight taken into account
            self.class_log_prior_ = (log_class_count -
                                     np.log(self.class_count_.sum()))
```

```
        else:
            self.class_log_prior_ = np.full(n_classes, -np.log(n_classes))

    def _check_alpha(self):
        if np.min(self.alpha) < 0:
            raise ValueError('Smoothing parameter alpha = %.1e. '
                             'alpha should be > 0.' % np.min(self.alpha))
        if isinstance(self.alpha, np.ndarray):
            if not self.alpha.shape[0] == self.n_features_in_:
                raise ValueError("alpha should be a scalar or a numpy array "
                                 "with shape [n_features]")
        if np.min(self.alpha) < _ALPHA_MIN:
            warnings.warn('alpha too small will result in numeric errors, '
                          'setting alpha = %.1e' % _ALPHA_MIN)
            return np.maximum(self.alpha, _ALPHA_MIN)
        return self.alpha

    def partial_fit(self, X, y, classes=None, sample_weight=None):
        """Incremental fit on a batch of samples.
        This method is expected to be called several times consecutively
        on different chunks of a dataset so as to implement out-of-core
        or online learning.
        This is especially useful when the whole dataset is too big to fit in
        memory at once.
        This method has some performance overhead hence it is better to call
        partial_fit on chunks of data that are as large as possible
        (as long as fitting in the memory budget) to hide the overhead.
        Parameters
        ----------
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like of shape (n_samples,)
            Target values.
        classes : array-like of shape (n_classes,), default=None
            List of all the classes that can possibly appear in the y vector.
            Must be provided at the first call to partial_fit, can be omitted
            in subsequent calls.
        sample_weight : array-like of shape (n_samples,), default=None
            Weights applied to individual samples (1. for unweighted).
        Returns
        -------
        self : object
        """
        first_call = not hasattr(self, "classes_")
        X, y = self._check_X_y(X, y, reset=first_call)
        _, n_features = X.shape

        if _check_partial_fit_first_call(self, classes):
            # This is the first call to partial_fit:
            # initialize various cumulative counters
            n_classes = len(classes)
            self._init_counters(n_classes, n_features)

        Y = label_binarize(y, classes=self.classes_)
        if Y.shape[1] == 1:
            if len(self.classes_) == 2:
```

```
            Y = np.concatenate((1 - Y, Y), axis=1)
        else:    # degenerate case: just one class
            Y = np.ones_like(Y)

    if X.shape[0] != Y.shape[0]:
        msg = "X.shape[0]=%d and y.shape[0]=%d are incompatible."
        raise ValueError(msg % (X.shape[0], y.shape[0]))

    # label_binarize() returns arrays with dtype=np.int64.
    # We convert it to np.float64 to support sample_weight consistently
    Y = Y.astype(np.float64, copy=False)
    if sample_weight is not None:
        sample_weight = _check_sample_weight(sample_weight, X)
        sample_weight = np.atleast_2d(sample_weight)
        Y *= sample_weight.T

    class_prior = self.class_prior

    # Count raw events from data before updating the class log prior
    # and feature log probas
    self._count(X, Y)

    # XXX: OPTIM: we could introduce a public finalization method to
    # be called by the user explicitly just once after several consecutive
    # calls to partial_fit and prior any call to predict[_[log_]proba]
    # to avoid computing the smooth log probas at each call to partial fit
    alpha = self._check_alpha()
    self._update_feature_log_prob(alpha)
    self._update_class_log_prior(class_prior=class_prior)
    return self

def fit(self, X, y, sample_weight=None):
    """Fit Naive Bayes classifier according to X, y
    Parameters
    ----------
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.
    y : array-like of shape (n_samples,)
        Target values.
    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).
    Returns
    -------
    self : object
    """
    X, y = self._check_X_y(X, y)
    _, n_features = X.shape

    labelbin = LabelBinarizer()
    Y = labelbin.fit_transform(y)
    self.classes_ = labelbin.classes_
    if Y.shape[1] == 1:
        if len(self.classes_) == 2:
            Y = np.concatenate((1 - Y, Y), axis=1)
        else:    # degenerate case: just one class
            Y = np.ones_like(Y)
```

```
        # LabelBinarizer().fit_transform() returns arrays with dtype=np.int64.
        # We convert it to np.float64 to support sample_weight consistently;
        # this means we also don't have to cast X to floating point
        if sample_weight is not None:
            Y = Y.astype(np.float64, copy=False)
            sample_weight = _check_sample_weight(sample_weight, X)
            sample_weight = np.atleast_2d(sample_weight)
            Y *= sample_weight.T

        class_prior = self.class_prior

        # Count raw events from data before updating the class log prior
        # and feature log probas
        n_classes = Y.shape[1]
        self._init_counters(n_classes, n_features)
        self._count(X, Y)
        alpha = self._check_alpha()
        self._update_feature_log_prob(alpha)
        self._update_class_log_prior(class_prior=class_prior)
        return self

    def _init_counters(self, n_classes, n_features):
        self.class_count_ = np.zeros(n_classes, dtype=np.float64)
        self.feature_count_ = np.zeros((n_classes, n_features),
                                       dtype=np.float64)

    # mypy error: Decorated property not supported
    @deprecated("Attribute coef_ was deprecated in "  # type: ignore
                "version 0.24 and will be removed in 1.1 (renaming of 0.26).")
    @property
    def coef_(self):
        return (self.feature_log_prob_[1:]
                if len(self.classes_) == 2 else self.feature_log_prob_)

    # mypy error: Decorated property not supported
    @deprecated("Attribute intercept_ was deprecated in "  # type: ignore
                "version 0.24 and will be removed in 1.1 (renaming of 0.26).")
    @property
    def intercept_(self):
        return (self.class_log_prior_[1:]
                if len(self.classes_) == 2 else self.class_log_prior_)

    def _more_tags(self):
        return {'poor_score': True}

    # TODO: Remove in 1.2
    # mypy error: Decorated property not supported
    @deprecated(  # type: ignore
        "Attribute n_features_ was deprecated in version 1.0 and will be "
        "removed in 1.2. Use 'n_features_in_' instead."
    )
    @property
    def n_features_(self):
        return self.n_features_in_
```

```
class MultinomialNB(_BaseDiscreteNB):
    """
    Naive Bayes classifier for multinomial models
    The multinomial Naive Bayes classifier is suitable for classification with
    discrete features (e.g., word counts for text classification). The
    multinomial distribution normally requires integer feature counts. Howeve
r,
    in practice, fractional counts such as tf-idf may also work.
    Read more in the :ref:`User Guide <multinomial_naive_bayes>`.
    Parameters
    ----------
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).
    fit_prior : bool, default=True
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.
    class_prior : array-like of shape (n_classes,), default=None
        Prior probabilities of the classes. If specified the priors are not
        adjusted according to the data.
    Attributes
    ----------
    class_count_ : ndarray of shape (n_classes,)
        Number of samples encountered for each class during fitting. This
        value is weighted by the sample weight when provided.
    class_log_prior_ : ndarray of shape (n_classes,)
        Smoothed empirical log probability for each class.
    classes_ : ndarray of shape (n_classes,)
        Class labels known to the classifier
    coef_ : ndarray of shape (n_classes, n_features)
        Mirrors ``feature_log_prob_`` for interpreting `MultinomialNB`
        as a linear model.
        .. deprecated:: 0.24
            ``coef_`` is deprecated in 0.24 and will be removed in 1.1
            (renaming of 0.26).
    feature_count_ : ndarray of shape (n_classes, n_features)
        Number of samples encountered for each (class, feature)
        during fitting. This value is weighted by the sample weight when
        provided.
    feature_log_prob_ : ndarray of shape (n_classes, n_features)
        Empirical log probability of features
        given a class, ``P(x_i|y)``.
    intercept_ : ndarray of shape (n_classes,)
        Mirrors ``class_log_prior_`` for interpreting `MultinomialNB`
        as a linear model.
        .. deprecated:: 0.24
            ``intercept_`` is deprecated in 0.24 and will be removed in 1.1
            (renaming of 0.26).
    n_features_ : int
        Number of features of each sample.
        .. deprecated:: 1.0
            Attribute `n_features_` was deprecated in version 1.0 and will be
            removed in 1.2. Use `n_features_in_` instead.
    Examples
    --------
    >>> import numpy as np
    >>> rng = np.random.RandomState(1)
```

```
    >>> X = rng.randint(5, size=(6, 100))
    >>> y = np.array([1, 2, 3, 4, 5, 6])
    >>> from sklearn.naive_bayes import MultinomialNB
    >>> clf = MultinomialNB()
    >>> clf.fit(X, y)
    MultinomialNB()
    >>> print(clf.predict(X[2:3]))
    [3]
    Notes
    -----
    For the rationale behind the names `coef_` and `intercept_`, i.e.
    naive Bayes as a linear classifier, see J. Rennie et al. (2003),
    Tackling the poor assumptions of naive Bayes text classifiers, ICML.
    References
    ----------
    C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to
    Information Retrieval. Cambridge University Press, pp. 234-265.
    https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classif
ication-1.html
    """

    @_deprecate_positional_args
    def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None):
        self.alpha = alpha
        self.fit_prior = fit_prior
        self.class_prior = class_prior

    def _more_tags(self):
        return {'requires_positive_X': True}

    def _count(self, X, Y):
        """Count and smooth feature occurrences."""
        check_non_negative(X, "MultinomialNB (input X)")
        self.feature_count_ += safe_sparse_dot(Y.T, X)
        self.class_count_ += Y.sum(axis=0)

    def _update_feature_log_prob(self, alpha):
        """Apply smoothing to raw counts and recompute log probabilities"""
        smoothed_fc = self.feature_count_ + alpha
        smoothed_cc = smoothed_fc.sum(axis=1)

        self.feature_log_prob_ = (np.log(smoothed_fc) -
                                  np.log(smoothed_cc.reshape(-1, 1)))

    def _joint_log_likelihood(self, X):
        """Calculate the posterior log probability of the samples X"""
        return (safe_sparse_dot(X, self.feature_log_prob_.T) +
                self.class_log_prior_)
```

Links to libraries imported in this project

The following are the links and source codes of the libraries that were imported and used in the project.

Preprocessing https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/preprocessing/_label.py

In [ ]:
```
# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
#          Joel Nothman <joel.nothman@gmail.com>
#          Hamzeh Alsalhi <ha258@cornell.edu>
# License: BSD 3 clause

from collections import defaultdict
import itertools
import array
import warnings

import numpy as np
import scipy.sparse as sp

from ..base import BaseEstimator, TransformerMixin

from ..utils.sparsefuncs import min_max_axis
from ..utils import column_or_1d
from ..utils.validation import check_array
from ..utils.validation import check_is_fitted
from ..utils.validation import _num_samples
from ..utils.validation import _deprecate_positional_args
from ..utils.multiclass import unique_labels
from ..utils.multiclass import type_of_target
from ..utils._encode import _encode, _unique


__all__ = [
    'label_binarize',
    'LabelBinarizer',
    'LabelEncoder',
    'MultiLabelBinarizer',
]
```

In [ ]:

```
class LabelEncoder(TransformerMixin, BaseEstimator):
    """Encode target labels with value between 0 and n_classes-1.
    This transformer should be used to encode target values, *i.e.* `y`, and
    not the input `X`.
    Read more in the :ref:`User Guide <preprocessing_targets>`.
    .. versionadded:: 0.12
    Attributes
    ----------
    classes_ : ndarray of shape (n_classes,)
        Holds the label for each class.
    Examples
    --------
    `LabelEncoder` can be used to normalize labels.
    >>> from sklearn import preprocessing
    >>> le = preprocessing.LabelEncoder()
    >>> le.fit([1, 2, 2, 6])
    LabelEncoder()
    >>> le.classes_
    array([1, 2, 6])
    >>> le.transform([1, 1, 2, 6])
    array([0, 0, 1, 2]...)
    >>> le.inverse_transform([0, 0, 1, 2])
    array([1, 1, 2, 6])
    It can also be used to transform non-numerical labels (as long as they are
    hashable and comparable) to numerical labels.
    >>> le = preprocessing.LabelEncoder()
    >>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
    LabelEncoder()
    >>> list(le.classes_)
    ['amsterdam', 'paris', 'tokyo']
    >>> le.transform(["tokyo", "tokyo", "paris"])
    array([2, 2, 1]...)
    >>> list(le.inverse_transform([2, 2, 1]))
    ['tokyo', 'tokyo', 'paris']
    See Also
    --------
    OrdinalEncoder : Encode categorical features using an ordinal encoding
        scheme.
    OneHotEncoder : Encode categorical features as a one-hot numeric array.
    """

    def fit(self, y):
        """Fit label encoder.
        Parameters
        ----------
        y : array-like of shape (n_samples,)
            Target values.
        Returns
        -------
        self : returns an instance of self.
        """
        y = column_or_1d(y, warn=True)
        self.classes_ = _unique(y)
        return self

    def fit_transform(self, y):
```

```
        """Fit label encoder and return encoded labels.
        Parameters
        ----------
        y : array-like of shape (n_samples,)
            Target values.
        Returns
        -------
        y : array-like of shape (n_samples,)
        """
        y = column_or_1d(y, warn=True)
        self.classes_, y = _unique(y, return_inverse=True)
        return y

    def transform(self, y):
        """Transform labels to normalized encoding.
        Parameters
        ----------
        y : array-like of shape (n_samples,)
            Target values.
        Returns
        -------
        y : array-like of shape (n_samples,)
        """
        check_is_fitted(self)
        y = column_or_1d(y, warn=True)
        # transform of empty array is empty array
        if _num_samples(y) == 0:
            return np.array([])

        return _encode(y, uniques=self.classes_)

    def inverse_transform(self, y):
        """Transform labels back to original encoding.
        Parameters
        ----------
        y : ndarray of shape (n_samples,)
            Target values.
        Returns
        -------
        y : ndarray of shape (n_samples,)
        """
        check_is_fitted(self)
        y = column_or_1d(y, warn=True)
        # inverse transform of empty array is empty array
        if _num_samples(y) == 0:
            return np.array([])

        diff = np.setdiff1d(y, np.arange(len(self.classes_)))
        if len(diff):
            raise ValueError(
                    "y contains previously unseen labels: %s" % str(diff))
        y = np.asarray(y)
        return self.classes_[y]

    def _more_tags(self):
        return {'X_types': ['1dlabels']}
```

Classification Metrics https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/metrics/_classification.py

In [ ]:

```
"""Metrics to assess performance on classification task given class predictio
n.
Functions named as ``*_score`` return a scalar value to maximize: the higher
the better.
Function named as ``*_error`` or ``*_loss`` return a scalar value to minimize:
the lower the better.
"""

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Arnaud Joly <a.joly@ulg.ac.be>
#          Jochen Wersdorfer <jochen@wersdoerfer.de>
#          Lars Buitinck
#          Joel Nothman <joel.nothman@gmail.com>
#          Noel Dawe <noel@dawe.me>
#          Jatin Shah <jatindshah@gmail.com>
#          Saurabh Jha <saurabh.jhaa@gmail.com>
#          Bernardo Stein <bernardovstein@gmail.com>
#          Shangwu Yao <shangwuyao@gmail.com>
# License: BSD 3 clause


import warnings
import numpy as np

from scipy.sparse import coo_matrix
from scipy.sparse import csr_matrix

from ..preprocessing import LabelBinarizer
from ..preprocessing import LabelEncoder
from ..utils import assert_all_finite
from ..utils import check_array
from ..utils import check_consistent_length
from ..utils import column_or_1d
from ..utils.multiclass import unique_labels
from ..utils.multiclass import type_of_target
from ..utils.validation import _num_samples
from ..utils.validation import _deprecate_positional_args
from ..utils.sparsefuncs import count_nonzero
from ..exceptions import UndefinedMetricWarning

from ._base import _check_pos_label_consistency
```

In [ ]:

```
def _check_zero_division(zero_division):
    if isinstance(zero_division, str) and zero_division == "warn":
        return
    elif isinstance(zero_division, (int, float)) and zero_division in [0, 1]:
        return
    raise ValueError('Got zero_division={0}.'
                     ' Must be one of ["warn", 0, 1]'.format(zero_division))


def _check_targets(y_true, y_pred):
    """Check that y_true and y_pred belong to the same classification task.
    This converts multiclass or binary types to a common shape, and raises a
    ValueError for a mix of multilabel and multiclass targets, a mix of
    multilabel formats, for the presence of continuous-valued or multioutput
    targets, or for targets of different lengths.
    Column vectors are squeezed to 1d, while multilabel formats are returned
    as CSR sparse label indicators.
    Parameters
    ----------
    y_true : array-like
    y_pred : array-like
    Returns
    -------
    type_true : one of {'multilabel-indicator', 'multiclass', 'binary'}
        The type of the true target data, as output by
        ``utils.multiclass.type_of_target``.
    y_true : array or indicator matrix
    y_pred : array or indicator matrix
    """
    check_consistent_length(y_true, y_pred)
    type_true = type_of_target(y_true)
    type_pred = type_of_target(y_pred)

    y_type = {type_true, type_pred}
    if y_type == {"binary", "multiclass"}:
        y_type = {"multiclass"}

    if len(y_type) > 1:
        raise ValueError("Classification metrics can't handle a mix of {0} "
                         "and {1} targets".format(type_true, type_pred))

    # We can't have more than one value on y_type => The set is no more needed
    y_type = y_type.pop()

    # No metrics support "multiclass-multioutput" format
    if (y_type not in ["binary", "multiclass", "multilabel-indicator"]):
        raise ValueError("{0} is not supported".format(y_type))

    if y_type in ["binary", "multiclass"]:
        y_true = column_or_1d(y_true)
        y_pred = column_or_1d(y_pred)
        if y_type == "binary":
            try:
                unique_values = np.union1d(y_true, y_pred)
            except TypeError as e:
                # We expect y_true and y_pred to be of the same data type.
```

```
                # If `y_true` was provided to the classifier as strings,
                # `y_pred` given by the classifier will also be encoded with
                # strings. So we raise a meaningful error
                raise TypeError(
                    f"Labels in y_true and y_pred should be of the same type.
 "
                    f"Got y_true={np.unique(y_true)} and "
                    f"y_pred={np.unique(y_pred)}. Make sure that the "
                    f"predictions provided by the classifier coincides with "
                    f"the true labels."
                ) from e
            if len(unique_values) > 2:
                y_type = "multiclass"

    if y_type.startswith('multilabel'):
        y_true = csr_matrix(y_true)
        y_pred = csr_matrix(y_pred)
        y_type = 'multilabel-indicator'

    return y_type, y_true, y_pred


def _weighted_sum(sample_score, sample_weight, normalize=False):
    if normalize:
        return np.average(sample_score, weights=sample_weight)
    elif sample_weight is not None:
        return np.dot(sample_score, sample_weight)
    else:
        return sample_score.sum()


@_deprecate_positional_args
def accuracy_score(y_true, y_pred, *, normalize=True, sample_weight=None):
    """Accuracy classification score.
    In multilabel classification, this function computes subset accuracy:
    the set of labels predicted for a sample must *exactly* match the
    corresponding set of labels in y_true.
    Read more in the :ref:`User Guide <accuracy_score>`.
    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.
    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.
    normalize : bool, default=True
        If ``False``, return the number of correctly classified samples.
        Otherwise, return the fraction of correctly classified samples.
    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights.
    Returns
    -------
    score : float
        If ``normalize == True``, return the fraction of correctly
        classified samples (float), else returns the number of correctly
        classified samples (int).
        The best performance is 1 with ``normalize == True`` and the number
        of samples with ``normalize == False``.
```

```
    See Also
    --------
    jaccard_score, hamming_loss, zero_one_loss
    Notes
    -----
    In binary and multiclass classification, this function is equal
    to the ``jaccard_score`` function.
    Examples
    --------
    >>> from sklearn.metrics import accuracy_score
    >>> y_pred = [0, 2, 1, 3]
    >>> y_true = [0, 1, 2, 3]
    >>> accuracy_score(y_true, y_pred)
    0.5
    >>> accuracy_score(y_true, y_pred, normalize=False)
    2
    In the multilabel case with binary label indicators:
    >>> import numpy as np
    >>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
    0.5
    """

    # Compute accuracy for each possible representation
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    check_consistent_length(y_true, y_pred, sample_weight)
    if y_type.startswith('multilabel'):
        differing_labels = count_nonzero(y_true - y_pred, axis=1)
        score = differing_labels == 0
    else:
        score = y_true == y_pred

    return _weighted_sum(score, sample_weight, normalize)


@_deprecate_positional_args
def confusion_matrix(y_true, y_pred, *, labels=None, sample_weight=None,
                     normalize=None):
    """Compute confusion matrix to evaluate the accuracy of a classification.
    By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}`
    is equal to the number of observations known to be in group :math:`i` and
    predicted to be in group :math:`j`.
    Thus in binary classification, the count of true negatives is
    :math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is
    :math:`C_{1,1}` and false positives is :math:`C_{0,1}`.
    Read more in the :ref:`User Guide <confusion_matrix>`.
    Parameters
    ----------
    y_true : array-like of shape (n_samples,)
        Ground truth (correct) target values.
    y_pred : array-like of shape (n_samples,)
        Estimated targets as returned by a classifier.
    labels : array-like of shape (n_classes), default=None
        List of labels to index the matrix. This may be used to reorder
        or select a subset of labels.
        If ``None`` is given, those that appear at least once
        in ``y_true`` or ``y_pred`` are used in sorted order.
    sample_weight : array-like of shape (n_samples,), default=None
```

```
        Sample weights.
        .. versionadded:: 0.18
    normalize : {'true', 'pred', 'all'}, default=None
        Normalizes confusion matrix over the true (rows), predicted (columns)
        conditions or all the population. If None, confusion matrix will not b
e
        normalized.
    Returns
    -------
    C : ndarray of shape (n_classes, n_classes)
        Confusion matrix whose i-th row and j-th
        column entry indicates the number of
        samples with true label being i-th class
        and predicted label being j-th class.
    See Also
    --------
    ConfusionMatrixDisplay.from_estimator : Plot the confusion matrix
        given an estimator, the data, and the label.
    ConfusionMatrixDisplay.from_predictions : Plot the confusion matrix
        given the true and predicted labels.
    ConfusionMatrixDisplay : Confusion Matrix visualization.
    References
    ----------
    .. [1] `Wikipedia entry for the Confusion matrix
           <https://en.wikipedia.org/wiki/Confusion_matrix>`_
           (Wikipedia and other references may use a different
           convention for axes).
    Examples
    --------
    >>> from sklearn.metrics import confusion_matrix
    >>> y_true = [2, 0, 2, 2, 0, 1]
    >>> y_pred = [0, 0, 2, 2, 0, 2]
    >>> confusion_matrix(y_true, y_pred)
    array([[2, 0, 0],
           [0, 0, 1],
           [1, 0, 2]])
    >>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
    >>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
    >>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
    array([[2, 0, 0],
           [0, 0, 1],
           [1, 0, 2]])
    In the binary case, we can extract true positives, etc as follows:
    >>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
    >>> (tn, fp, fn, tp)
    (0, 2, 1, 1)
    """
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    if y_type not in ("binary", "multiclass"):
        raise ValueError("%s is not supported" % y_type)

    if labels is None:
        labels = unique_labels(y_true, y_pred)
    else:
        labels = np.asarray(labels)
        n_labels = labels.size
        if n_labels == 0:
```

```
        raise ValueError("'labels' should contains at least one label.")
    elif y_true.size == 0:
        return np.zeros((n_labels, n_labels), dtype=int)
    elif len(np.intersect1d(y_true, labels)) == 0:
        raise ValueError("At least one label specified must be in y_true")

if sample_weight is None:
    sample_weight = np.ones(y_true.shape[0], dtype=np.int64)
else:
    sample_weight = np.asarray(sample_weight)

check_consistent_length(y_true, y_pred, sample_weight)

if normalize not in ['true', 'pred', 'all', None]:
    raise ValueError("normalize must be one of {'true', 'pred', "
                     "'all', None}")

n_labels = labels.size
# If labels are not consecutive integers starting from zero, then
# y_true and y_pred must be converted into index form
need_index_conversion = not (
    labels.dtype.kind in {'i', 'u', 'b'} and
    np.all(labels == np.arange(n_labels)) and
    y_true.min() >= 0 and y_pred.min() >= 0
)
if need_index_conversion:
    label_to_ind = {y: x for x, y in enumerate(labels)}
    y_pred = np.array([label_to_ind.get(x, n_labels + 1) for x in y_pred])
    y_true = np.array([label_to_ind.get(x, n_labels + 1) for x in y_true])

# intersect y_pred, y_true with labels, eliminate items not in labels
ind = np.logical_and(y_pred < n_labels, y_true < n_labels)
if not np.all(ind):
    y_pred = y_pred[ind]
    y_true = y_true[ind]
    # also eliminate weights of eliminated items
    sample_weight = sample_weight[ind]

# Choose the accumulator dtype to always have high precision
if sample_weight.dtype.kind in {'i', 'u', 'b'}:
    dtype = np.int64
else:
    dtype = np.float64

cm = coo_matrix((sample_weight, (y_true, y_pred)),
                shape=(n_labels, n_labels), dtype=dtype,
                ).toarray()

with np.errstate(all='ignore'):
    if normalize == 'true':
        cm = cm / cm.sum(axis=1, keepdims=True)
    elif normalize == 'pred':
        cm = cm / cm.sum(axis=0, keepdims=True)
    elif normalize == 'all':
        cm = cm / cm.sum()
    cm = np.nan_to_num(cm)
```

```
    return cm
```

TestTrainSplit https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/model_selection/_split.py

In [ ]:

```
"""
The :mod:`sklearn.model_selection._split` module includes classes and
functions to split the data based on a preset strategy.
"""

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Gael Varoquaux <gael.varoquaux@normalesup.org>
#         Olivier Grisel <olivier.grisel@ensta.org>
#         Raghav RV <rvraghav93@gmail.com>
#         Leandro Hermida <hermidal@cs.umd.edu>
#         Rodion Martynov <marrodion@gmail.com>
# License: BSD 3 clause

from collections.abc import Iterable
from collections import defaultdict
import warnings
from itertools import chain, combinations
from math import ceil, floor
import numbers
from abc import ABCMeta, abstractmethod
from inspect import signature

import numpy as np
from scipy.special import comb

from ..utils import indexable, check_random_state, _safe_indexing
from ..utils import _approximate_mode
from ..utils.validation import _num_samples, column_or_1d
from ..utils.validation import check_array
from ..utils.validation import _deprecate_positional_args
from ..utils.multiclass import type_of_target
from ..base import _pprint
```

In [ ]:

```
__all__ = ['BaseCrossValidator',
           'KFold',
           'GroupKFold',
           'LeaveOneGroupOut',
           'LeaveOneOut',
           'LeavePGroupsOut',
           'LeavePOut',
           'RepeatedStratifiedKFold',
           'RepeatedKFold',
           'ShuffleSplit',
           'GroupShuffleSplit',
           'StratifiedKFold',
           'StratifiedGroupKFold',
           'StratifiedShuffleSplit',
           'PredefinedSplit',
           'train_test_split',
           'check_cv']


class BaseCrossValidator(metaclass=ABCMeta):
    """Base class for all cross-validators
    Implementations must define `_iter_test_masks` or `_iter_test_indices`.
    """
def train_test_split(*arrays,
                     test_size=None,
                     train_size=None,
                     random_state=None,
                     shuffle=True,
                     stratify=None):
    """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 or int, default=None
        If float, should be between 0.0 and 1.0 and represent the proportion
        of the dataset to include in the test split. If int, represents the
        absolute number of test samples. If None, the value is set to the
        complement of the train size. If ``train_size`` is also None, it will
        be set to 0.25.
    train_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the
        proportion of the dataset to include in the train split. If
        int, represents the absolute number of train samples. If None,
        the value is automatically set to the complement of the test size.
    random_state : int, RandomState instance or None, default=None
        Controls the shuffling applied to the data before applying the split.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.
    shuffle : bool, default=True
```

```
        Whether or not to shuffle the data before splitting. If shuffle=False
        then stratify must be None.
    stratify : array-like, default=None
        If not None, data is split in a stratified fashion, using this as
        the class labels.
        Read more in the :ref:`User Guide <stratification>`.
    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]]
    """
    n_arrays = len(arrays)
    if n_arrays == 0:
        raise ValueError("At least one array required as input")

    arrays = indexable(*arrays)

    n_samples = _num_samples(arrays[0])
    n_train, n_test = _validate_shuffle_split(n_samples, test_size, train_size
,
                                              default_test_size=0.25)

    if shuffle is False:
        if stratify is not None:
```

```
            raise ValueError(
                "Stratified train/test split is not implemented for "
                "shuffle=False")

        train = np.arange(n_train)
        test = np.arange(n_train, n_train + n_test)

    else:
        if stratify is not None:
            CVClass = StratifiedShuffleSplit
        else:
            CVClass = ShuffleSplit

        cv = CVClass(test_size=n_test,
                     train_size=n_train,
                     random_state=random_state)

        train, test = next(cv.split(X=arrays[0], y=stratify))

    return list(chain.from_iterable((_safe_indexing(a, train),
                                     _safe_indexing(a, test)) for a in arrays
))


# Tell nose that train_test_split is not a test.
# (Needed for external libraries that may use nose.)
# Use setattr to avoid mypy errors when monkeypatching.
setattr(train_test_split, '__test__', False)
```

Matplotlib.pyplot https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/pyplot.py

In [ ]:

```
# Note: The first part of this file can be modified in place, but the latter
# part is autogenerated by the boilerplate.py script.

"""
`matplotlib.pyplot` is a state-based interface to matplotlib. It provides
a MATLAB-like way of plotting.
pyplot is mainly intended for interactive plots and simple cases of
programmatic plot generation::
    import numpy as np
    import matplotlib.pyplot as plt
    x = np.arange(0, 5, 0.1)
    y = np.sin(x)
    plt.plot(x, y)
The object-oriented API is recommended for more complex plots.
"""

import functools
import importlib
import inspect
import logging
from numbers import Number
import re
import sys
import time
try:
    import threading
except ImportError:
    import dummy_threading as threading

from cycler import cycler
import matplotlib
import matplotlib.colorbar
import matplotlib.image
from matplotlib import _api
from matplotlib import rcsetup, style
from matplotlib import _pylab_helpers, interactive
from matplotlib import cbook
from matplotlib import docstring
from matplotlib.backend_bases import FigureCanvasBase, MouseButton
from matplotlib.figure import Figure, figaspect
from matplotlib.gridspec import GridSpec, SubplotSpec
from matplotlib import rcParams, rcParamsDefault, get_backend, rcParamsOrig
from matplotlib.rcsetup import interactive_bk as _interactive_bk
from matplotlib.artist import Artist
from matplotlib.axes import Axes, Subplot
from matplotlib.projections import PolarAxes
from matplotlib import mlab  # for detrend_none, window_hanning
from matplotlib.scale import get_scale_names

from matplotlib import cm
from matplotlib.cm import get_cmap, register_cmap

import numpy as np

# We may not need the following imports here:
from matplotlib.colors import Normalize
```

```
from matplotlib.lines import Line2D
from matplotlib.text import Text, Annotation
from matplotlib.patches import Polygon, Rectangle, Circle, Arrow
from matplotlib.widgets import SubplotTool, Button, Slider, Widget

from .ticker import (
    TickHelper, Formatter, FixedFormatter, NullFormatter, FuncFormatter,
    FormatStrFormatter, ScalarFormatter, LogFormatter, LogFormatterExponent,
    LogFormatterMathtext, Locator, IndexLocator, FixedLocator, NullLocator,
    LinearLocator, LogLocator, AutoLocator, MultipleLocator, MaxNLocator)

_log = logging.getLogger(__name__)


_code_objs = {
    _api.rename_parameter:
        _api.rename_parameter("", "old", "new", lambda new: None).__code__,
    _api.make_keyword_only:
        _api.make_keyword_only("", "p", lambda p: None).__code__,
}
# This function's signature is rewritten upon backend-load by switch_backend.
def show(*args, **kwargs):
    """
    Display all open figures.
    Parameters
    ----------
    block : bool, optional
        Whether to wait for all figures to be closed before returning.
        If `True` block and run the GUI main loop until all figure windows
        are closed.
        If `False` ensure that all figure windows are displayed and return
        immediately.  In this case, you are responsible for ensuring
        that the event loop is running to have responsive figures.
        Defaults to True in non-interactive mode and to False in interactive
        mode (see `.pyplot.isinteractive`).
    See Also
    --------
    ion : Enable interactive mode, which shows / updates the figure after
          every plotting command, so that calling ``show()`` is not necessary.
    ioff : Disable interactive mode.
    savefig : Save the figure to an image file instead of showing it on scree
n.
    Notes
    -----
    **Saving figures to file and showing a window at the same time**
    If you want an image file as well as a user interface window, use
    `.pyplot.savefig` before `.pyplot.show`. At the end of (a blocking)
    ``show()`` the figure is closed and thus unregistered from pyplot. Calling
    `.pyplot.savefig` afterwards would save a new and thus empty figure. This
    limitation of command order does not apply if the show is non-blocking or
    if you keep a reference to the figure and use `.Figure.savefig`.
    **Auto-show in jupyter notebooks**
    The jupyter backends (activated via ``%matplotlib inline``,
    ``%matplotlib notebook``, or ``%matplotlib widget``), call ``show()`` at
    the end of every cell by default. Thus, you usually don't have to call it
    explicitly there.
    """
```

```
    _warn_if_gui_out_of_main_thread()
    return _backend_mod.show(*args, **kwargs)
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
@_copy_docstring_and_deprecators(Axes.hist)
def hist(
        x, bins=None, range=None, density=False, weights=None,
        cumulative=False, bottom=None, histtype='bar', align='mid',
        orientation='vertical', rwidth=None, log=False, color=None,
        label=None, stacked=False, *, data=None, **kwargs):
    return gca().hist(
        x, bins=bins, range=range, density=density, weights=weights,
        cumulative=cumulative, bottom=bottom, histtype=histtype,
        align=align, orientation=orientation, rwidth=rwidth, log=log,
        color=color, label=label, stacked=stacked,
        **({"data": data} if data is not None else {}), **kwargs)
@_copy_docstring_and_deprecators(Figure.colorbar)
def colorbar(mappable=None, cax=None, ax=None, **kw):
    if mappable is None:
        mappable = gci()
        if mappable is None:
            raise RuntimeError('No mappable was found to use for colorbar '
                               'creation. First define a mappable such as '
                               'an image (with imshow) or a contour set ('
                               'with contourf).')
    ret = gcf().colorbar(mappable, cax=cax, ax=ax, **kw)
    return ret
```

graphviz https://github.com/scikit-learn/scikit-learn/blob/114616d9f6ce9eba7c1aacd3d4a254f868010e25/sklearn/tree/_export.py

In [ ]:

```
"""
This module defines export functions for decision trees.
"""

# Authors: Gilles Louppe <g.louppe@gmail.com>
#          Peter Prettenhofer <peter.prettenhofer@gmail.com>
#          Brian Holt <bdholt1@gmail.com>
#          Noel Dawe <noel@dawe.me>
#          Satrajit Gosh <satrajit.ghosh@gmail.com>
#          Trevor Stephens <trev.stephens@gmail.com>
#          Li Li <aiki.nogard@gmail.com>
#          Giuseppe Vettigli <vettigli@gmail.com>
# License: BSD 3 clause
```

In [ ]:

```
from io import StringIO
from numbers import Integral

import numpy as np

from ..utils.validation import check_is_fitted
from ..utils.validation import _deprecate_positional_args
from ..base import is_classifier

from . import _criterion
from . import _tree
from ._reingold_tilford import buchheim, Tree
from . import DecisionTreeClassifier

import warnings


def _color_brew(n):
    """Generate n colors with equally spaced hues.

    Parameters
    ----------
    n : int
        The number of colors required.

    Returns
    -------
    color_list : list, length n
        List of n tuples of form (R, G, B) being the components of each color.
    """
    color_list = []

    # Initialize saturation & value; calculate chroma & value shift
    s, v = 0.75, 0.9
    c = s * v
    m = v - c

    for h in np.arange(25, 385, 360. / n).astype(int):
        # Calculate some intermediate values
        h_bar = h / 60.
        x = c * (1 - abs((h_bar % 2) - 1))
        # Initialize RGB with same hue & chroma as our color
        rgb = [(c, x, 0),
               (x, c, 0),
               (0, c, x),
               (0, x, c),
               (x, 0, c),
               (c, 0, x),
               (c, x, 0)]
        r, g, b = rgb[int(h_bar)]
        # Shift the initial RGB values to match value and store
        rgb = [(int(255 * (r + m))),
               (int(255 * (g + m))),
               (int(255 * (b + m)))]
        color_list.append(rgb)
```

```
    return color_list


class Sentinel:
    def __repr__(self):
        return '"tree.dot"'


SENTINEL = Sentinel()


@_deprecate_positional_args
def plot_tree(decision_tree, *, max_depth=None, feature_names=None,
              class_names=None, label='all', filled=False,
              impurity=True, node_ids=False,
              proportion=False, rotate='deprecated', rounded=False,
              precision=3, ax=None, fontsize=None):
    """Plot a decision tree.

    The sample counts that are shown are weighted with any sample_weights that
    might be present.

    The visualization is fit automatically to the size of the axis.
    Use the ``figsize`` or ``dpi`` arguments of ``plt.figure``  to control
    the size of the rendering.

    Read more in the :ref:`User Guide <tree>`.

    .. versionadded:: 0.21

    Parameters
    ----------
    decision_tree : decision tree regressor or classifier
        The decision tree to be plotted.

    max_depth : int, default=None
        The maximum depth of the representation. If None, the tree is fully
        generated.

    feature_names : list of strings, default=None
        Names of each of the features.
        If None, generic names will be used ("X[0]", "X[1]", ...).

    class_names : list of str or bool, default=None
        Names of each of the target classes in ascending numerical order.
        Only relevant for classification and not supported for multi-output.
        If ``True``, shows a symbolic representation of the class name.

    label : {'all', 'root', 'none'}, default='all'
        Whether to show informative labels for impurity, etc.
        Options include 'all' to show at every node, 'root' to show only at
        the top root node, or 'none' to not show at any node.

    filled : bool, default=False
        When set to ``True``, paint nodes to indicate majority class for
        classification, extremity of values for regression, or purity of node
        for multi-output.
```

```
    impurity : bool, default=True
        When set to ``True``, show the impurity at each node.

    node_ids : bool, default=False
        When set to ``True``, show the ID number on each node.

    proportion : bool, default=False
        When set to ``True``, change the display of 'values' and/or 'samples'
        to be proportions and percentages respectively.

    rotate : bool, default=False
        This parameter has no effect on the matplotlib tree visualisation and
        it is kept here for backward compatibility.

        .. deprecated:: 0.23
           ``rotate`` is deprecated in 0.23 and will be removed in 1.0
           (renaming of 0.25).

    rounded : bool, default=False
        When set to ``True``, draw node boxes with rounded corners and use
        Helvetica fonts instead of Times-Roman.

    precision : int, default=3
        Number of digits of precision for floating point in the values of
        impurity, threshold and value attributes of each node.

    ax : matplotlib axis, default=None
        Axes to plot to. If None, use current axis. Any previous content
        is cleared.

    fontsize : int, default=None
        Size of text font. If None, determined automatically to fit figure.

    Returns
    -------
    annotations : list of artists
        List containing the artists for the annotation boxes making up the
        tree.

    Examples
    --------
    >>> from sklearn.datasets import load_iris
    >>> from sklearn import tree

    >>> clf = tree.DecisionTreeClassifier(random_state=0)
    >>> iris = load_iris()

    >>> clf = clf.fit(iris.data, iris.target)
    >>> tree.plot_tree(clf)  # doctest: +SKIP
    [Text(251.5,345.217,'X[3] <= 0.8...

    """

    check_is_fitted(decision_tree)

    if rotate != 'deprecated':
```

```
        warnings.warn(("'rotate' has no effect and is deprecated in 0.23. "
                       "It will be removed in 1.0 (renaming of 0.25)."),
                      FutureWarning)

    exporter = _MPLTreeExporter(
        max_depth=max_depth, feature_names=feature_names,
        class_names=class_names, label=label, filled=filled,
        impurity=impurity, node_ids=node_ids,
        proportion=proportion, rotate=rotate, rounded=rounded,
        precision=precision, fontsize=fontsize)
    return exporter.export(decision_tree, ax=ax)


class _BaseTreeExporter:
    def __init__(self, max_depth=None, feature_names=None,
                 class_names=None, label='all', filled=False,
                 impurity=True, node_ids=False,
                 proportion=False, rotate=False, rounded=False,
                 precision=3, fontsize=None):
        self.max_depth = max_depth
        self.feature_names = feature_names
        self.class_names = class_names
        self.label = label
        self.filled = filled
        self.impurity = impurity
        self.node_ids = node_ids
        self.proportion = proportion
        self.rotate = rotate
        self.rounded = rounded
        self.precision = precision
        self.fontsize = fontsize

    def get_color(self, value):
        # Find the appropriate color & intensity for a node
        if self.colors['bounds'] is None:
            # Classification tree
            color = list(self.colors['rgb'][np.argmax(value)])
            sorted_values = sorted(value, reverse=True)
            if len(sorted_values) == 1:
                alpha = 0
            else:
                alpha = ((sorted_values[0] - sorted_values[1])
                         / (1 - sorted_values[1]))
        else:
            # Regression tree or multi-output
            color = list(self.colors['rgb'][0])
            alpha = ((value - self.colors['bounds'][0]) /
                     (self.colors['bounds'][1] - self.colors['bounds'][0]))
        # unpack numpy scalars
        alpha = float(alpha)
        # compute the color as alpha against white
        color = [int(round(alpha * c + (1 - alpha) * 255, 0)) for c in color]
        # Return html color code in #RRGGBB format
        return '#%2x%2x%2x' % tuple(color)

    def get_fill_color(self, tree, node_id):
        # Fetch appropriate color for node
```

```
        if 'rgb' not in self.colors:
            # Initialize colors and bounds if required
            self.colors['rgb'] = _color_brew(tree.n_classes[0])
            if tree.n_outputs != 1:
                # Find max and min impurities for multi-output
                self.colors['bounds'] = (np.min(-tree.impurity),
                                         np.max(-tree.impurity))
            elif (tree.n_classes[0] == 1 and
                  len(np.unique(tree.value)) != 1):
                # Find max and min values in leaf nodes for regression
                self.colors['bounds'] = (np.min(tree.value),
                                         np.max(tree.value))
        if tree.n_outputs == 1:
            node_val = (tree.value[node_id][0, :] /
                        tree.weighted_n_node_samples[node_id])
            if tree.n_classes[0] == 1:
                # Regression
                node_val = tree.value[node_id][0, :]
        else:
            # If multi-output color node by impurity
            node_val = -tree.impurity[node_id]
        return self.get_color(node_val)

    def node_to_str(self, tree, node_id, criterion):
        # Generate the node content string
        if tree.n_outputs == 1:
            value = tree.value[node_id][0, :]
        else:
            value = tree.value[node_id]

        # Should labels be shown?
        labels = (self.label == 'root' and node_id == 0) or self.label == 'al
l'

        characters = self.characters
        node_string = characters[-1]

        # Write node ID
        if self.node_ids:
            if labels:
                node_string += 'node '
            node_string += characters[0] + str(node_id) + characters[4]

        # Write decision criteria
        if tree.children_left[node_id] != _tree.TREE_LEAF:
            # Always write node decision criteria, except for leaves
            if self.feature_names is not None:
                feature = self.feature_names[tree.feature[node_id]]
            else:
                feature = "X%s%s%s" % (characters[1],
                                       tree.feature[node_id],
                                       characters[2])
            node_string += '%s %s %s%s' % (feature,
                                           characters[3],
                                           round(tree.threshold[node_id],
                                                 self.precision),
                                           characters[4])
```

```
        # Write impurity
        if self.impurity:
            if isinstance(criterion, _criterion.FriedmanMSE):
                criterion = "friedman_mse"
            elif (isinstance(criterion, _criterion.MSE)
                  or criterion == "squared_error"):
                criterion = "squared_error"
            elif not isinstance(criterion, str):
                criterion = "impurity"
            if labels:
                node_string += '%s = ' % criterion
            node_string += (str(round(tree.impurity[node_id], self.precision))
                            + characters[4])

        # Write node sample count
        if labels:
            node_string += 'samples = '
        if self.proportion:
            percent = (100. * tree.n_node_samples[node_id] /
                       float(tree.n_node_samples[0]))
            node_string += (str(round(percent, 1)) + '%' +
                            characters[4])
        else:
            node_string += (str(tree.n_node_samples[node_id]) +
                            characters[4])

        # Write node class distribution / regression value
        if self.proportion and tree.n_classes[0] != 1:
            # For classification this will show the proportion of samples
            value = value / tree.weighted_n_node_samples[node_id]
        if labels:
            node_string += 'value = '
        if tree.n_classes[0] == 1:
            # Regression
            value_text = np.around(value, self.precision)
        elif self.proportion:
            # Classification
            value_text = np.around(value, self.precision)
        elif np.all(np.equal(np.mod(value, 1), 0)):
            # Classification without floating-point weights
            value_text = value.astype(int)
        else:
            # Classification with floating-point weights
            value_text = np.around(value, self.precision)
        # Strip whitespace
        value_text = str(value_text.astype('S32')).replace("b'", "'")
        value_text = value_text.replace("' '", ", ").replace("'", "")
        if tree.n_classes[0] == 1 and tree.n_outputs == 1:
            value_text = value_text.replace("[", "").replace("]", "")
        value_text = value_text.replace("\n ", characters[4])
        node_string += value_text + characters[4]

        # Write node majority class
        if (self.class_names is not None and
                tree.n_classes[0] != 1 and
                tree.n_outputs == 1):
```

```
            # Only done for single-output classification trees
            if labels:
                node_string += 'class = '
            if self.class_names is not True:
                class_name = self.class_names[np.argmax(value)]
            else:
                class_name = "y%s%s%s" % (characters[1],
                                          np.argmax(value),
                                          characters[2])
            node_string += class_name

        # Clean up any trailing newlines
        if node_string.endswith(characters[4]):
            node_string = node_string[:-len(characters[4])]

        return node_string + characters[5]


class _DOTTreeExporter(_BaseTreeExporter):
    def __init__(self, out_file=SENTINEL, max_depth=None,
                 feature_names=None, class_names=None, label='all',
                 filled=False, leaves_parallel=False, impurity=True,
                 node_ids=False, proportion=False, rotate=False, rounded=False
,
                 special_characters=False, precision=3, fontname='helvetica'):

        super().__init__(
            max_depth=max_depth, feature_names=feature_names,
            class_names=class_names, label=label, filled=filled,
            impurity=impurity, node_ids=node_ids, proportion=proportion,
            rotate=rotate, rounded=rounded, precision=precision)
        self.leaves_parallel = leaves_parallel
        self.out_file = out_file
        self.special_characters = special_characters
        self.fontname = fontname

        # PostScript compatibility for special characters
        if special_characters:
            self.characters = ['&#35;', '<SUB>', '</SUB>', '&le;', '<br/>',
                               '>', '<']
        else:
            self.characters = ['#', '[', ']', '<=', '\\n', '"', '"']

        # validate
        if isinstance(precision, Integral):
            if precision < 0:
                raise ValueError("'precision' should be greater or equal to
 0."
                                 " Got {} instead.".format(precision))
        else:
            raise ValueError("'precision' should be an integer. Got {}"
                             " instead.".format(type(precision)))

        # The depth of each node for plotting with 'leaf' option
        self.ranks = {'leaves': []}
        # The colors to render each node with
        self.colors = {'bounds': None}
```

```
    def export(self, decision_tree):
        # Check length of feature_names before getting into the tree node
        # Raise error if length of feature_names does not match
        # n_features_ in the decision_tree
        if self.feature_names is not None:
            if len(self.feature_names) != decision_tree.n_features_:
                raise ValueError("Length of feature_names, %d "
                                 "does not match number of features, %d"
                                 % (len(self.feature_names),
                                    decision_tree.n_features_))
        # each part writes to out_file
        self.head()
        # Now recurse the tree and add node & edge attributes
        if isinstance(decision_tree, _tree.Tree):
            self.recurse(decision_tree, 0, criterion="impurity")
        else:
            self.recurse(decision_tree.tree_, 0,
                         criterion=decision_tree.criterion)

        self.tail()

    def tail(self):
        # If required, draw leaf nodes at same depth as each other
        if self.leaves_parallel:
            for rank in sorted(self.ranks):
                self.out_file.write(
                    "{rank=same ; " +
                    "; ".join(r for r in self.ranks[rank]) + "} ;\n")
        self.out_file.write("}")

    def head(self):
        self.out_file.write('digraph Tree {\n')

        # Specify node aesthetics
        self.out_file.write('node [shape=box')
        rounded_filled = []
        if self.filled:
            rounded_filled.append('filled')
        if self.rounded:
            rounded_filled.append('rounded')
        if len(rounded_filled) > 0:
            self.out_file.write(
                ', style="%s", color="black"'
                % ", ".join(rounded_filled))

        self.out_file.write(', fontname="%s"' % self.fontname)
        self.out_file.write('] ;\n')

        # Specify graph & edge aesthetics
        if self.leaves_parallel:
            self.out_file.write(
                'graph [ranksep=equally, splines=polyline] ;\n')

        self.out_file.write('edge [fontname="%s"] ;\n' % self.fontname)

        if self.rotate:
```

```
            self.out_file.write('rankdir=LR ;\n')

        def recurse(self, tree, node_id, criterion, parent=None, depth=0):
            if node_id == _tree.TREE_LEAF:
                raise ValueError("Invalid node_id %s" % _tree.TREE_LEAF)

            left_child = tree.children_left[node_id]
            right_child = tree.children_right[node_id]

            # Add node with description
            if self.max_depth is None or depth <= self.max_depth:

                # Collect ranks for 'leaf' option in plot_options
                if left_child == _tree.TREE_LEAF:
                    self.ranks['leaves'].append(str(node_id))
                elif str(depth) not in self.ranks:
                    self.ranks[str(depth)] = [str(node_id)]
                else:
                    self.ranks[str(depth)].append(str(node_id))

                self.out_file.write(
                    '%d [label=%s' % (node_id, self.node_to_str(tree, node_id,
                                                                criterion)))

                if self.filled:
                    self.out_file.write(', fillcolor="%s"'
                                        % self.get_fill_color(tree, node_id))
                self.out_file.write('] ;\n')

                if parent is not None:
                    # Add edge to parent
                    self.out_file.write('%d -> %d' % (parent, node_id))
                    if parent == 0:
                        # Draw True/False labels if parent is root node
                        angles = np.array([45, -45]) * ((self.rotate - .5) * -2)
                        self.out_file.write(' [labeldistance=2.5, labelangle=')
                        if node_id == 1:
                            self.out_file.write('%d, headlabel="True"]' %
                                                angles[0])
                        else:
                            self.out_file.write('%d, headlabel="False"]' %
                                                angles[1])
                    self.out_file.write(' ;\n')

                if left_child != _tree.TREE_LEAF:
                    self.recurse(tree, left_child, criterion=criterion,
                                 parent=node_id, depth=depth + 1)
                    self.recurse(tree, right_child, criterion=criterion,
                                 parent=node_id, depth=depth + 1)

            else:
                self.ranks['leaves'].append(str(node_id))

                self.out_file.write('%d [label="(...)"' % node_id)
                if self.filled:
                    # color cropped nodes grey
                    self.out_file.write(', fillcolor="#C0C0C0"')
```

```
            self.out_file.write('] ;\n' % node_id)

            if parent is not None:
                # Add edge to parent
                self.out_file.write('%d -> %d ;\n' % (parent, node_id))


class _MPLTreeExporter(_BaseTreeExporter):
    def __init__(self, max_depth=None, feature_names=None,
                 class_names=None, label='all', filled=False,
                 impurity=True, node_ids=False,
                 proportion=False, rotate=False, rounded=False,
                 precision=3, fontsize=None):

        super().__init__(
            max_depth=max_depth, feature_names=feature_names,
            class_names=class_names, label=label, filled=filled,
            impurity=impurity, node_ids=node_ids, proportion=proportion,
            rotate=rotate, rounded=rounded, precision=precision)
        self.fontsize = fontsize

        # validate
        if isinstance(precision, Integral):
            if precision < 0:
                raise ValueError("'precision' should be greater or equal to
 0."
                                 " Got {} instead.".format(precision))
        else:
            raise ValueError("'precision' should be an integer. Got {}"
                             " instead.".format(type(precision)))

        # The depth of each node for plotting with 'leaf' option
        self.ranks = {'leaves': []}
        # The colors to render each node with
        self.colors = {'bounds': None}

        self.characters = ['#', '[', ']', '<=', '\n', '', '']
        self.bbox_args = dict()
        if self.rounded:
            self.bbox_args['boxstyle'] = "round"

        self.arrow_args = dict(arrowstyle="<-")

    def _make_tree(self, node_id, et, criterion, depth=0):
        # traverses _tree.Tree recursively, builds intermediate
        # "_reingold_tilford.Tree" object
        name = self.node_to_str(et, node_id, criterion=criterion)
        if (et.children_left[node_id] != _tree.TREE_LEAF
                and (self.max_depth is None or depth <= self.max_depth)):
            children = [self._make_tree(et.children_left[node_id], et,
                                        criterion, depth=depth + 1),
                        self._make_tree(et.children_right[node_id], et,
                                        criterion, depth=depth + 1)]
        else:
            return Tree(name, node_id)
        return Tree(name, node_id, *children)
```

```
    def export(self, decision_tree, ax=None):
        import matplotlib.pyplot as plt
        from matplotlib.text import Annotation

        if ax is None:
            ax = plt.gca()
        ax.clear()
        ax.set_axis_off()
        my_tree = self._make_tree(0, decision_tree.tree_,
                                  decision_tree.criterion)
        draw_tree = buchheim(my_tree)

        # important to make sure we're still
        # inside the axis after drawing the box
        # this makes sense because the width of a box
        # is about the same as the distance between boxes
        max_x, max_y = draw_tree.max_extents() + 1
        ax_width = ax.get_window_extent().width
        ax_height = ax.get_window_extent().height

        scale_x = ax_width / max_x
        scale_y = ax_height / max_y

        self.recurse(draw_tree, decision_tree.tree_, ax,
                     scale_x, scale_y, ax_height)

        anns = [ann for ann in ax.get_children()
                if isinstance(ann, Annotation)]

        # update sizes of all bboxes
        renderer = ax.figure.canvas.get_renderer()

        for ann in anns:
            ann.update_bbox_position_size(renderer)

        if self.fontsize is None:
            # get figure to data transform
            # adjust fontsize to avoid overlap
            # get max box width and height
            extents = [ann.get_bbox_patch().get_window_extent()
                       for ann in anns]
            max_width = max([extent.width for extent in extents])
            max_height = max([extent.height for extent in extents])
            # width should be around scale_x in axis coordinates
            size = anns[0].get_fontsize() * min(scale_x / max_width,
                                                scale_y / max_height)
            for ann in anns:
                ann.set_fontsize(size)

        return anns

    def recurse(self, node, tree, ax, scale_x, scale_y, height, depth=0):
        import matplotlib.pyplot as plt
        kwargs = dict(bbox=self.bbox_args.copy(), ha='center', va='center',
                      zorder=100 - 10 * depth, xycoords='axes pixels',
                      arrowprops=self.arrow_args.copy())
        kwargs['arrowprops']['edgecolor'] = plt.rcParams['text.color']
```

```
        if self.fontsize is not None:
            kwargs['fontsize'] = self.fontsize

        # offset things by .5 to center them in plot
        xy = ((node.x + .5) * scale_x, height - (node.y + .5) * scale_y)

        if self.max_depth is None or depth <= self.max_depth:
            if self.filled:
                kwargs['bbox']['fc'] = self.get_fill_color(tree,
                                                           node.tree.node_id)
            else:
                kwargs['bbox']['fc'] = ax.get_facecolor()

            if node.parent is None:
                # root
                ax.annotate(node.tree.label, xy, **kwargs)
            else:
                xy_parent = ((node.parent.x + .5) * scale_x,
                             height - (node.parent.y + .5) * scale_y)
                ax.annotate(node.tree.label, xy_parent, xy, **kwargs)
            for child in node.children:
                self.recurse(child, tree, ax, scale_x, scale_y, height,
                             depth=depth + 1)

        else:
            xy_parent = ((node.parent.x + .5) * scale_x,
                         height - (node.parent.y + .5) * scale_y)
            kwargs['bbox']['fc'] = 'grey'
            ax.annotate("\n  (...)  \n", xy_parent, xy, **kwargs)


@_deprecate_positional_args
def export_graphviz(decision_tree, out_file=None, *, max_depth=None,
                    feature_names=None, class_names=None, label='all',
                    filled=False, leaves_parallel=False, impurity=True,
                    node_ids=False, proportion=False, rotate=False,
                    rounded=False, special_characters=False, precision=3,
                    fontname='helvetica'):
    """Export a decision tree in DOT format.

    This function generates a GraphViz representation of the decision tree,
    which is then written into `out_file`. Once exported, graphical renderings
    can be generated using, for example::

        $ dot -Tps tree.dot -o tree.ps      (PostScript format)
        $ dot -Tpng tree.dot -o tree.png    (PNG format)

    The sample counts that are shown are weighted with any sample_weights that
    might be present.

    Read more in the :ref:`User Guide <tree>`.

    Parameters
    ----------
    decision_tree : decision tree classifier
        The decision tree to be exported to GraphViz.
```

```
out_file : object or str, default=None
    Handle or name of the output file. If ``None``, the result is
    returned as a string.

    .. versionchanged:: 0.20
        Default of out_file changed from "tree.dot" to None.

max_depth : int, default=None
    The maximum depth of the representation. If None, the tree is fully
    generated.

feature_names : list of str, default=None
    Names of each of the features.
    If None generic names will be used ("feature_0", "feature_1", ...).

class_names : list of str or bool, default=None
    Names of each of the target classes in ascending numerical order.
    Only relevant for classification and not supported for multi-output.
    If ``True``, shows a symbolic representation of the class name.

label : {'all', 'root', 'none'}, default='all'
    Whether to show informative labels for impurity, etc.
    Options include 'all' to show at every node, 'root' to show only at
    the top root node, or 'none' to not show at any node.

filled : bool, default=False
    When set to ``True``, paint nodes to indicate majority class for
    classification, extremity of values for regression, or purity of node
    for multi-output.

leaves_parallel : bool, default=False
    When set to ``True``, draw all leaf nodes at the bottom of the tree.

impurity : bool, default=True
    When set to ``True``, show the impurity at each node.

node_ids : bool, default=False
    When set to ``True``, show the ID number on each node.

proportion : bool, default=False
    When set to ``True``, change the display of 'values' and/or 'samples'
    to be proportions and percentages respectively.

rotate : bool, default=False
    When set to ``True``, orient tree left to right rather than top-down.

rounded : bool, default=False
    When set to ``True``, draw node boxes with rounded corners.

special_characters : bool, default=False
    When set to ``False``, ignore special characters for PostScript
    compatibility.

precision : int, default=3
    Number of digits of precision for floating point in the values of
    impurity, threshold and value attributes of each node.
```

```
    fontname : str, default='helvetica'
        Name of font used to render text.

    Returns
    -------
    dot_data : string
        String representation of the input tree in GraphViz dot format.
        Only returned if ``out_file`` is None.

        .. versionadded:: 0.18

    Examples
    --------
    >>> from sklearn.datasets import load_iris
    >>> from sklearn import tree

    >>> clf = tree.DecisionTreeClassifier()
    >>> iris = load_iris()

    >>> clf = clf.fit(iris.data, iris.target)
    >>> tree.export_graphviz(clf)
    'digraph Tree {...
    """

    check_is_fitted(decision_tree)
    own_file = False
    return_string = False
    try:
        if isinstance(out_file, str):
            out_file = open(out_file, "w", encoding="utf-8")
            own_file = True

        if out_file is None:
            return_string = True
            out_file = StringIO()

        exporter = _DOTTreeExporter(
            out_file=out_file, max_depth=max_depth,
            feature_names=feature_names, class_names=class_names, label=label,
            filled=filled, leaves_parallel=leaves_parallel, impurity=impurity,
            node_ids=node_ids, proportion=proportion, rotate=rotate,
            rounded=rounded, special_characters=special_characters,
            precision=precision, fontname=fontname)
        exporter.export(decision_tree)

        if return_string:
            return exporter.out_file.getvalue()

    finally:
        if own_file:
            out_file.close()


def _compute_depth(tree, node):
    """
    Returns the depth of the subtree rooted in node.
```

```
    """
    def compute_depth_(current_node, current_depth,
                       children_left, children_right, depths):
        depths += [current_depth]
        left = children_left[current_node]
        right = children_right[current_node]
        if left != -1 and right != -1:
            compute_depth_(left, current_depth+1,
                           children_left, children_right, depths)
            compute_depth_(right, current_depth+1,
                           children_left, children_right, depths)

    depths = []
    compute_depth_(node, 1, tree.children_left, tree.children_right, depths)
    return max(depths)


@_deprecate_positional_args
def export_text(decision_tree, *, feature_names=None, max_depth=10,
                spacing=3, decimals=2, show_weights=False):
    """Build a text report showing the rules of a decision tree.

    Note that backwards compatibility may not be supported.

    Parameters
    ----------
    decision_tree : object
        The decision tree estimator to be exported.
        It can be an instance of
        DecisionTreeClassifier or DecisionTreeRegressor.

    feature_names : list of str, default=None
        A list of length n_features containing the feature names.
        If None generic names will be used ("feature_0", "feature_1", ...).

    max_depth : int, default=10
        Only the first max_depth levels of the tree are exported.
        Truncated branches will be marked with "...".

    spacing : int, default=3
        Number of spaces between edges. The higher it is, the wider the resul
t.

    decimals : int, default=2
        Number of decimal digits to display.

    show_weights : bool, default=False
        If true the classification weights will be exported on each leaf.
        The classification weights are the number of samples each class.

    Returns
    -------
    report : string
        Text summary of all the rules in the decision tree.

    Examples
    --------
```

```
    >>> from sklearn.datasets import load_iris
    >>> from sklearn.tree import DecisionTreeClassifier
    >>> from sklearn.tree import export_text
    >>> iris = load_iris()
    >>> X = iris['data']
    >>> y = iris['target']
    >>> decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
    >>> decision_tree = decision_tree.fit(X, y)
    >>> r = export_text(decision_tree, feature_names=iris['feature_names'])
    >>> print(r)
    |--- petal width (cm) <= 0.80
    |   |--- class: 0
    |--- petal width (cm) >  0.80
    |   |--- petal width (cm) <= 1.75
    |   |   |--- class: 1
    |   |--- petal width (cm) >  1.75
    |   |   |--- class: 2
    """
    check_is_fitted(decision_tree)
    tree_ = decision_tree.tree_
    if is_classifier(decision_tree):
        class_names = decision_tree.classes_
    right_child_fmt = "{} {} <= {}\n"
    left_child_fmt = "{} {} >  {}\n"
    truncation_fmt = "{} {}\n"

    if max_depth < 0:
        raise ValueError("max_depth bust be >= 0, given %d" % max_depth)

    if (feature_names is not None and
            len(feature_names) != tree_.n_features):
        raise ValueError("feature_names must contain "
                         "%d elements, got %d" % (tree_.n_features,
                                                  len(feature_names)))

    if spacing <= 0:
        raise ValueError("spacing must be > 0, given %d" % spacing)

    if decimals < 0:
        raise ValueError("decimals must be >= 0, given %d" % decimals)

    if isinstance(decision_tree, DecisionTreeClassifier):
        value_fmt = "{}{} weights: {}\n"
        if not show_weights:
            value_fmt = "{}{}{}\n"
    else:
        value_fmt = "{}{} value: {}\n"

    if feature_names:
        feature_names_ = [feature_names[i] if i != _tree.TREE_UNDEFINED
                          else None for i in tree_.feature]
    else:
        feature_names_ = ["feature_{}".format(i) for i in tree_.feature]

    export_text.report = ""
```

```
    def _add_leaf(value, class_name, indent):
        val = ''
        is_classification = isinstance(decision_tree,
                                       DecisionTreeClassifier)
        if show_weights or not is_classification:
            val = ["{1:.{0}f}, ".format(decimals, v) for v in value]
            val = '['+''.join(val)[:-2]+']'
        if is_classification:
            val += ' class: ' + str(class_name)
        export_text.report += value_fmt.format(indent, '', val)

    def print_tree_recurse(node, depth):
        indent = ("|" + (" " * spacing)) * depth
        indent = indent[:-spacing] + "-" * spacing

        value = None
        if tree_.n_outputs == 1:
            value = tree_.value[node][0]
        else:
            value = tree_.value[node].T[0]
        class_name = np.argmax(value)

        if (tree_.n_classes[0] != 1 and
                tree_.n_outputs == 1):
            class_name = class_names[class_name]

        if depth <= max_depth+1:
            info_fmt = ""
            info_fmt_left = info_fmt
            info_fmt_right = info_fmt

            if tree_.feature[node] != _tree.TREE_UNDEFINED:
                name = feature_names_[node]
                threshold = tree_.threshold[node]
                threshold = "{1:.{0}f}".format(decimals, threshold)
                export_text.report += right_child_fmt.format(indent,
                                                             name,
                                                             threshold)
                export_text.report += info_fmt_left
                print_tree_recurse(tree_.children_left[node], depth+1)

                export_text.report += left_child_fmt.format(indent,
                                                            name,
                                                            threshold)
                export_text.report += info_fmt_right
                print_tree_recurse(tree_.children_right[node], depth+1)
            else:  # leaf
                _add_leaf(value, class_name, indent)
        else:
            subtree_depth = _compute_depth(tree_, node)
            if subtree_depth == 1:
                _add_leaf(value, class_name, indent)
            else:
                trunc_report = 'truncated branch of depth %d' % subtree_depth
                export_text.report += truncation_fmt.format(indent,
                                                            trunc_report)
```

```
    print_tree_recurse(0, 1)
    return export_text.report
```

pydotplus https://github.com/carlos-jenkins/pydotplus/blob/master/lib/pydotplus/graphviz.py

In [ ]:

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2014 Carlos Jenkins <carlos@jenkins.co.cr>
# Copyright (c) 2014 Lance Hepler
# Copyright (c) 2004-2011 Ero Carrera <ero@dkbza.org>
# Copyright (c) 2004-2007 Michael Krause <michael@krause-software.de>
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to dea
l
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FRO
M,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

"""
Graphviz's dot language Python interface.

This module provides with a full interface to create handle modify
and process graphs in Graphviz's dot language.
"""

from __future__ import division, print_function

import os
import re
import subprocess
import sys
import tempfile
import copy

from operator import itemgetter

from . import parser
```

In [ ]:

```
PY3 = not sys.version_info < (3, 0, 0)

if PY3:
    NULL_SEP = b''
    basestring = str
    long = int
    unicode = str
else:
    NULL_SEP = ''


GRAPH_ATTRIBUTES = set([
    'Damping', 'K', 'URL', 'aspect', 'bb', 'bgcolor',
    'center', 'charset', 'clusterrank', 'colorscheme', 'comment', 'compound',
    'concentrate', 'defaultdist', 'dim', 'dimen', 'diredgeconstraints',
    'dpi', 'epsilon', 'esep', 'fontcolor', 'fontname', 'fontnames',
    'fontpath', 'fontsize', 'id', 'label', 'labeljust', 'labelloc',
    'landscape', 'layers', 'layersep', 'layout', 'levels', 'levelsgap',
    'lheight', 'lp', 'lwidth', 'margin', 'maxiter', 'mclimit', 'mindist',
    'mode', 'model', 'mosek', 'nodesep', 'nojustify', 'normalize', 'nslimit',
    'nslimit1', 'ordering', 'orientation', 'outputorder', 'overlap',
    'overlap_scaling', 'pack', 'packmode', 'pad', 'page', 'pagedir',
    'quadtree', 'quantum', 'rankdir', 'ranksep', 'ratio', 'remincross',
    'repulsiveforce', 'resolution', 'root', 'rotate', 'searchsize', 'sep',
    'showboxes', 'size', 'smoothing', 'sortv', 'splines', 'start',
    'stylesheet', 'target', 'truecolor', 'viewport', 'voro_margin',
    # for subgraphs
    'rank'
])


EDGE_ATTRIBUTES = set([
    'URL', 'arrowhead', 'arrowsize', 'arrowtail',
    'color', 'colorscheme', 'comment', 'constraint', 'decorate', 'dir',
    'edgeURL', 'edgehref', 'edgetarget', 'edgetooltip', 'fontcolor',
    'fontname', 'fontsize', 'headURL', 'headclip', 'headhref', 'headlabel',
    'headport', 'headtarget', 'headtooltip', 'href', 'id', 'label',
    'labelURL', 'labelangle', 'labeldistance', 'labelfloat', 'labelfontcolor',
    'labelfontname', 'labelfontsize', 'labelhref', 'labeltarget',
    'labeltooltip', 'layer', 'len', 'lhead', 'lp', 'ltail', 'minlen',
    'nojustify', 'penwidth', 'pos', 'samehead', 'sametail', 'showboxes',
    'style', 'tailURL', 'tailclip', 'tailhref', 'taillabel', 'tailport',
    'tailtarget', 'tailtooltip', 'target', 'tooltip', 'weight',
    'rank'
])


NODE_ATTRIBUTES = set([
    'URL', 'color', 'colorscheme', 'comment',
    'distortion', 'fillcolor', 'fixedsize', 'fontcolor', 'fontname',
    'fontsize', 'group', 'height', 'id', 'image', 'imagescale', 'label',
    'labelloc', 'layer', 'margin', 'nojustify', 'orientation', 'penwidth',
    'peripheries', 'pin', 'pos', 'rects', 'regular', 'root', 'samplepoints',
    'shape', 'shapefile', 'showboxes', 'sides', 'skew', 'sortv', 'style',
    'target', 'tooltip', 'vertices', 'width', 'z',
    # The following are attributes dot2tex
```

```
    'texlbl', 'texmode'
])


CLUSTER_ATTRIBUTES = set([
    'K', 'URL', 'bgcolor', 'color', 'colorscheme',
    'fillcolor', 'fontcolor', 'fontname', 'fontsize', 'label', 'labeljust',
    'labelloc', 'lheight', 'lp', 'lwidth', 'nojustify', 'pencolor',
    'penwidth', 'peripheries', 'sortv', 'style', 'target', 'tooltip'
])


def is_string_like(obj):  # from John Hunter, types-free version
    """Check if obj is string."""
    try:
        obj + ''
    except (TypeError, ValueError):
        return False
    return True


def get_fobj(fname, mode='w+'):
    """Obtain a proper file object.

    Parameters
    ----------
    fname : string, file object, file descriptor
        If a string or file descriptor, then we create a file object.
        If *fname*
        is a file object, then we do nothing and ignore the specified *mode*
        parameter.
    mode : str
        The mode of the file to be opened.

    Returns
    -------
    fobj : file object
        The file object.
    close : bool
        If *fname* was a string, then *close* will be *True* to signify that
        the file object should be closed after writing to it. Otherwise,
        *close* will be *False* signifying that the user, in essence, created
        the file object already and that subsequent operations should not
        close it.

    """
    if is_string_like(fname):
        fobj = open(fname, mode)
        close = True
    elif hasattr(fname, 'write'):
        # fname is a file-like object, perhaps a StringIO (for example)
        fobj = fname
        close = False
    else:
        # assume it is a file descriptor
        fobj = os.fdopen(fname, mode)
        close = False
```

```
    return fobj, close


#
# Extented version of ASPN's Python Cookbook Recipe:
# Frozen dictionaries.
# http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/414283
#
# This version freezes dictionaries used as values within dictionaries.
#
class frozendict(dict):
    def _blocked_attribute(obj):
        raise AttributeError("A frozendict cannot be modified.")
    _blocked_attribute = property(_blocked_attribute)

    __delitem__ = __setitem__ = clear = _blocked_attribute
    pop = popitem = setdefault = update = _blocked_attribute

    def __new__(cls, *args, **kw):
        new = dict.__new__(cls)

        args_ = []
        for arg in args:
            if isinstance(arg, dict):
                arg = copy.copy(arg)
                for k, v in arg.items():
                    if isinstance(v, frozendict):
                        arg[k] = v
                    elif isinstance(v, dict):
                        arg[k] = frozendict(v)
                    elif isinstance(v, list):
                        v_ = list()
                        for elm in v:
                            if isinstance(elm, dict):
                                v_.append(frozendict(elm))
                            else:
                                v_.append(elm)
                        arg[k] = tuple(v_)
                args_.append(arg)
            else:
                args_.append(arg)

        dict.__init__(new, *args_, **kw)
        return new

    def __init__(self, *args, **kw):
        pass

    def __hash__(self):
        try:
            return self._cached_hash
        except AttributeError:
            h = self._cached_hash = hash(tuple(sorted(self.items())))
            return h

    def __repr__(self):
        return "frozendict(%s)" % dict.__repr__(self)
```

```
dot_keywords = ['graph', 'subgraph', 'digraph', 'node', 'edge', 'strict']

id_re_alpha_nums = re.compile('^[_a-zA-Z][a-zA-Z0-9_,]*$', re.UNICODE)
id_re_alpha_nums_with_ports = re.compile(
    '^[_a-zA-Z][a-zA-Z0-9_,:\"]*[a-zA-Z0-9_,\"]+$', re.UNICODE
)
id_re_num = re.compile('^[0-9,]+$', re.UNICODE)
id_re_with_port = re.compile('^([^:]*):([^:]*)$', re.UNICODE)
id_re_dbl_quoted = re.compile('^\".*\"$', re.S | re.UNICODE)
id_re_html = re.compile('^<.*>$', re.S | re.UNICODE)


def needs_quotes(s):
    """Checks whether a string is a dot language ID.

    It will check whether the string is solely composed
    by the characters allowed in an ID or not.
    If the string is one of the reserved keywords it will
    need quotes too but the user will need to add them
    manually.
    """

    # If the name is a reserved keyword it will need quotes but pydot
    # can't tell when it's being used as a keyword or when it's simply
    # a name. Hence the user needs to supply the quotes when an element
    # would use a reserved keyword as name. This function will return
    # false indicating that a keyword string, if provided as-is, won't
    # need quotes.
    if s in dot_keywords:
        return False

    chars = [ord(c) for c in s if ord(c) > 0x7f or ord(c) == 0]
    if chars and not id_re_dbl_quoted.match(s) and not id_re_html.match(s):
        return True

    for test_re in [
            id_re_alpha_nums, id_re_num, id_re_dbl_quoted,
            id_re_html, id_re_alpha_nums_with_ports]:
        if test_re.match(s):
            return False

    m = id_re_with_port.match(s)
    if m:
        return needs_quotes(m.group(1)) or needs_quotes(m.group(2))

    return True


def quote_if_necessary(s):
    # Older versions of graphviz throws a syntax error for empty values withou
t
    # quotes, e.g. [label=]
    if s == '':
        return '""'
```

```
    if isinstance(s, bool):
        if s is True:
            return 'True'
        return 'False'

    if not isinstance(s, basestring):
        return s

    if not s:
        return s

    if needs_quotes(s):
        replace = {'"': r'\"', "\n": r'\n', "\r": r'\r'}
        for (a, b) in replace.items():
            s = s.replace(a, b)

        return '"' + s + '"'

    return s


def graph_from_dot_data(data):
    """Load graph as defined by data in DOT format.

    The data is assumed to be in DOT format. It will
    be parsed and a Dot class will be returned,
    representing the graph.
    """

    return parser.parse_dot_data(data)


def graph_from_dot_file(path):
    """Load graph as defined by a DOT file.

    The file is assumed to be in DOT format. It will
    be loaded, parsed and a Dot class will be returned,
    representing the graph.
    """

    fd = open(path, 'rb')
    data = fd.read()
    fd.close()

    return graph_from_dot_data(data)


def graph_from_edges(edge_list, node_prefix='', directed=False):
    """Creates a basic graph out of an edge list.

    The edge list has to be a list of tuples representing
    the nodes connected by the edge.
    The values can be anything: bool, int, float, str.

    If the graph is undirected by default, it is only
    calculated from one of the symmetric halves of the matrix.
    """
```

```
    if directed:
        graph = Dot(graph_type='digraph')

    else:
        graph = Dot(graph_type='graph')

    for edge in edge_list:

        if isinstance(edge[0], str):
            src = node_prefix + edge[0]
        else:
            src = node_prefix + str(edge[0])

        if isinstance(edge[1], str):
            dst = node_prefix + edge[1]
        else:
            dst = node_prefix + str(edge[1])

        e = Edge(src, dst)
        graph.add_edge(e)

    return graph


def graph_from_adjacency_matrix(matrix, node_prefix='', directed=False):
    """Creates a basic graph out of an adjacency matrix.

    The matrix has to be a list of rows of values
    representing an adjacency matrix.
    The values can be anything: bool, int, float, as long
    as they can evaluate to True or False.
    """

    node_orig = 1

    if directed:
        graph = Dot(graph_type='digraph')
    else:
        graph = Dot(graph_type='graph')

    for row in matrix:
        if not directed:
            skip = matrix.index(row)
            r = row[skip:]
        else:
            skip = 0
            r = row
        node_dest = skip + 1

        for e in r:
            if e:
                graph.add_edge(
                    Edge(
                        node_prefix + node_orig,
                        node_prefix + node_dest
                    )
```

```
                )
            node_dest += 1
        node_orig += 1

    return graph


def graph_from_incidence_matrix(matrix, node_prefix='', directed=False):
    """Creates a basic graph out of an incidence matrix.

    The matrix has to be a list of rows of values
    representing an incidence matrix.
    The values can be anything: bool, int, float, as long
    as they can evaluate to True or False.
    """

    if directed:
        graph = Dot(graph_type='digraph')
    else:
        graph = Dot(graph_type='graph')

    for row in matrix:
        nodes = []
        c = 1

        for node in row:
            if node:
                nodes.append(c * node)
            c += 1

        nodes.sort()

        if len(nodes) == 2:
            graph.add_edge(
                Edge(
                    node_prefix + abs(nodes[0]),
                    node_prefix + nodes[1]
                )
            )

    if not directed:
        graph.set_simplify(True)

    return graph


def __find_executables(path):
    """Used by find_graphviz

    path - single directory as a string

    If any of the executables are found, it will return a dictionary
    containing the program names as keys and their paths as values.

    Otherwise returns None
    """
```

```
    success = False
    progs = {
        'dot': '',
        'twopi': '',
        'neato': '',
        'circo': '',
        'fdp': '',
        'sfdp': ''
    }

    was_quoted = False
    path = path.strip()
    if path.startswith('"') and path.endswith('"'):
        path = path[1:-1]
        was_quoted = True

    if os.path.isdir(path):
        for prg in progs.keys():
            if progs[prg]:
                continue

            if os.path.exists(os.path.join(path, prg)):
                if was_quoted:
                    progs[prg] = '"' + os.path.join(path, prg) + '"'
                else:
                    progs[prg] = os.path.join(path, prg)

                success = True

            elif os.path.exists(os.path.join(path, prg + '.exe')):
                if was_quoted:
                    progs[prg] = '"' + os.path.join(path, prg + '.exe') + '"'
                else:
                    progs[prg] = os.path.join(path, prg + '.exe')

                success = True

    if success:
        return progs
    else:
        return None


# The multi-platform version of this 'find_graphviz' function was
# contributed by Peter Cock
def find_graphviz():
    """Locate Graphviz's executables in the system.

    Tries three methods:

    First: Windows Registry (Windows only)
    This requires Mark Hammond's pywin32 is installed.

    Secondly: Search the path
    It will look for 'dot', 'twopi' and 'neato' in all the directories
    specified in the PATH environment variable.
```

```
    Thirdly: Default install location (Windows only)
    It will look for 'dot', 'twopi' and 'neato' in the default install
    location under the "Program Files" directory.

    It will return a dictionary containing the program names as keys
    and their paths as values.

    If this fails, it returns None.
    """

    # Method 1 (Windows only)
    if os.sys.platform == 'win32':

        HKEY_LOCAL_MACHINE = 0x80000002
        KEY_QUERY_VALUE = 0x0001

        RegOpenKeyEx = None
        RegQueryValueEx = None
        RegCloseKey = None

        try:
            import win32api
            RegOpenKeyEx = win32api.RegOpenKeyEx
            RegQueryValueEx = win32api.RegQueryValueEx
            RegCloseKey = win32api.RegCloseKey

        except ImportError:
            # Print a messaged suggesting they install these?
            pass

        try:
            import ctypes

            def RegOpenKeyEx(key, subkey, opt, sam):
                result = ctypes.c_uint(0)
                ctypes.windll.advapi32.RegOpenKeyExA(
                    key, subkey, opt, sam, ctypes.byref(result)
                )
                return result.value

            def RegQueryValueEx(hkey, valuename):
                data_type = ctypes.c_uint(0)
                data_len = ctypes.c_uint(1024)
                data = ctypes.create_string_buffer(1024)

                # this has a return value, which we should probably check
                ctypes.windll.advapi32.RegQueryValueExA(
                    hkey, valuename, 0, ctypes.byref(data_type),
                    data, ctypes.byref(data_len)
                )

                return data.value

            RegCloseKey = ctypes.windll.advapi32.RegCloseKey

        except ImportError:
            # Print a messaged suggesting they install these?
```

```
            pass

        if RegOpenKeyEx is not None:
            # Get the GraphViz install path from the registry
            hkey = None
            potentialKeys = [
                "SOFTWARE\\ATT\\Graphviz",
                "SOFTWARE\\AT&T Research Labs\\Graphviz"
            ]
            for potentialKey in potentialKeys:

                try:
                    hkey = RegOpenKeyEx(
                        HKEY_LOCAL_MACHINE,
                        potentialKey, 0, KEY_QUERY_VALUE
                    )

                    if hkey is not None:
                        path = RegQueryValueEx(hkey, "InstallPath")
                        RegCloseKey(hkey)

                        # The regitry variable might exist, left by old
                        # installations but with no value, in those cases we
                        # keep searching...
                        if not path:
                            continue

                        # Now append the "bin" subdirectory:
                        path = os.path.join(path, "bin")
                        progs = __find_executables(path)
                        if progs is not None:
                            # print("Used Windows registry")
                            return progs

                except Exception:
                    # raise
                    pass
                else:
                    break

    # Method 2 (Linux, Windows etc)
    if 'PATH' in os.environ:
        for path in os.environ['PATH'].split(os.pathsep):
            progs = __find_executables(path)
            if progs is not None:
                # print("Used path")
                return progs

    # Method 3 (Windows only)
    if os.sys.platform == 'win32':

        # Try and work out the equivalent of "C:\Program Files" on this
        # machine (might be on drive D:, or in a different language)
        if 'PROGRAMFILES' in os.environ:
            # Note, we could also use the win32api to get this
            # information, but win32api may not be installed.
            path = os.path.join(
```

```
                os.environ['PROGRAMFILES'], 'ATT', 'GraphViz', 'bin'
            )
        else:
            # Just in case, try the default...
            path = r"C:\Program Files\att\Graphviz\bin"

        progs = __find_executables(path)

        if progs is not None:

            # print("Used default install location")
            return progs

    for path in (
            '/usr/bin', '/usr/local/bin',
            '/opt/local/bin',
            '/opt/bin', '/sw/bin', '/usr/share',
            '/Applications/Graphviz.app/Contents/MacOS/'):

        progs = __find_executables(path)
        if progs is not None:
            # print("Used path")
            return progs

    # Failed to find GraphViz
    return None


class Common(object):
    """Common information to several classes.

    Should not be directly used, several classes are derived from
    this one.
    """

    def __getstate__(self):

        dict = copy.copy(self.obj_dict)

        return dict

    def __setstate__(self, state):

        self.obj_dict = state

    def __get_attribute__(self, attr):
        """Look for default attributes for this node"""

        attr_val = self.obj_dict['attributes'].get(attr, None)

        if attr_val is None:
            # get the defaults for nodes/edges

            default_node_name = self.obj_dict['type']

            # The defaults for graphs are set on a node named 'graph'
            if default_node_name in ('subgraph', 'digraph', 'cluster'):
```

```
                default_node_name = 'graph'

            g = self.get_parent_graph()
            if g is not None:
                defaults = g.get_node(default_node_name)
            else:
                return None

            # Multiple defaults could be set by having repeated 'graph [...]'
            # 'node [...]', 'edge [...]' statements. In such case, if the
            # same attribute is set in different statements, only the first
            # will be returned. In order to get all, one would call the
            # get_*_defaults() methods and handle those. Or go node by node
            # (of the ones specifying defaults) and modify the attributes
            # individually.
            #
            if not isinstance(defaults, (list, tuple)):
                defaults = [defaults]

            for default in defaults:
                attr_val = default.obj_dict['attributes'].get(attr, None)
                if attr_val:
                    return attr_val
        else:
            return attr_val

        return None

    def set_parent_graph(self, parent_graph):

        self.obj_dict['parent_graph'] = parent_graph

    def get_parent_graph(self):

        return self.obj_dict.get('parent_graph', None)

    def set(self, name, value):
        """Set an attribute value by name.

        Given an attribute 'name' it will set its value to 'value'.
        There's always the possibility of using the methods:

            set_'name'(value)

        which are defined for all the existing attributes.
        """

        self.obj_dict['attributes'][name] = value

    def get(self, name):
        """Get an attribute value by name.

        Given an attribute 'name' it will get its value.
        There's always the possibility of using the methods:

            get_'name'()
```

```
        which are defined for all the existing attributes.
        """

        return self.obj_dict['attributes'].get(name, None)

    def get_attributes(self):
        """"""

        return self.obj_dict['attributes']

    def set_sequence(self, seq):

        self.obj_dict['sequence'] = seq

    def get_sequence(self):

        return self.obj_dict['sequence']

    def create_attribute_methods(self, obj_attributes):

        # for attr in self.obj_dict['attributes']:
        for attr in obj_attributes:

            # Generate all the Setter methods.
            #
            self.__setattr__(
                'set_' + attr,
                lambda x, a=attr: self.obj_dict['attributes'].__setitem__(a, x
)
            )

            # Generate all the Getter methods.
            #
            self.__setattr__(
                'get_' + attr,
                lambda a=attr: self.__get_attribute__(a)
            )


class Error(Exception):
    """General error handling class.
    """
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return self.value


class InvocationException(Exception):
    """
    To indicate that a ploblem occurred while running any of the GraphViz
    executables.
    """
    def __init__(self, value):
        self.value = value
```

```
    def __str__(self):
        return self.value


class Node(Common):
    """A graph node.

    This class represents a graph's node with all its attributes.

    node(name, attribute=value, ...)

    name: node's name

    All the attributes defined in the Graphviz dot language should
    be supported.
    """

    def __init__(self, name='', obj_dict=None, **attrs):

        #
        # Nodes will take attributes of all other types because the defaults
        # for any GraphViz object are dealt with as if they were Node
        # definitions
        #

        if obj_dict is not None:
            self.obj_dict = obj_dict
        else:
            self.obj_dict = dict()

            # Copy the attributes
            #
            self.obj_dict['attributes'] = dict(attrs)
            self.obj_dict['type'] = 'node'
            self.obj_dict['parent_graph'] = None
            self.obj_dict['parent_node_list'] = None
            self.obj_dict['sequence'] = None

            # Remove the compass point
            #
            port = None
            if isinstance(name, basestring) and not name.startswith('"'):
                idx = name.find(':')
                if idx > 0 and idx + 1 < len(name):
                    name, port = name[:idx], name[idx:]

            if isinstance(name, (long, int)):
                name = str(name)

            self.obj_dict['name'] = quote_if_necessary(name)
            self.obj_dict['port'] = port

        self.create_attribute_methods(NODE_ATTRIBUTES)

    def set_name(self, node_name):
        """Set the node's name."""
```

```
        self.obj_dict['name'] = node_name

    def get_name(self):
        """Get the node's name."""

        return self.obj_dict['name']

    def get_port(self):
        """Get the node's port."""

        return self.obj_dict['port']

    def add_style(self, style):

        styles = self.obj_dict['attributes'].get('style', None)
        if not styles and style:
            styles = [style]
        else:
            styles = styles.split(',')
            styles.append(style)

        self.obj_dict['attributes']['style'] = ','.join(styles)

    def to_string(self):
        """Returns a string representation of the node in dot language.
        """

        # RMF: special case defaults for node, edge and graph properties.
        #
        node = quote_if_necessary(self.obj_dict['name'])

        node_attr = list()

        for attr, value in sorted(
                self.obj_dict['attributes'].items(),
                key=itemgetter(0)):
            if value is not None:
                node_attr.append('%s=%s' % (attr, quote_if_necessary(value)))
            else:
                node_attr.append(attr)

        # No point in having nodes setting any defaults if the don't set
        # any attributes...
        #
        if node in ('graph', 'node', 'edge') and len(node_attr) == 0:
            return ''

        node_attr = ', '.join(node_attr)

        if node_attr:
            node += ' [' + node_attr + ']'

        return node + ';'


class Edge(Common):
    """A graph edge.
```

```
    This class represents a graph's edge with all its attributes.

    edge(src, dst, attribute=value, ...)

    src: source node's name
    dst: destination node's name

    All the attributes defined in the Graphviz dot language should
    be supported.

    Attributes can be set through the dynamically generated methods:

     set_[attribute name], i.e. set_label, set_fontname

    or directly by using the instance's special dictionary:

     Edge.obj_dict['attributes'][attribute name], i.e.

        edge_instance.obj_dict['attributes']['label']
        edge_instance.obj_dict['attributes']['fontname']

    """

    def __init__(self, src='', dst='', obj_dict=None, **attrs):

        if isinstance(src, (list, tuple)) and dst == '':
            src, dst = src

        if obj_dict is not None:

            self.obj_dict = obj_dict

        else:

            self.obj_dict = dict()

            # Copy the attributes
            #
            self.obj_dict['attributes'] = dict(attrs)
            self.obj_dict['type'] = 'edge'
            self.obj_dict['parent_graph'] = None
            self.obj_dict['parent_edge_list'] = None
            self.obj_dict['sequence'] = None

            if isinstance(src, Node):
                src = src.get_name()

            if isinstance(dst, Node):
                dst = dst.get_name()

            points = (quote_if_necessary(src), quote_if_necessary(dst))

            self.obj_dict['points'] = points

        self.create_attribute_methods(EDGE_ATTRIBUTES)
```

```
    def get_source(self):
        """Get the edges source node name."""

        return self.obj_dict['points'][0]

    def get_destination(self):
        """Get the edge's destination node name."""

        return self.obj_dict['points'][1]

    def __hash__(self):
        return hash(hash(self.get_source()) + hash(self.get_destination()))

    def __eq__(self, edge):
        """Compare two edges.

        If the parent graph is directed, arcs linking
        node A to B are considered equal and A->B != B->A

        If the parent graph is undirected, any edge
        connecting two nodes is equal to any other
        edge connecting the same nodes, A->B == B->A
        """

        if not isinstance(edge, Edge):
            raise Error("Can't compare and edge to a non-edge object.")

        if self.get_parent_graph().get_top_graph_type() == 'graph':

            # If the graph is undirected, the edge has neither
            # source nor destination.
            #
            if ((self.get_source() == edge.get_source() and
                    self.get_destination() == edge.get_destination()) or
                (edge.get_source() == self.get_destination() and
                    edge.get_destination() == self.get_source())):
                return True

        else:
            if (self.get_source() == edge.get_source() and
                    self.get_destination() == edge.get_destination()):
                return True

        return False

    def parse_node_ref(self, node_str):

        if not isinstance(node_str, str):
            return node_str

        if node_str.startswith('"') and node_str.endswith('"'):
            return node_str

        node_port_idx = node_str.rfind(':')

        if (node_port_idx > 0 and node_str[0] == '"' and
                node_str[node_port_idx - 1] == '"'):
```

```
            return node_str

        if node_port_idx > 0:
            a = node_str[:node_port_idx]
            b = node_str[node_port_idx + 1:]

            node = quote_if_necessary(a)

            node += ':' + quote_if_necessary(b)

            return node

        return node_str

    def to_string(self):
        """Returns a string representation of the edge in dot language.
        """

        src = self.parse_node_ref(self.get_source())
        dst = self.parse_node_ref(self.get_destination())

        if isinstance(src, frozendict):
            edge = [Subgraph(obj_dict=src).to_string()]
        elif isinstance(src, (int, long)):
            edge = [str(src)]
        else:
            edge = [src]

        if (self.get_parent_graph() and
                self.get_parent_graph().get_top_graph_type() and
                self.get_parent_graph().get_top_graph_type() == 'digraph'):

            edge.append('->')

        else:
            edge.append('--')

        if isinstance(dst, frozendict):
            edge.append(Subgraph(obj_dict=dst).to_string())
        elif isinstance(dst, (int, long)):
            edge.append(str(dst))
        else:
            edge.append(dst)

        edge_attr = list()

        for attr, value in sorted(
                self.obj_dict['attributes'].items(),
                key=itemgetter(0)):
            if value is not None:
                edge_attr.append('%s=%s' % (attr, quote_if_necessary(value)))
            else:
                edge_attr.append(attr)

        edge_attr = ', '.join(edge_attr)

        if edge_attr:
```

```
            edge.append(' [' + edge_attr + ']')

        return ' '.join(edge) + ';'


class Graph(Common):
    """Class representing a graph in Graphviz's dot language.

    This class implements the methods to work on a representation
    of a graph in Graphviz's dot language.

    graph(graph_name='G', graph_type='digraph',
        strict=False, suppress_disconnected=False, attribute=value, ...)

    graph_name:
        the graph's name
    graph_type:
        can be 'graph' or 'digraph'
    suppress_disconnected:
        defaults to False, which will remove from the
        graph any disconnected nodes.
    simplify:
        if True it will avoid displaying equal edges, i.e.
        only one edge between two nodes. removing the
        duplicated ones.

    All the attributes defined in the Graphviz dot language should
    be supported.

    Attributes can be set through the dynamically generated methods:

     set_[attribute name], i.e. set_size, set_fontname

    or using the instance's attributes:

     Graph.obj_dict['attributes'][attribute name], i.e.

        graph_instance.obj_dict['attributes']['label']
        graph_instance.obj_dict['attributes']['fontname']
    """

    def __init__(
            self, graph_name='G', obj_dict=None, graph_type='digraph',
            strict=False, suppress_disconnected=False, simplify=False,
            **attrs):

        if obj_dict is not None:
            self.obj_dict = obj_dict
        else:
            self.obj_dict = dict()

            self.obj_dict['attributes'] = dict(attrs)

            if graph_type not in ['graph', 'digraph']:
                raise Error((
                    'Invalid type "%s". Accepted graph types are: '
                    'graph, digraph, subgraph' % graph_type
```

```
            ))

        self.obj_dict['name'] = quote_if_necessary(graph_name)
        self.obj_dict['type'] = graph_type

        self.obj_dict['strict'] = strict
        self.obj_dict['suppress_disconnected'] = suppress_disconnected
        self.obj_dict['simplify'] = simplify

        self.obj_dict['current_child_sequence'] = 1
        self.obj_dict['nodes'] = dict()
        self.obj_dict['edges'] = dict()
        self.obj_dict['subgraphs'] = dict()

        self.set_parent_graph(self)

    self.create_attribute_methods(GRAPH_ATTRIBUTES)

def get_graph_type(self):
    return self.obj_dict['type']

def get_top_graph_type(self):
    parent = self
    while True:
        parent_ = parent.get_parent_graph()
        if parent_ == parent:
            break
        parent = parent_

    return parent.obj_dict['type']

def set_graph_defaults(self, **attrs):
    self.add_node(Node('graph', **attrs))

def get_graph_defaults(self, **attrs):

    graph_nodes = self.get_node('graph')

    if isinstance(graph_nodes, (list, tuple)):
        return [node.get_attributes() for node in graph_nodes]

    return graph_nodes.get_attributes()

def set_node_defaults(self, **attrs):
    self.add_node(Node('node', **attrs))

def get_node_defaults(self, **attrs):
    graph_nodes = self.get_node('node')

    if isinstance(graph_nodes, (list, tuple)):
        return [node.get_attributes() for node in graph_nodes]

    return graph_nodes.get_attributes()

def set_edge_defaults(self, **attrs):
    self.add_node(Node('edge', **attrs))
```

```
    def get_edge_defaults(self, **attrs):
        graph_nodes = self.get_node('edge')

        if isinstance(graph_nodes, (list, tuple)):
            return [node.get_attributes() for node in graph_nodes]

        return graph_nodes.get_attributes()

    def set_simplify(self, simplify):
        """Set whether to simplify or not.

        If True it will avoid displaying equal edges, i.e.
        only one edge between two nodes. removing the
        duplicated ones.
        """

        self.obj_dict['simplify'] = simplify

    def get_simplify(self):
        """Get whether to simplify or not.

        Refer to set_simplify for more information.
        """

        return self.obj_dict['simplify']

    def set_type(self, graph_type):
        """Set the graph's type, 'graph' or 'digraph'."""

        self.obj_dict['type'] = graph_type

    def get_type(self):
        """Get the graph's type, 'graph' or 'digraph'."""

        return self.obj_dict['type']

    def set_name(self, graph_name):
        """Set the graph's name."""

        self.obj_dict['name'] = graph_name

    def get_name(self):
        """Get the graph's name."""

        return self.obj_dict['name']

    def set_strict(self, val):
        """Set graph to 'strict' mode.

        This option is only valid for top level graphs.
        """

        self.obj_dict['strict'] = val

    def get_strict(self, val):
        """Get graph's 'strict' mode (True, False).
```

```
        This option is only valid for top level graphs.
        """

        return self.obj_dict['strict']

    def set_suppress_disconnected(self, val):
        """Suppress disconnected nodes in the output graph.

        This option will skip nodes in the graph with no incoming or outgoing
        edges. This option works also for subgraphs and has effect only in the
        current graph/subgraph.
        """

        self.obj_dict['suppress_disconnected'] = val

    def get_suppress_disconnected(self, val):
        """Get if suppress disconnected is set.

        Refer to set_suppress_disconnected for more information.
        """

        return self.obj_dict['suppress_disconnected']

    def get_next_sequence_number(self):
        seq = self.obj_dict['current_child_sequence']
        self.obj_dict['current_child_sequence'] += 1
        return seq

    def add_node(self, graph_node):
        """Adds a node object to the graph.

        It takes a node object as its only argument and returns
        None.
        """

        if not isinstance(graph_node, Node):
            raise TypeError(
                'add_node() received a non node '
                'class object: {}'.format(str(graph_node))
            )

        node = self.get_node(graph_node.get_name())

        if not node:
            self.obj_dict['nodes'][graph_node.get_name()] = [
                graph_node.obj_dict
            ]

            # self.node_dict[graph_node.get_name()] = graph_node.attributes
            graph_node.set_parent_graph(self.get_parent_graph())
        else:
            self.obj_dict['nodes'][graph_node.get_name()].append(
                graph_node.obj_dict
            )

        graph_node.set_sequence(self.get_next_sequence_number())
```

```
    def del_node(self, name, index=None):
        """Delete a node from the graph.

        Given a node's name all node(s) with that same name
        will be deleted if 'index' is not specified or set
        to None.
        If there are several nodes with that same name and
        'index' is given, only the node in that position
        will be deleted.

        'index' should be an integer specifying the position
        of the node to delete. If index is larger than the
        number of nodes with that name, no action is taken.

        If nodes are deleted it returns True. If no action
        is taken it returns False.
        """

        if isinstance(name, Node):
            name = name.get_name()

        if name in self.obj_dict['nodes']:
            if index is not None and index < len(self.obj_dict['nodes'][name
]):
                del self.obj_dict['nodes'][name][index]
                return True
            else:
                del self.obj_dict['nodes'][name]
                return True

        return False

    def get_node(self, name):
        """Retrieve a node from the graph.

        Given a node's name the corresponding Node
        instance will be returned.

        If one or more nodes exist with that name a list of
        Node instances is returned.
        An empty list is returned otherwise.
        """

        match = list()

        if name in self.obj_dict['nodes']:
            match.extend([
                Node(obj_dict=obj_dict)
                for obj_dict
                in self.obj_dict['nodes'][name]
            ])

        return match

    def get_nodes(self):
        """Get the list of Node instances."""
```

```
        return self.get_node_list()

    def get_node_list(self):
        """Get the list of Node instances.

        This method returns the list of Node instances
        composing the graph.
        """

        node_objs = list()

        for node, obj_dict_list in self.obj_dict['nodes'].items():
            node_objs.extend([
                Node(obj_dict=obj_d)
                for obj_d
                in obj_dict_list
            ])

        return node_objs

    def add_edge(self, graph_edge):
        """Adds an edge object to the graph.

        It takes a edge object as its only argument and returns
        None.
        """

        if not isinstance(graph_edge, Edge):
            raise TypeError(
                'add_edge() received a non '
                'edge class object: {}'.format(str(graph_edge))
            )

        edge_points = (graph_edge.get_source(), graph_edge.get_destination())

        if edge_points in self.obj_dict['edges']:

            edge_list = self.obj_dict['edges'][edge_points]
            edge_list.append(graph_edge.obj_dict)
        else:
            self.obj_dict['edges'][edge_points] = [graph_edge.obj_dict]

        graph_edge.set_sequence(self.get_next_sequence_number())
        graph_edge.set_parent_graph(self.get_parent_graph())

    def del_edge(self, src_or_list, dst=None, index=None):
        """Delete an edge from the graph.

        Given an edge's (source, destination) node names all
        matching edges(s) will be deleted if 'index' is not
        specified or set to None.
        If there are several matching edges and 'index' is
        given, only the edge in that position will be deleted.

        'index' should be an integer specifying the position
        of the edge to delete. If index is larger than the
        number of matching edges, no action is taken.
```

```
        If edges are deleted it returns True. If no action
        is taken it returns False.
        """

        if isinstance(src_or_list, (list, tuple)):
            if dst is not None and isinstance(dst, (int, long)):
                index = dst
            src, dst = src_or_list
        else:
            src, dst = src_or_list, dst

        if isinstance(src, Node):
            src = src.get_name()

        if isinstance(dst, Node):
            dst = dst.get_name()

        if (src, dst) in self.obj_dict['edges']:
            if index is not None and index < len(
                    self.obj_dict['edges'][(src, dst)]):
                del self.obj_dict['edges'][(src, dst)][index]
                return True
            else:
                del self.obj_dict['edges'][(src, dst)]
                return True

        return False

    def get_edge(self, src_or_list, dst=None):
        """Retrieved an edge from the graph.

        Given an edge's source and destination the corresponding
        Edge instance(s) will be returned.

        If one or more edges exist with that source and destination
        a list of Edge instances is returned.
        An empty list is returned otherwise.
        """

        if isinstance(src_or_list, (list, tuple)) and dst is None:
            edge_points = tuple(src_or_list)
            edge_points_reverse = (edge_points[1], edge_points[0])
        else:
            edge_points = (src_or_list, dst)
            edge_points_reverse = (dst, src_or_list)

        match = list()

        if edge_points in self.obj_dict['edges'] or (
                self.get_top_graph_type() == 'graph' and
                edge_points_reverse in self.obj_dict['edges']):

            edges_obj_dict = self.obj_dict['edges'].get(
                edge_points,
                self.obj_dict['edges'].get(edge_points_reverse, None))
```

```
            for edge_obj_dict in edges_obj_dict:
                match.append(Edge(
                    edge_points[0],
                    edge_points[1],
                    obj_dict=edge_obj_dict
                ))

        return match

    def get_edges(self):
        return self.get_edge_list()

    def get_edge_list(self):
        """Get the list of Edge instances.

        This method returns the list of Edge instances
        composing the graph.
        """

        edge_objs = list()

        for edge, obj_dict_list in self.obj_dict['edges'].items():
            edge_objs.extend([
                Edge(obj_dict=obj_d)
                for obj_d
                in obj_dict_list
            ])

        return edge_objs

    def add_subgraph(self, sgraph):
        """Adds an subgraph object to the graph.

        It takes a subgraph object as its only argument and returns
        None.
        """

        if not isinstance(sgraph, Subgraph) and \
                not isinstance(sgraph, Cluster):
            raise TypeError(
                'add_subgraph() received a non '
                'subgraph class object:'.format(str(sgraph))
            )

        if sgraph.get_name() in self.obj_dict['subgraphs']:

            sgraph_list = self.obj_dict['subgraphs'][sgraph.get_name()]
            sgraph_list.append(sgraph.obj_dict)

        else:
            self.obj_dict['subgraphs'][sgraph.get_name()] = [sgraph.obj_dict]

        sgraph.set_sequence(self.get_next_sequence_number())

        sgraph.set_parent_graph(self.get_parent_graph())

    def get_subgraph(self, name):
```

```
        """Retrieved a subgraph from the graph.

        Given a subgraph's name the corresponding
        Subgraph instance will be returned.

        If one or more subgraphs exist with the same name, a list of
        Subgraph instances is returned.
        An empty list is returned otherwise.
        """

        match = list()

        if name in self.obj_dict['subgraphs']:
            sgraphs_obj_dict = self.obj_dict['subgraphs'].get(name)

            for obj_dict_list in sgraphs_obj_dict:
                match.append(Subgraph(obj_dict=obj_dict_list))

        return match

    def get_subgraphs(self):
        return self.get_subgraph_list()

    def get_subgraph_list(self):
        """Get the list of Subgraph instances.

        This method returns the list of Subgraph instances
        in the graph.
        """

        sgraph_objs = list()

        for sgraph, obj_dict_list in self.obj_dict['subgraphs'].items():
            sgraph_objs.extend([
                Subgraph(obj_dict=obj_d)
                for obj_d
                in obj_dict_list
            ])

        return sgraph_objs

    def set_parent_graph(self, parent_graph):

        self.obj_dict['parent_graph'] = parent_graph

        for obj_list in self.obj_dict['nodes'].values():
            for obj in obj_list:
                obj['parent_graph'] = parent_graph

        for obj_list in self.obj_dict['edges'].values():
            for obj in obj_list:
                obj['parent_graph'] = parent_graph

        for obj_list in self.obj_dict['subgraphs'].values():
            for obj in obj_list:
                Graph(obj_dict=obj).set_parent_graph(parent_graph)
```

```
    def to_string(self):
        """Returns a string representation of the graph in dot language.

        It will return the graph and all its subelements in string from.
        """

        graph = list()

        if self.obj_dict.get('strict', None) is not None:
            if self == self.get_parent_graph() and self.obj_dict['strict']:
                graph.append('strict ')

        if self.obj_dict['name'] == '':
            if 'show_keyword' in self.obj_dict and \
                    self.obj_dict['show_keyword']:
                graph.append('subgraph {\n')
            else:
                graph.append('{\n')
        else:
            graph.append(
                '{} {} {{\n'.format(
                    self.obj_dict['type'],
                    self.obj_dict['name']
                )
            )

        for attr, value in sorted(
                self.obj_dict['attributes'].items(),
                key=itemgetter(0)):
            if value is not None:
                graph.append('%s=%s' % (attr, quote_if_necessary(value)))
            else:
                graph.append(attr)

            graph.append(';\n')

        edges_done = set()

        edge_obj_dicts = list()
        for e in self.obj_dict['edges'].values():
            edge_obj_dicts.extend(e)

        if edge_obj_dicts:
            edge_src_set, edge_dst_set = list(
                zip(*[obj['points'] for obj in edge_obj_dicts])
            )
            edge_src_set, edge_dst_set = set(edge_src_set), set(edge_dst_set)
        else:
            edge_src_set, edge_dst_set = set(), set()

        node_obj_dicts = list()
        for e in self.obj_dict['nodes'].values():
            node_obj_dicts.extend(e)

        sgraph_obj_dicts = list()
        for sg in self.obj_dict['subgraphs'].values():
            sgraph_obj_dicts.extend(sg)
```

```
        obj_list = sorted([
            (obj['sequence'], obj)
            for obj
            in (edge_obj_dicts + node_obj_dicts + sgraph_obj_dicts)
        ])

        for idx, obj in obj_list:
            if obj['type'] == 'node':
                node = Node(obj_dict=obj)

                if self.obj_dict.get('suppress_disconnected', False):
                    if (node.get_name() not in edge_src_set and
                            node.get_name() not in edge_dst_set):
                        continue

                graph.append(node.to_string() + '\n')

            elif obj['type'] == 'edge':
                edge = Edge(obj_dict=obj)

                if self.obj_dict.get('simplify', False) and edge in edges_done
:
                    continue

                graph.append(edge.to_string() + '\n')
                edges_done.add(edge)
            else:
                sgraph = Subgraph(obj_dict=obj)
                graph.append(sgraph.to_string() + '\n')

        graph.append('}\n')

        return ''.join(graph)


class Subgraph(Graph):

    """Class representing a subgraph in Graphviz's dot language.

    This class implements the methods to work on a representation
    of a subgraph in Graphviz's dot language.

    subgraph(
        graph_name='subG', suppress_disconnected=False, attribute=value, ...
    )

    graph_name:
        the subgraph's name
    suppress_disconnected:
        defaults to false, which will remove from the
        subgraph any disconnected nodes.
    All the attributes defined in the Graphviz dot language should
    be supported.

    Attributes can be set through the dynamically generated methods:
```

```
     set_[attribute name], i.e. set_size, set_fontname

    or using the instance's attributes:

     Subgraph.obj_dict['attributes'][attribute name], i.e.

        subgraph_instance.obj_dict['attributes']['label']
        subgraph_instance.obj_dict['attributes']['fontname']
    """

    # RMF: subgraph should have all the attributes of graph so it can be passe
d
    # as a graph to all methods
    #
    def __init__(
            self, graph_name='', obj_dict=None, suppress_disconnected=False,
            simplify=False, **attrs):

        Graph.__init__(
            self, graph_name=graph_name, obj_dict=obj_dict,
            suppress_disconnected=suppress_disconnected,
            simplify=simplify, **attrs)

        if obj_dict is None:
            self.obj_dict['type'] = 'subgraph'


class Cluster(Graph):

    """Class representing a cluster in Graphviz's dot language.

    This class implements the methods to work on a representation
    of a cluster in Graphviz's dot language.

    cluster(
        graph_name='subG', suppress_disconnected=False, attribute=value, ...
    )

    graph_name:
        the cluster's name (the string 'cluster' will be always prepended)
    suppress_disconnected:
        defaults to false, which will remove from the
        cluster any disconnected nodes.
    All the attributes defined in the Graphviz dot language should
    be supported.

    Attributes can be set through the dynamically generated methods:

     set_[attribute name], i.e. set_color, set_fontname

    or using the instance's attributes:

     Cluster.obj_dict['attributes'][attribute name], i.e.

        cluster_instance.obj_dict['attributes']['label']
        cluster_instance.obj_dict['attributes']['fontname']
    """
```

```
    def __init__(
            self, graph_name='subG', obj_dict=None,
            suppress_disconnected=False,
            simplify=False, **attrs):

        Graph.__init__(
            self, graph_name=graph_name, obj_dict=obj_dict,
            suppress_disconnected=suppress_disconnected, simplify=simplify,
            **attrs
        )

        if obj_dict is None:
            self.obj_dict['type'] = 'subgraph'
            self.obj_dict['name'] = 'cluster_' + graph_name

        self.create_attribute_methods(CLUSTER_ATTRIBUTES)


class Dot(Graph):
    """A container for handling a dot language file.

    This class implements methods to write and process
    a dot language file. It is a derived class of
    the base class 'Graph'.
    """

    def __init__(self, *argsl, **argsd):
        Graph.__init__(self, *argsl, **argsd)

        self.shape_files = list()
        self.progs = None
        self.formats = [
            'canon', 'cmap', 'cmapx', 'cmapx_np', 'dia', 'dot',
            'fig', 'gd', 'gd2', 'gif', 'hpgl', 'imap', 'imap_np', 'ismap',
            'jpe', 'jpeg', 'jpg', 'mif', 'mp', 'pcl', 'pdf', 'pic', 'plain',
            'plain-ext', 'png', 'ps', 'ps2', 'svg', 'svgz', 'vml', 'vmlz',
            'vrml', 'vtx', 'wbmp', 'xdot', 'xlib'
        ]
        self.prog = 'dot'

        # Automatically creates all the methods enabling the creation
        # of output in any of the supported formats.
        for frmt in self.formats:
            self.__setattr__(
                'create_' + frmt,
                lambda f=frmt, prog=self.prog: self.create(format=f, prog=prog
)
            )
            f = self.__dict__['create_' + frmt]
            f.__doc__ = (
                '''Refer to the docstring accompanying the'''
                '''create' method for more information.'''
            )

        for frmt in self.formats + ['raw']:
            self.__setattr__(
```

```
            'write_' + frmt,
            lambda path,
            f=frmt,
            prog=self.prog: self.write(path, format=f, prog=prog)
        )

        f = self.__dict__['write_' + frmt]
        f.__doc__ = (
            '''Refer to the docstring accompanying the'''
            ''''write' method for more information.'''
        )

def __getstate__(self):
    return copy.copy(self.obj_dict)

def __setstate__(self, state):
    self.obj_dict = state

def set_shape_files(self, file_paths):
    """Add the paths of the required image files.

    If the graph needs graphic objects to be used as shapes or otherwise
    those need to be in the same folder as the graph is going to be
    rendered from. Alternatively the absolute path to the files can be
    specified when including the graphics in the graph.

    The files in the location pointed to by the path(s) specified as
    arguments to this method will be copied to the same temporary location
    where the graph is going to be rendered.
    """

    if isinstance(file_paths, basestring):
        self.shape_files.append(file_paths)

    if isinstance(file_paths, (list, tuple)):
        self.shape_files.extend(file_paths)

def set_prog(self, prog):
    """Sets the default program.

    Sets the default program in charge of processing
    the dot file into a graph.
    """
    self.prog = prog

def set_graphviz_executables(self, paths):
    """
    This method allows to manually specify the location of the GraphViz
    executables.

    The argument to this method should be a dictionary where the keys are
    as follows:

        {'dot': '', 'twopi': '', 'neato': '', 'circo': '', 'fdp': ''}

    and the values are the paths to the corresponding executable,
    including the name of the executable itself.
```

```
        """

        self.progs = paths

    def write(self, path, prog=None, format='raw'):
        """
        Given a filename 'path' it will open/create and truncate
        such file and write on it a representation of the graph
        defined by the dot object and in the format specified by
        'format'. 'path' can also be an open file-like object, such as
        a StringIO instance.

        The format 'raw' is used to dump the string representation
        of the Dot object, without further processing.
        The output can be processed by any of graphviz tools, defined
        in 'prog', which defaults to 'dot'
        Returns True or False according to the success of the write
        operation.

        There's also the preferred possibility of using:

            write_'format'(path, prog='program')

        which are automatically defined for all the supported formats.
        [write_ps(), write_gif(), write_dia(), ...]

        """
        if prog is None:
            prog = self.prog

        fobj, close = get_fobj(path, 'w+b')
        try:
            if format == 'raw':
                data = self.to_string()
                if isinstance(data, basestring):
                    if not isinstance(data, unicode):
                        try:
                            data = unicode(data, 'utf-8')
                        except:
                            pass

                try:
                    charset = self.get_charset()
                    if not PY3 or not charset:
                        charset = 'utf-8'
                    data = data.encode(charset)
                except:
                    if PY3:
                        data = data.encode('utf-8')
                    pass

                fobj.write(data)

            else:
                fobj.write(self.create(prog, format))
        finally:
            if close:
```

```
                fobj.close()

        return True

    def create(self, prog=None, format='ps'):
        """Creates and returns a Postscript representation of the graph.

        create will write the graph to a temporary dot file and process
        it with the program given by 'prog' (which defaults to 'twopi'),
        reading the Postscript output and returning it as a string is the
        operation is successful.
        On failure None is returned.

        There's also the preferred possibility of using:

            create_'format'(prog='program')

        which are automatically defined for all the supported formats.
        [create_ps(), create_gif(), create_dia(), ...]

        If 'prog' is a list instead of a string the fist item is expected
        to be the program name, followed by any optional command-line
        arguments for it:

            ['twopi', '-Tdot', '-s10']
        """

        if prog is None:
            prog = self.prog

        if isinstance(prog, (list, tuple)):
            prog, args = prog[0], prog[1:]
        else:
            args = []

        if self.progs is None:
            self.progs = find_graphviz()
            if self.progs is None:
                raise InvocationException(
                    'GraphViz\'s executables not found')

        if prog not in self.progs:
            raise InvocationException(
                'GraphViz\'s executable "%s" not found' % prog)

        if not os.path.exists(self.progs[prog]) or \
                not os.path.isfile(self.progs[prog]):
            raise InvocationException(
                'GraphViz\'s executable "{}" is not'
                ' a file or doesn\'t exist'.format(self.progs[prog])
            )

        tmp_fd, tmp_name = tempfile.mkstemp()
        os.close(tmp_fd)
        self.write(tmp_name)
        tmp_dir = os.path.dirname(tmp_name)
```

```
        # For each of the image files...
        for img in self.shape_files:

            # Get its data
            f = open(img, 'rb')
            f_data = f.read()
            f.close()

            # And copy it under a file with the same name in the temporary
            # directory
            f = open(os.path.join(tmp_dir, os.path.basename(img)), 'wb')
            f.write(f_data)
            f.close()

        cmdline = [self.progs[prog], '-T' + format, tmp_name] + args

        p = subprocess.Popen(
            cmdline,
            cwd=tmp_dir,
            stderr=subprocess.PIPE, stdout=subprocess.PIPE)

        stderr = p.stderr
        stdout = p.stdout

        stdout_output = list()
        while True:
            data = stdout.read()
            if not data:
                break
            stdout_output.append(data)
        stdout.close()

        stdout_output = NULL_SEP.join(stdout_output)

        if not stderr.closed:
            stderr_output = list()
            while True:
                data = stderr.read()
                if not data:
                    break
                stderr_output.append(data)
            stderr.close()

            if stderr_output:
                stderr_output = NULL_SEP.join(stderr_output)
                if PY3:
                    stderr_output = stderr_output.decode(sys.stderr.encoding)

        # pid, status = os.waitpid(p.pid, 0)
        status = p.wait()

        if status != 0:
            raise InvocationException(
                'Program terminated with status: %d. stderr follows: %s' % (
                    status, stderr_output))
        elif stderr_output:
            print(stderr_output)
```

```
        # For each of the image files...
        for img in self.shape_files:

            # remove it
            os.unlink(os.path.join(tmp_dir, os.path.basename(img)))

        os.unlink(tmp_name)

        return stdout_output
```

References

[1] Afroz Chakure,"Decision Tree Classification", https://medium.com/swlh/decision-tree-classification-de64fc4d5aac

[2] Scikit-learn,"Naive Bayes", https://scikit-learn.org/stable/modules/naive_bayes.html

[3] Scikit-learn,"Decision Tree", https://scikit-learn.org/stable/modules/tree.html#classification

[4] Scikit-learn,"sklearn.model_selection.KFold", https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

[5] Jason Brownlee, "A Gentle Introduction to k-fold Cross-Validation", May 23,2018, https://machinelearningmastery.com/k-fold-cross-validation/

[6] Scikit-learn,"sklearn.naive_bayes.MultinomialNB", https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

[7] Sarang Nakarke,"Understanding Confusion matrix",May 9th,2018, https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62