In [0]:
```
## 17BCE0136 R.S.Rahul Sai

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

Dataset Description

This dataset is composed of a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease (PD). Each column in the table is a particular voice measure, and each row corresponds one of 195 voice recording from these individuals ("name" column). The main aim of the data is to discriminate healthy people from those with PD, according to "status" column which is set to 0 for healthy and 1 for PD.The rows of the CSV file contain an instance corresponding to one voice recording. There are around six recordings per patient, the name of the patient is identified in the first column.

In [0]:
```
data=pd.read_csv("parkinsons.data.csv")
```

In [0]:
```
Y=data['status']
data.drop(['status','name'],inplace=True,axis=1)
X=data
```

In [0]:
```
X_train,X_test,Y_train,Y_test=train_test_split(X,Y,test_size=0.3,random_state=0)
```

Decision Tree Classifier (Library)

In [5]:
```
from sklearn.tree import DecisionTreeClassifier,export_graphviz
from sklearn.metrics import r2_score,mean_squared_error
from IPython.display import Image
import pydotplus

dectree=DecisionTreeClassifier(max_depth=5,criterion='gini')
dectree.fit(X_train,Y_train)
```

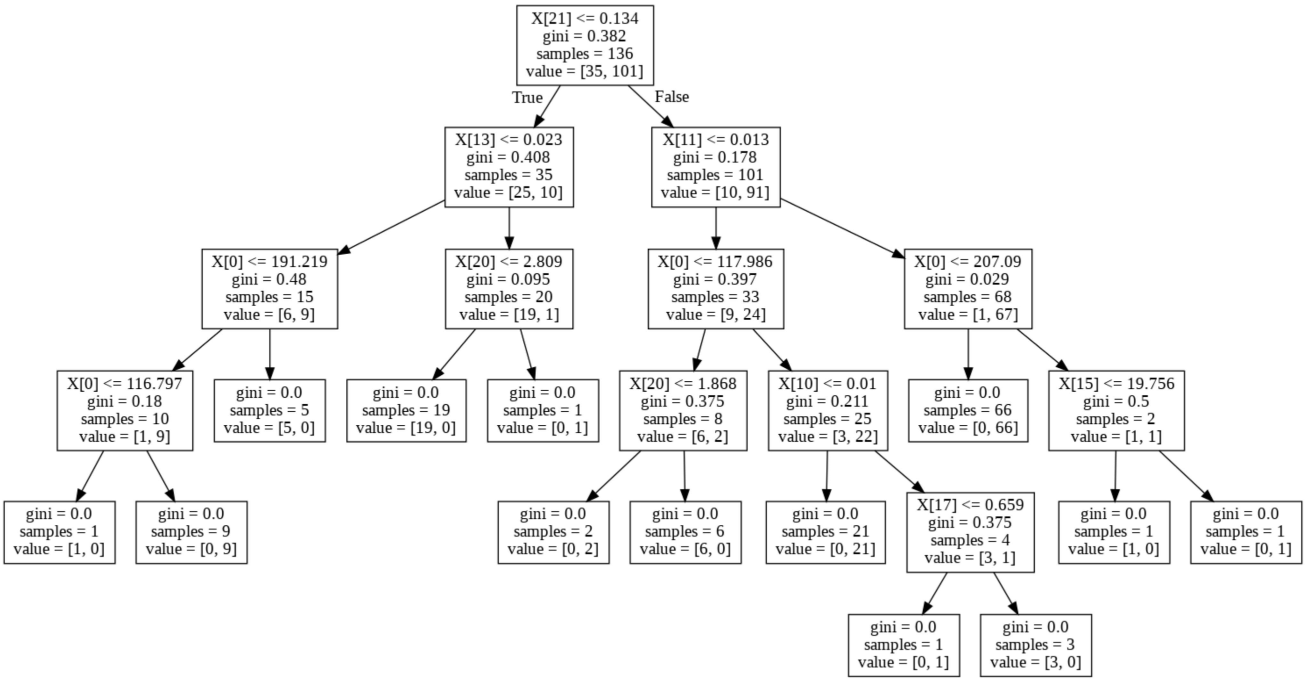
Out[5]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=5, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [6]:
```
print("Mean Accuracy: %.3f%%" %(dectree.score(X_test,Y_test)*100))
```

```
Mean Accuracy: 89.831%
```

In [7]:
```
dot_data=export_graphviz(dectree)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Out[7]:



Decision Tree Classifier (Code)

In [0]:
```python
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
        file = open(filename, "rt")
        lines = reader(file)
        dataset = list(lines)
        return dataset
```

In [0]:
```python
# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
        dataset_split = list()
        dataset_copy = list(dataset)
        fold_size = int(len(dataset) / n_folds)
        for i in range(n_folds):
                fold = list()
                while len(fold) < fold_size:
                        index = randrange(len(dataset_copy))
                        fold.append(dataset_copy.pop(index))
                dataset_split.append(fold)
        return dataset_split
```

In [0]:
```
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
	folds = cross_validation_split(dataset, n_folds)
	scores = list()
	for fold in folds:
		train_set = list(folds)
		train_set.remove(fold)
		train_set = sum(train_set, [])
		test_set = list()
		for row in fold:
			row_copy = list(row)
			test_set.append(row_copy)
			row_copy[-1] = None
		predicted = algorithm(train_set, test_set, *args)
		actual = [row[-1] for row in fold]
		accuracy = accuracy_metric(actual, predicted)
		scores.append(accuracy)
	return scores
```

In [0]:
```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
	left, right = list(), list()
	for row in dataset:
		if row[index] < value:
			left.append(row)
		else:
			right.append(row)
	return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
	# count all samples at split point
	n_instances = float(sum([len(group) for group in groups]))
	# sum weighted Gini index for each group
	gini = 0.0
	for group in groups:
		size = float(len(group))
		# avoid divide by zero
		if size == 0:
			continue
		score = 0.0
		# score the group based on the score for each class
		for class_val in classes:
			p = [row[-1] for row in group].count(class_val) / size
			score += p * p
		# weight the group score by its relative size
		gini += (1.0 - score) * (size / n_instances)
	return gini
```

In [0]:

```
# Select the best split point for a dataset
def get_split(dataset):
	class_values = list(set(row[-1] for row in dataset))
	b_index, b_value, b_score, b_groups = 999, 999, 999, None
	for index in range(len(dataset[0])-1):
		for row in dataset:
			groups = test_split(index, row[index], dataset)
			gini = gini_index(groups, class_values)
			if gini < b_score:
				b_index, b_value, b_score, b_groups = index, row[inde
x], gini, groups
	return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
	outcomes = [row[-1] for row in group]
	return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
	left, right = node['groups']
	del(node['groups'])
	# check for a no split
	if not left or not right:
		node['left'] = node['right'] = to_terminal(left + right)
		return
	# check for max depth
	if depth >= max_depth:
		node['left'], node['right'] = to_terminal(left), to_terminal(right)
		return
	# process left child
	if len(left) <= min_size:
		node['left'] = to_terminal(left)
	else:
		node['left'] = get_split(left)
		split(node['left'], max_depth, min_size, depth+1)
	# process right child
	if len(right) <= min_size:
		node['right'] = to_terminal(right)
	else:
		node['right'] = get_split(right)
		split(node['right'], max_depth, min_size, depth+1)
```

In [0]:

```python
# Build a decision tree
def build_tree(train, max_depth, min_size):
	root = get_split(train)
	split(root, max_depth, min_size, 1)
	return root

# Make a prediction with a decision tree
def predict(node, row):
	if row[node['index']] < node['value']:
		if isinstance(node['left'], dict):
			return predict(node['left'], row)
		else:
			return node['left']
	else:
		if isinstance(node['right'], dict):
			return predict(node['right'], row)
		else:
			return node['right']

# Classification and Regression Tree Algorithm
def decision_tree(train, test, max_depth, min_size):
	tree = build_tree(train, max_depth, min_size)
	predictions = list()
	for row in test:
		prediction = predict(tree, row)
		predictions.append(prediction)
	return(predictions)
```

In [15]:

```python
# Test CART on Parkinson Dataset
seed(1)
# load and prepare data
filename = 'parkinsons_mod.csv'
dataset = load_csv(filename)

# evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 10
scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
Scores: [76.92307692307693, 84.61538461538461, 84.61538461538461, 82.05128205128204,
84.61538461538461]
Mean Accuracy: 82.564%
```

Inference

Performance Comparison:

- Classification Accuracy (sklearn model) | 89.831% |
- Classification Accuracy (custom model) | 82.564% |

The performance difference might be due to the parameters and optimization of the sklearn library model as the library functions are optimized better with respect to the data.

In [0]:
```
## 17BCE0136 R.S.Rahul Sai

import pandas as pd
import numpy as np
```

Dataset Description

This dataset was used to develop quantitative regression QSAR models to predict acute aquatic toxicity towards the fish Pimephales promelas (fathead minnow) on a set of 908 chemicals. LC50 data, which is the concentration that causes death in 50% of test fish over a test duration of 96 hours, was used as model response.

In [2]:
```
data=pd.read_csv("qsar_fish_toxicity.csv",sep=';')
data.head()
```

Out[2]:

	CIC0	SM1	GATS1i	NdsCH	NdssC	MLOGP	LC50
0	3.000	0.000	0.938	1	0	2.851	3.513
1	2.620	0.499	0.990	0	0	2.942	4.402
2	2.834	0.134	0.950	0	0	1.591	3.021
3	2.405	0.134	0.843	0	0	1.769	3.210
4	2.728	0.223	0.953	0	0	1.591	2.371

In [0]:
```
X=data.iloc[:,:-1]
Y=data.iloc[:,-1]
from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test=train_test_split(X,Y,test_size=0.3,random_state=0)
```

Decision Tree Regressor (Library)

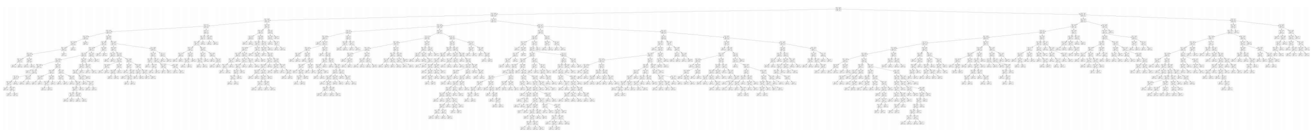
In [0]:
```
from sklearn.tree import DecisionTreeRegressor,export_graphviz
from sklearn.metrics import r2_score,mean_squared_error
from IPython.display import Image
import pydotplus

dectree=DecisionTreeRegressor()
dectree.fit(X_train,Y_train)
Y_pred=dectree.predict(X_test)
```

In [5]:
```
# RMSE value
mean_squared_error(Y_test,Y_pred,squared=False)
```

Out[5]: 1.234072442785528

In [6]:
```
dot_data=export_graphviz(dectree)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Out[6]:



Decision Tree Regressor (Code)

In [0]:
```
class Node:

    def __init__(self, x, y, idxs, min_leaf=5):
        self.x = x
        self.y = y
        self.idxs = idxs
        self.min_leaf = min_leaf
        self.row_count = len(idxs)
        self.col_count = x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()

    def find_varsplit(self):
        for c in range(self.col_count): self.find_better_split(c)
        if self.is_leaf: return
        x = self.split_col
        lhs = np.nonzero(x <= self.split)[0]
        rhs = np.nonzero(x > self.split)[0]
        self.lhs = Node(self.x, self.y, self.idxs[lhs], self.min_leaf)
        self.rhs = Node(self.x, self.y, self.idxs[rhs], self.min_leaf)

    def find_better_split(self, var_idx):

        x = self.x.values[self.idxs, var_idx]

        for r in range(self.row_count):
            lhs = x <= x[r]
            rhs = x > x[r]
            if rhs.sum() < self.min_leaf or lhs.sum() < self.min_leaf: continue

            curr_score = self.find_score(lhs, rhs)
            if curr_score < self.score:
                self.var_idx = var_idx
                self.score = curr_score
                self.split = x[r]

    def find_score(self, lhs, rhs):
        y = self.y[self.idxs]
        lhs_std = y[lhs].std()
        rhs_std = y[rhs].std()
        return lhs_std * lhs.sum() + rhs_std * rhs.sum()

    @property
    def split_col(self): return self.x.values[self.idxs,self.var_idx]

    @property
    def is_leaf(self): return self.score == float('inf')

    def predict(self, x):
        return np.array([self.predict_row(xi) for xi in x])

    def predict_row(self, xi):
        if self.is_leaf: return self.val
        node = self.lhs if xi[self.var_idx] <= self.split else self.rhs
        return node.predict_row(xi)
```

In [0]:
```
class CDecisionTreeRegressor:

  def fit(self, X, y, min_leaf = 5):
    self.dtree = Node(X, y, np.array(np.arange(len(y))), min_leaf)
    return self

  def predict(self, X):
    return self.dtree.predict(X.values)
```

In [9]:
```
dectreeN=CDecisionTreeRegressor().fit(X_train,Y_train)
Y_pred=dectreeN.predict(X_test)
```

```
/usr/local/lib/python3.6/dist-packages/pandas/core/series.py:1146: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-
reindex-listlike
  return self.loc[key]
```

In [10]:
```
# RMSE value
mean_squared_error(Y_test,Y_pred,squared=False)
```

Out[10]:
```
1.9435314927330645
```

Inference

Performance Comparison on basis of RMSE(Root Mean Square Error):

- Regression RMSE Value (sklearn model) | 1.2340 |
- Regression RMSE Value (custom model) | 1.9435 |

The performance difference might be due to the parameters and optimization of the sklearn library model as the library functions are optimized better with respect to the data.