In [1]:
```
from sklearn.model_selection import cross_val_score
from IPython.display import Image
from sklearn import tree
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import mean_squared_error
```

```
/home/nimloth/anaconda3/envs/python27/lib/python2.7/site-packages/sklearn/cross_validation.py:44: Deprecatio
nWarning: This module was deprecated in version 0.18 in favor of the model_selection module into which all t
he refactored classes and functions are moved. Also note that the interface of the new CV iterators are diff
erent from that of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

Сначала мне что-то ударило в голову и я написала классификатор :(

In [2]:

```
class DecisionTreeClassifier:

    def __init__(self, max_depth):
        self.max_depth = max_depth
        if self.max_depth != 1:
            self.L = DecisionTreeClassifier(max_depth=self.max_depth - 1)
            self.R = DecisionTreeClassifier(max_depth=self.max_depth - 1)

    def advanced_log(self, x):
        if x == 0:
            return 0
        else:
            return math.log(x)

    def gini(self, A_p_0, A_p_1, B_p_0, B_p_1):
        N = A_p_0 + A_p_1 + B_p_0 + B_p_1
        L = A_p_0 + A_p_1
        R = B_p_0 + B_p_1
        if L == 0 or R == 0:
            return 1
        A_p_0 = A_p_0 / L
        A_p_1 = A_p_1 / L
        B_p_0 = B_p_0 / R
        B_p_1 = B_p_1 / R
        g = L / N * (- A_p_0 * self.advanced_log(A_p_0) - A_p_1 * self.advanced_log(A_p_1))
        + R / N * (- B_p_0 * self.advanced_log(B_p_0) - B_p_1 * self.advanced_log(B_p_1))
        return g

    def fit(self, train_data, train_target):
        class0 = 0
        class1 = 0
        for elem in train_target:
            if elem == 0:
                class0 = class0 + 1
            else:
                class1 = class1 + 1
        if class0 >= class1:
            self.ans = 0
        else:
            self.ans = 1

        if self.max depth == 1:
```

```
            return

        min_g = 1
        for ith_arg in range(len(train_data[0])):
            for jth_threshold in range(len(train_data)):
                threshold = train_data[jth_threshold][ith_arg]
                A_p_0 = A_p_1 = B_p_0 = B_p_1 = 0
                for jth_elem in range(len(train_data)):
                    if train_data[jth_elem][ith_arg] <= threshold:
                        if train_target[jth_elem] == 0:
                            A_p_0 = A_p_0 + 1
                        else:
                            A_p_1 = A_p_1 + 1
                    else:
                        if train_target[jth_elem] == 0:
                            B_p_0 = B_p_0 + 1
                        else:
                            B_p_1 = B_p_1 + 1
                g = self.gini(A_p_0, A_p_1, B_p_0, B_p_1)
                if g < min_g:
                    self.threshold = threshold
                    self.arg = ith_arg
                    min_g = g
        if min_g == 1:
            return
        A_train = []
        A_target = []
        B_train = []
        B_target = []
        for jth_elem in range(len(train_data)):
            if train_data[jth_elem][self.arg] <= self.threshold:
                A_train.append(train_data[jth_elem])
                A_target.append(train_target[jth_elem])
            else:
                B_train.append(train_data[jth_elem])
                B_target.append(train_target[jth_elem])
        self.L.fit(A_train, A_target)
        self.R.fit(B_train, B_target)

    def predict_for_one(self, test_data):
        if hasattr(self, 'threshold'):
            if test_data[self.arg] <= self.threshold:
```

```
                return self.L.predict_for_one(test_data)
            else:
                return self.R.predict_for_one(test_data)
        else:
            if hasattr(self, 'ans'):
                return self.ans
            else:
                print "Node does not have answer"

    def predict(self, test_data):
        ans = []
        for elem in test_data:
            ans.append(self.predict_for_one(elem))
        return ans
```

Проверила его на выборке из задания 2.

In [3]:
```
samples = pd.read_csv('german_credit.csv')
N = samples.shape[0]
y = samples['Creditability'].reshape(1, N)[0]
x = np.array(samples)
x = np.array([elem[1:] for elem in x]).reshape(N, 20)
```

```
/home/nimloth/anaconda3/envs/python27/lib/python2.7/site-packages/ipykernel/__main__.py:3: FutureWarning: re
shape is deprecated and will raise in a subsequent release. Please use .values.reshape(...) instead
  app.launch_new_instance()
```

In [4]:
```
tree = DecisionTreeClassifier(max_depth=4)
```

In [5]:
```
train_data, test_data, train_target, test_target = train_test_split(x, y,
                                                                    test_size = 0.2)
```

In [6]:
```
tree.fit(train_data, train_target)
```

In [7]:
```
test_predictions = tree.predict(test_data)
```

In [8]:
```
print accuracy_score(test_predictions, test_target)
```

```
0.725
```

Вау! Он классно работает! Прямо как во втором задании точность.

Немного переписав решающее дерево получим класс для решения задачи регрессии.

In [73]:

```
#основные атрибуты это self.threshold, self.arg - по какому признаку и порогу делим вершину
#атрибут self.ans отвечает за ответ в данной вершине
#атрибуты self.L, self.R содержат в себе левое и правле поддеревья
class DecisionTreeRegressor:
    #инициалицируем все дерево, используя глубину
    def __init__(self, max_depth):
        self.max_depth = max_depth
        if self.max_depth != 1:
            self.L = DecisionTreeRegressor(max_depth=self.max_depth - 1)
            self.R = DecisionTreeRegressor(max_depth=self.max_depth - 1)
    #взвешенная функция нехорошести
    def variance(self, A, B):
        N = len(A) + len(B)
        L = len(A)
        R = len(B)
        if L == 0 or R == 0:
            return 100000000
        g = L / N * np.array(A).var()
        + R / N * np.array(B).var()
        return g

    def fit(self, train_data, train_target):

        self.ans = np.array(train_target).mean()

        if self.max_depth == 1:
            return

        #перебираем все признаки и пороги в поиске наименьшей функции нехорошести
        #устанавливаем атрибуты self.arg, self.thresholf
        min_g = 100000000
        for ith_arg in range(len(train_data[0])):
            for jth_threshold in range(len(train_data)):
                threshold = train_data[jth_threshold][ith_arg]
                A = []
                B = []
                for jth_elem in range(len(train_data)):
                    if train_data[jth_elem][ith_arg] <= threshold:
                        A.append(train_target[jth_elem])
                    else:
                        B.append(train_target[jth_elem])
                g = self.variance(A, B)
```

```
                if g < min_g:
                    self.threshold = threshold
                    self.arg = ith_arg
                    min_g = g

        if min_g == 100000000:
            return

        #делим выборку на 2 вершины и запускаемся от них рекурсивно
        A_train = []
        A_target = []
        B_train = []
        B_target = []
        for jth_elem in range(len(train_data)):
            if train_data[jth_elem][self.arg] <= self.threshold:
                A_train.append(train_data[jth_elem])
                A_target.append(train_target[jth_elem])
            else:
                B_train.append(train_data[jth_elem])
                B_target.append(train_target[jth_elem])
        self.L.fit(A_train, A_target)
        self.R.fit(B_train, B_target)

    #пропихивает выборку по вершинам решающего дерева пока может, иначе выдает ответ в вершине
    def predict_for_one(self, test_data):
        if hasattr(self, 'threshold'):
            if test_data[self.arg] <= self.threshold:
                return self.L.predict_for_one(test_data)
            else:
                return self.R.predict_for_one(test_data)
        else:
            if hasattr(self, 'ans'):
                return self.ans
            else:
                print "Node does not have answer"

    def predict(self, test_data):
        ans = []
        for elem in test_data:
            ans.append(self.predict_for_one(elem))
        return ans
```

Проверим точность на датасете Бостон.

In [88]:
```
from sklearn.datasets import load_boston
```

In [96]:
```
samples = load_boston()
x = samples.data
y = samples.target
```

In [132]:
```
tree = DecisionTreeRegressor(max_depth=4)
```

In [133]:
```
train_data, test_data, train_target, test_target = train_test_split(x, y,
                                                                    test_size = 0.25)
```

In [134]:
```
tree.fit(train_data, train_target)
```

In [135]:
```
test_predictions = tree.predict(test_data)
```

In [136]:
```
print mean_squared_error(test_predictions, test_target)
```

```
46.0218550283
```

Неплохо... Но могло бы быть и лучше...