```
# 1. В коде из методички реализуйте один или несколько из критериев останова
# (количество листьев, количество используемых признаков, глубина дерева и т.д.)

import matplotlib.pyplot as plt
import random


from matplotlib.colors import ListedColormap
from sklearn import datasets

import numpy as np

# сгенерируем данные
classification_data, classification_labels = datasets.make_classification(n_features=4
, n_informative=2,
                                                                          n_classes=2,
n_redundant=0,

n_clusters_per_class=1, random_state=5)

# визуализируем сгенерированные данные

colors = ListedColormap(['red', 'blue'])
light_colors = ListedColormap(['lightcoral', 'lightblue'])

plt.figure(figsize=(8, 8))
plt.scatter(list(map(lambda x: x[0], classification_data)), list(map(lambda x: x[1],
classification_data)),
            c=classification_labels, cmap=colors)
```

```
# Реализуем класс узла

class Node:

    def __init__(self, index, t, true_branch, false_branch, **kwargs):
        self.index = index  # индекс признака, по которому ведется сравнение с порогом
в этом узле
        self.t = t  # значение порога
        self.level = kwargs['count_levels']
        self.true_branch = true_branch  # поддерево, удовлетворяющее условию в узле
        self.false_branch = false_branch  # поддерево, не удовлетворяющее условию в
узле


# И класс терминального узла (листа)

class Leaf:

    def __init__(self, data, labels, **kwargs):
        self.data = data
        self.labels = labels
        self.prediction = self.predict()
        self.num_leaf = kwargs['count_leaf']

    def predict(self):
        # подсчет количества объектов разных классов
        classes = {}  # сформируем словарь "класс: количество объектов"
```

```
        for label in self.labels:
            if label not in classes:
                classes[label] = 0
            classes[label] += 1
        #  найдем класс, количество объектов которого будет максимальным в этом листе и
 вернем его
        prediction = max(classes, key=classes.get)
        return prediction

    # Расчет критерия Джини


def gini(labels):
    #  подсчет количества объектов разных классов
    classes = {}
    for label in labels:
        if label not in classes:
            classes[label] = 0
        classes[label] += 1

    #  расчет критерия
    impurity = 1
    for label in classes:
        p = classes[label] / len(labels)
        impurity -= p ** 2

    return impurity
```

```
# Расчет качества

def quality(left_labels, right_labels, current_gini):
    # доля выбоки, ушедшая в левое поддерево
    p = float(left_labels.shape[0]) / (left_labels.shape[0] + right_labels.shape[0])

    return current_gini - p * gini(left_labels) - (1 - p) * gini(right_labels)


# Разбиение датасета в узле

def split(data, labels, index, t):
    left = np.where(data[:, index] <= t)
    right = np.where(data[:, index] > t)

    true_data = data[left]
    false_data = data[right]
    true_labels = labels[left]
    false_labels = labels[right]

    return true_data, false_data, true_labels, false_labels


# Нахождение наилучшего разбиения

def find_best_split(data, labels, *args, **kwargs):
    #  обозначим минимальное количество объектов в узле
    min_leaf = 5
```

```
    current_gini = gini(labels)

    best_quality = 0
    best_t = None
    best_index = None

    n_features = data.shape[1]
    # Ограничиваем фитчи, фитчи будут браться от 0 до max_features
    if n_features > kwargs['max_features']:
        n_features = kwargs['max_features']

    for index in range(n_features):
        # будем проверять только уникальные значения признака, исключая повторения
        t_values = np.unique([row[index] for row in data])

        for t in t_values:
            true_data, false_data, true_labels, false_labels = split(data, labels,
index, t)
            #  пропускаем разбиения, в которых в узле остается менее 5 объектов
            if len(true_data) < min_leaf or len(false_data) < min_leaf:
                continue

            current_quality = quality(true_labels, false_labels, current_gini)

            #  выбираем порог, на котором получается максимальный прирост качества
            if current_quality > best_quality:
                best_quality, best_t, best_index = current_quality, t, index

    return best_quality, best_t, best_index
```

```
count_leaf = 0
# Построение дерева с помощью рекурсивной функции
def build_tree(data, labels, *args, **kwargs):
    # Настраиваем первое вхождение в рекурсию
    global count_leaf

    if 'count_levels' not in kwargs:
        kwargs['count_levels'] = 1
    if 'max_features' not in kwargs:
        kwargs['max_features'] = 100

    quality, t, index = find_best_split(data, labels, *args, **kwargs)


    #  Базовый случай - прекращаем рекурсию, когда нет прироста в качества
    # Если превышено количество листов, прекращаем рекурсию
    if quality == 0:
        count_leaf += 1
        return Leaf(data, labels, count_leaf=count_leaf)

    true_data, false_data, true_labels, false_labels = split(data, labels, index, t)

    # Рекурсивно строим два поддерева
    if kwargs['count_levels'] >= kwargs['max_levels'] or count_leaf >= kwargs['max_leaf
']:
        # Если количество уровней превышено возвращаем лист вместо NODE
        count_leaf += 1
        return Leaf(data, labels, count_leaf=count_leaf)
```

```
    else:
        kwargs['count_levels'] += 1
        true_branch = build_tree(true_data, true_labels, *args, **kwargs)
        false_branch = build_tree(false_data, false_labels, *args, **kwargs)

        # Возвращаем класс узла со всеми поддеревьями, то есть целого дерева
        return Node(index, t, true_branch, false_branch, count_levels=kwargs['
count_levels'])




def classify_object(obj, node, *args, **kwargs):
    #  Останавливаем рекурсию, если достигли листа
    if isinstance(node, Leaf):
        answer = node.prediction
        return answer

    if obj[node.index] <= node.t:
        return classify_object(obj, node.true_branch)
    else:
        return classify_object(obj, node.false_branch)


def predict(data, tree):
    classes = []
    for obj in data:
        prediction = classify_object(obj, tree)
        classes.append(prediction)
```

```
    return classes


# Разобьем выборку на обучающую и тестовую

from sklearn import model_selection

train_data, test_data, train_labels, test_labels = model_selection.train_test_split(
classification_data,

classification_labels,

test_size=0.3,

random_state=1)
max_features = 2
# Построим дерево по обучающей выборке
my_tree = build_tree(train_data, train_labels, max_leaf=4, max_features=max_features,
max_levels=2)


# Напечатаем ход нашего дерева
def print_tree(node, spacing=""):
    # Если лист, то выводим его прогноз
    if isinstance(node, Leaf):
        print(spacing + "Прогноз:", node.prediction)
        print(spacing + "Номер листа:", node.num_leaf)
        return
```

```
    # Выведем значение индекса и порога на этом узле
    print(spacing + 'Индекс', str(node.index))
    print(spacing + 'Порог', str(node.t))
    print(spacing + 'Уровень ветки', str(node.level))

    # Рекурсионный вызов функции на положительном поддереве
    print(spacing + '--> True:')
    print_tree(node.true_branch, spacing + "  ")

    # Рекурсионный вызов функции на положительном поддереве
    print(spacing + '--> False:')
    print_tree(node.false_branch, spacing + "  ")


print_tree(my_tree)
# Получим ответы для обучающей выборки
train_answers = predict(train_data, my_tree)

# И получим ответы для тестовой выборки
answers = predict(test_data, my_tree)


# Введем функцию подсчета точности как доли правильных ответов
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
```

```
# Точность на обучающей выборке
train_accuracy = accuracy_metric(train_labels, train_answers)
print(train_accuracy)

# Точность на тестовой выборке
test_accuracy = accuracy_metric(test_labels, answers)
print(test_accuracy)


# Визуализируем дерево на графике
def get_meshgrid(data, step=.05, border=1.2):
    x_min, x_max = data[:, 0].min() - border, data[:, 0].max() + border
    y_min, y_max = data[:, 1].min() - border, data[:, 1].max() + border
    return np.meshgrid(np.arange(x_min, x_max, step), np.arange(y_min, y_max, step))


plt.figure(figsize=(16, 7))

# график обучающей выборки
plt.subplot(1, 2, 1)
xx, yy = get_meshgrid(train_data)
mesh_predictions = np.array(predict(np.c_[xx.ravel(), yy.ravel()], my_tree)).reshape(xx
.shape)
plt.pcolormesh(xx, yy, mesh_predictions, cmap=light_colors, shading='auto')
plt.scatter(train_data[:, 0], train_data[:, 1], c=train_labels, cmap=colors)
plt.title(f'Train accuracy={train_accuracy:.2f}')
```

```
# график тестовой выборки
plt.subplot(1, 2, 2)
plt.pcolormesh(xx, yy, mesh_predictions, cmap=light_colors, shading='auto')
plt.scatter(test_data[:, 0], test_data[:, 1], c=test_labels, cmap=colors)
plt.title(f'Test accuracy={test_accuracy:.2f}')
plt.show()

# 2. * Реализуйте дерево для задачи регрессии.
# Возьмите за основу дерево, реализованное в методичке, заменив механизм предсказания в
 листе на взятие среднего значения по выборке,
# и критерий Джини на дисперсию значений.
```