Сформировать с помощью sklearn.make_classification датасет из 100 объектов с двумя признаками, обучить случайный лес из 1, 3, 10 и 50 деревьев и визуализировать их разделяющие гиперплоскости на графиках (по подобию визуализации деревьев из предыдущего урока, необходимо только заменить вызов функции predict на tree_vote). Сделать выводы о получаемой сложности гиперплоскости и недообучении или переобучении случайного леса в зависимости от количества деревьев в нем.

In [42]:

```
import matplotlib.pyplot as plt
import random

from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.metrics import accuracy_score

import numpy as np
```

In [43]:

```
# сгенерируем данные, представляющие собой 100 объектов с 2-мя признаками
classification_data, classification_labels = datasets.make_classification(n_samples=100,
                                                      n_features=2, n_informative=2,
                                                      n_classes=2, n_redundant=0,
                                                      n_clusters_per_class=1, random_state=
```

In [44]:

```
classification_data.shape
```

Out[44]:

```
(100, 2)
```

In [45]:

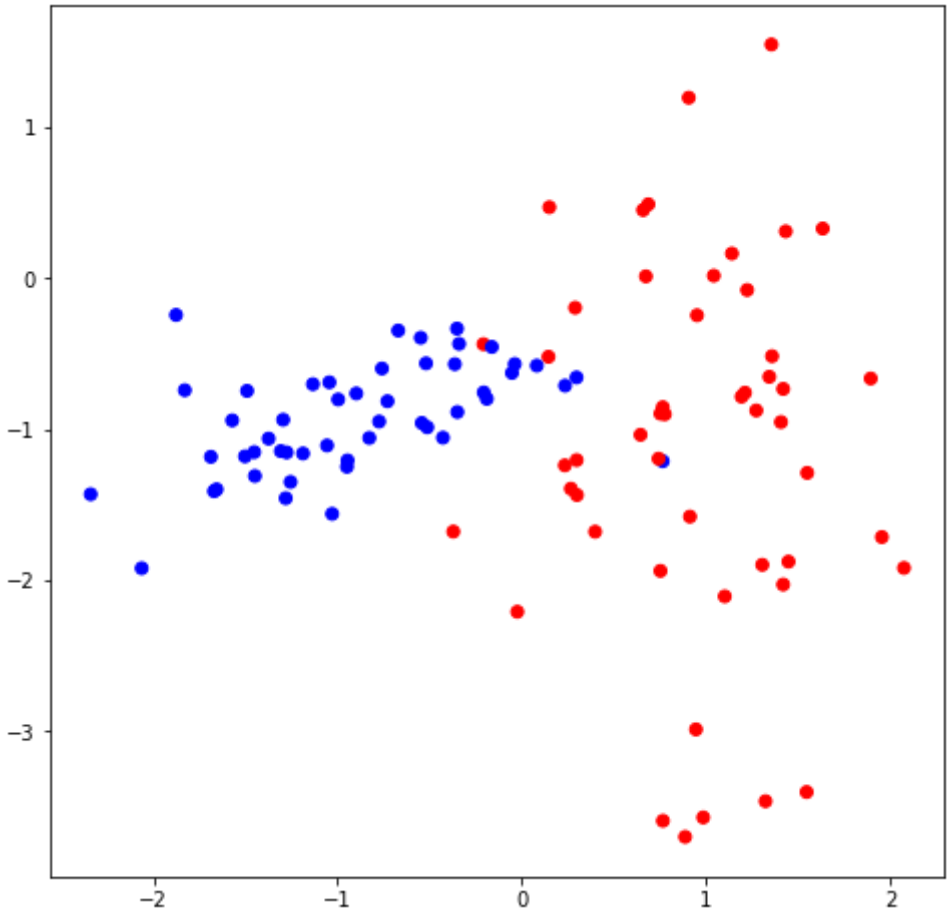
```
# визуализируем сгенерированные данные

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], classi
            c=classification_labels, cmap=colors)
```

Out[45]:

```
<matplotlib.collections.PathCollection at 0x1b2c2897c08>
```



In [46]:

```
random.seed(42)

def get_bootstrap(data, labels, N):
    n_samples = data.shape[0]
    bootstrap = []

    for i in range(N):
        b_data = np.zeros(data.shape)
        b_labels = np.zeros(labels.shape)

        for j in range(n_samples):
            sample_index = random.randint(0, n_samples-1)
            b_data[j] = data[sample_index]
            b_labels[j] = labels[sample_index]
        bootstrap.append((b_data, b_labels))

    return bootstrap
```

In [47]:

```
def get_bootstrap(labels, N):
    return np.random.randint(0, len(labels), size = (N, len(labels)))
```

In [48]:

```

def get_oob(bootstrap_idx: np.array) -> np.array:
    oob_idx = [
        list(set(range(len(bootstrap_idx[0])))-set(bootstrap))
        for bootstrap in bootstrap_idx
    ]
    return np.array(oob_idx)
```

In [49]:

```
def get_subsample(len_sample):
    # будем сохранять не сами признаки, а их индексы
    sample_indexes = [i for i in range(len_sample)]

    len_subsample = int(np.sqrt(len_sample))
    subsample = []

    random.shuffle(sample_indexes)
    for _ in range(len_subsample):
        subsample.append(sample_indexes.pop())

    return subsample
```

In [50]:

```
# Реализуем класс узла

class Node:

    def __init__(self, index, t, true_branch, false_branch):
        self.index = index  # индекс признака, по которому ведется сравнение с порогом в эт
        self.t = t  # значение порога
        self.true_branch = true_branch  # поддерево, удовлетворяющее условию в узле
        self.false_branch = false_branch  # поддерево, не удовлетворяющее условию в узле
```

In [51]:

```
# И класс терминального узла (листа)

class Leaf:

    def __init__(self, data, labels):
        self.data = data
        self.labels = labels
        self.prediction = self.predict()

    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 вер
        prediction = max(classes, key=classes.get)
        return prediction
```

In [52]:

```
# Расчет критерия Шеннона

def calc_entropy(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 = 0
    for label in classes:
        p = classes[label] / len(labels)
        impurity += p*np.log2(p)

    return -impurity
```

In [53]:

```
# 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 * calc_entropy(left_labels) - (1 - p) * calc_entropy(right_labe
```

In [54]:

```
# 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
```

In [55]:

```
# Нахождение наилучшего разбиения

def find_best_split(data, labels):

    #  обозначим минимальное количество объектов в узле
    min_leaf = 1

    current_gini = calc_entropy(labels)

    best_quality = 0
    best_t = None
    best_index = None

    n_features = data.shape[1]

    # выбор индекса из подвыборки длиной sqrt(n_features)
    subsample = get_subsample(n_features)

    for index in subsample:
        # 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
```

In [ ]:

```

```

Здесь мы меняем глубину и кол-во листов

In [56]:

```
# Построение дерева с помощью рекурсивной функции

def build_tree(data, labels):

    quality, t, index = find_best_split(data, labels)

    #  Базовый случай - прекращаем рекурсию, когда нет прироста в качества
    if quality == 0:
        return Leaf(data, labels)

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

    # Рекурсивно строим два поддерева
    true_branch = build_tree(true_data, true_labels)
    false_branch = build_tree(false_data, false_labels)

    # Возвращаем класс узла со всеми поддеревьями, то есть целого дерева
    return Node(index, t, true_branch, false_branch)
```

In [57]:

```
def random_forest(data, labels, n_trees):
    forest, scores = [], []
    bootstrap = get_bootstrap(data, N=n_trees)
    oob = get_oob(bootstrap)


    for idx in bootstrap:
        b_data, b_labels = data[idx], labels[idx]
        forest.append(build_tree(b_data, b_labels))

    for idx in oob:
        oob_data, oob_labels = data[idx], labels[idx]
        for tree in forest:
            y_pred = predict(oob_data, tree)
            score = accuracy_score(oob_labels, y_pred)
        scores.append(score)


    return forest, scores
```

In [58]:

```
# Функция классификации отдельного объекта

def classify_object(obj, node):

    #  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)
```

In [59]:

```
# функция формирования предсказания по выборке на одном дереве

def predict(data, tree):

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

In [60]:

```
# предсказание голосованием деревьев

def tree_vote(forest, data):

    # добавим предсказания всех деревьев в список
    predictions = []
    for tree in forest:
        predictions.append(predict(data, tree))

    # сформируем список с предсказаниями для каждого объекта
    predictions_per_object = list(zip(*predictions))

    # выберем в качестве итогового предсказания для каждого объекта то,
    # за которое проголосовало большинство деревьев
    voted_predictions = []
    for obj in predictions_per_object:
        voted_predictions.append(max(set(obj), key=obj.count))

    return voted_predictions
```

In [61]:

```
# Разобьем выборку на обучающую и тестовую

from sklearn import model_selection

train_data, test_data, train_labels, test_labels = model_selection.train_test_split(classif
                                                                                     classi
                                                                                     test_s
                                                                                     random
```

In [62]:

```
# 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
```

In [63]:

```
models = {}

for n_trees in [1,3,10,50]:
    models[n_trees] = random_forest(train_data, train_labels, n_trees)
    print(f"n_trees = {n_trees}, OOB = {round(np.mean(models[n_trees][1]),3)},accuracy_metr
```

```
n_trees = 1, OOB = 0.833,accuracy_metrics = 94.28571428571428
n_trees = 3, OOB = 0.972,accuracy_metrics = 97.14285714285714
n_trees = 10, OOB = 0.951,accuracy_metrics = 100.0

C:\Users\voron\AppData\Roaming\Python\Python37\site-packages\ipykernel_launc
her.py:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested
sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with diff
erent lengths or shapes) is deprecated. If you meant to do this, you must sp
ecify 'dtype=object' when creating the ndarray.

C:\Users\voron\AppData\Roaming\Python\Python37\site-packages\ipykernel_launc
her.py:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested
sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with diff
erent lengths or shapes) is deprecated. If you meant to do this, you must sp
ecify 'dtype=object' when creating the ndarray.

C:\Users\voron\AppData\Roaming\Python\Python37\site-packages\ipykernel_launc
her.py:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested
sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with diff
erent lengths or shapes) is deprecated. If you meant to do this, you must sp
ecify 'dtype=object' when creating the ndarray.


n_trees = 50, OOB = 0.946,accuracy_metrics = 100.0
```

я думаю переобучение могло быть при обучении на одном дереве

реализовать на графике у меня не получилось

In [ ]:

```
```

In [ ]:

```
```