B [20]:

```
import numpy as np
import matplotlib.pyplot as plt
import random
from matplotlib.colors import ListedColormap
from sklearn import datasets
from collections import Counter
```

B [21]:

```
# сгенерируем данные
X, y = datasets.make_classification(n_features = 2, n_informative = 2,
                                    n_classes = 2, n_redundant=0,
                                    n_clusters_per_class=1, rando
```

B [25]:

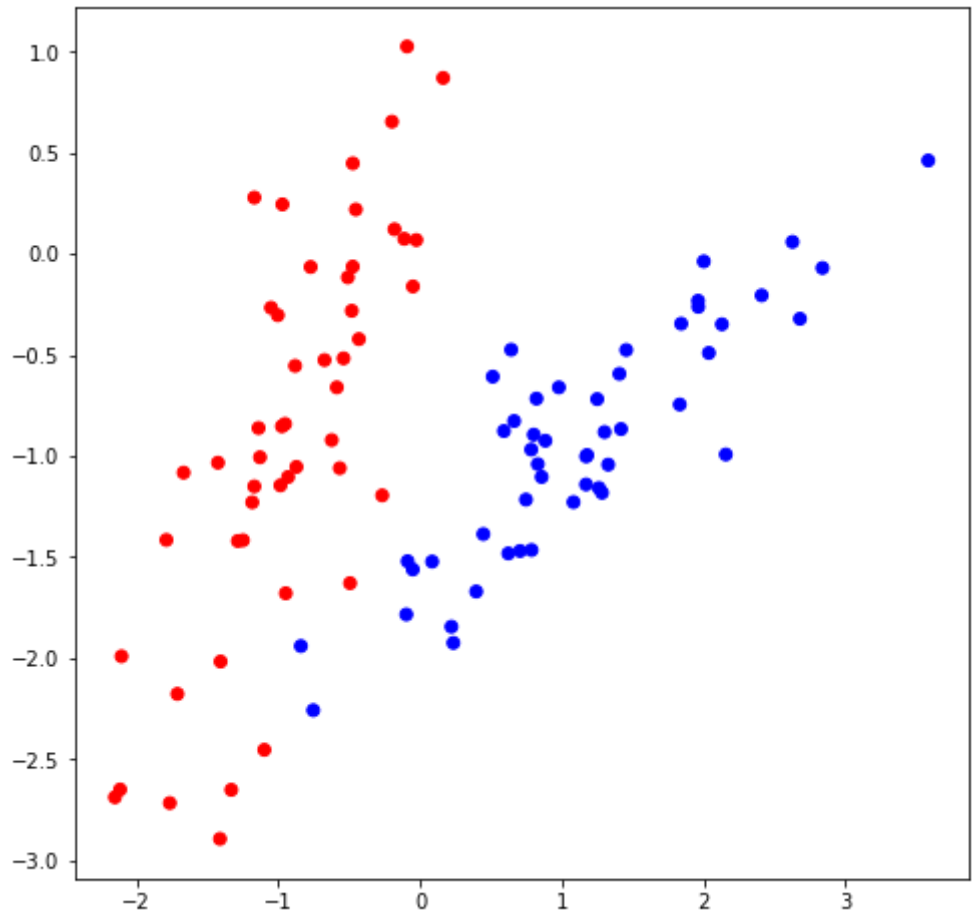
```
# визуализируем сгенерированные данные

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

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

Out[25]:

```
<matplotlib.collections.PathCollection at 0x2c55295b460>
```



В [17]:

```
# 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
```

1. В коде из методички реализуйте один или несколько из критериев останова (кол-во листьев, кол-во используемых признаков, глубина дерева и т.д.)

В [42]:

```
class DecisionTreeClassifier:

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


    # класс терминального узла (листа)
    class Leaf:
        def __init__(self, X, y):
            self.X = X
            self.y = y
            classes = Counter(self.y)
            self.prediction = max(classes, key=classes.get)

    def __init__(self, max_depth=None, quality_gain=None):
        self.max_depth = max_depth
        self.quality_gain = quality_gain

        self.tree = None

    def fit(self, X, y):
        self.tree = self.build_tree(np.array(X), np.array(y))

    def gini(self, y):
        impurity = 1
        for class_ct in Counter(y).values():
            p = class_ct / len(y)
            impurity -= p**2
        return impurity

    # Расчет качества
    def quality(self, 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

        return current_gini - p * self.gini(left_labels) - (1 - p) * self.gini(right

    # Разбиение датасета в узле
    def split(self, X, y, index, t):

        left = np.where(X[:, index] <= t)
        right = np.where(X[:, index] > t)

        true_data = X[left]
        false_data = X[right]
        true_labels = y[left]
        false_labels = y[right]

        return true_data, false_data, true_labels, false_labels

    # Нахождение наилучшего разбиения
    def find_best_split(self, X, y):

        #  обозначим минимальное количество объектов в узле
```

```
        min_leaf = 5

        current_gini = self.gini(y)

        best_quality = 0
        best_t = None
        best_index = None

        n_features = X.shape[1]

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

            for t in t_values:
                true_data, false_data, true_labels, false_labels = self.split(X, y,
                #  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 = self.quality(true_labels, false_labels, current_g

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

        return best_quality, best_t, best_index

    # Построение дерева с помощью рекурсивной функции
    def build_tree(self, X, y, current_depth = 0):

        # выходим если достигнута максимальная глубина дерева
        if self.max_depth is not None and current_depth == self.max_depth:
            return self.Leaf(X, y)

        quality, t, index = self.find_best_split(X, y)

        #  выходим если нет прироста в качества или он меньше заданного
        if quality == 0 or (quality is not None and quality < self.quality_gain):
            return self.Leaf(X, y)

        true_data, false_data, true_labels, false_labels = self.split(X, y, index,

        # Рекурсивно строим два поддерева
        true_branch = self.build_tree(true_data, true_labels, current_depth+1)
        false_branch = self.build_tree(false_data, false_labels, current_depth+1)

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

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

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

```
    def predict(self, X):
        res = [self.classify_object(obj, self.tree) for obj in X]
        return np.array(res)
```

Обучим модель

B [41]:

```
clf_tree = DecisionTreeClassifier(max_depth=3, quality_gain=0.1)
clf_tree.fit(X,y)
y_pred = clf_tree.predict(X)
accuracy_metric(y, y_pred)
```

Out[41]:

```
94.0
```

B [ ]:

```

```