

In [1]:

```
import matplotlib.pyplot as plt
import random

from matplotlib.colors import ListedColormap
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

import numpy as np
import math
```

In [2]:

```
# 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 этом узле
        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

# 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 вернем его
        prediction = max(classes, key=classes.get)
        return prediction
```

In [3]:

```
class RandomForest():

    @property
    def forest_(self):
        return self.forest

    @property
    def n_trees_(self):
        return self.n_trees
```

```

def __init__(self, n_trees = 1, max_depth=5, min_leaf=1, random_state=42, criterion='gini'):
    self.n_trees = n_trees
    self.random_state = random_state
    self.max_depth = max_depth
    self.min_leaf = min_leaf
    self.criterion = criterion
    self.forest = []

def bootstrap(self, X, y):
    n_samples = X.shape[0]
    random.seed(self.random_state)

    bootstrap = []
    for i in range(self.n_trees):
        b_data = np.zeros(X.shape)
        b_labels = np.zeros(y.shape)

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

    return bootstrap

def subsample(self, 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

def calc_criterion(self, y):
    """Расчет критерия информативности"""

    # подсчет количества объектов разных классов
    classes = {}
    for label in y:
        if label not in classes:
            classes[label] = 0
        classes[label] += 1

    if self.criterion == 'gini':
        # Расчет критерия Джини
        impurity = 1
        for label in classes:
            p = classes[label] / len(y)
            impurity -= p ** 2

    return impurity
    elif self.criterion == 'entropy':
        # Расчет критерия энтропии Шеннона

```

```

        impurity = 0
        for label in classes:
            p = classes[label] / len(y)
            impurity += p * math.log(p, 2)

        return impurity * -1
    else:
        return None

def quality(self, left_labels, right_labels, current_criterion):
    """Расчет качества"""

    # Доля выборки, ушедшая в левое поддерево
    p = float(left_labels.shape[0]) / (left_labels.shape[0] + right_labels.shape[0])

    return current_criterion - p * self.calc_criterion(left_labels) - (1 - p) * self.calc_criterion(right_labels)

@staticmethod
def split(X, y, index, t):
    """Разбиение датасета в узле"""

    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

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

    current_criterion = self.calc_criterion(y)

    best_quality = 0
    best_t = None
    best_index = None

    n_features = X.shape[1]

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

    for index in subsample:
        # будем проверять только уникальные значения признака, исключая повторения
        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, index, t)
                # пропускаем разбиения, в которых в узле остается менее 5 объектов
                if len(true_data) < self.min_leaf or len(false_data) < self.min_leaf:
                    continue

            current_criterion = self.calc_criterion(true_labels, false_labels)
            if current_criterion < best_quality:
                best_quality = current_criterion
                best_t = t
                best_index = index

```

```

        current_quality = self.quality(true_labels, false_labels,
current_criterion)

    # 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

def tree(self, X, y):

    # Построение дерева с помощью рекурсивной функции
    def build_tree(X, y, **kwargs):

        # ограничение по глубине дерева
        kwargs['depth'] += 1
        if kwargs['depth'] > self.max_depth:
            return Leaf(X, y)

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

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

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

        # Рекурсивно строим два поддерева
        true_branch = build_tree(true_data, true_labels, depth=kwargs[
'depth'])
        false_branch = build_tree(false_data, false_labels, depth=kwar
gs['depth'])

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

    return build_tree(X, y, depth=0)

def predict(self, X):
    """Предсказание голосованием деревьев"""

    def tree_predict(X, tree):
        """Функция формирования предсказания по выборке на одном дереве"""
        classes = []
        for obj in X:
            prediction = classify_object(obj, tree)
            classes.append(prediction)
        return classes

    def classify_object(obj, node):
        """Функция классификации отдельного объекта"""

```

```

# 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)

# добавим предсказания всех деревьев в список
predictions = []
for tree in self.forest:
    predictions.append(tree_predict(X, 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

def fit(self, X, y):
    bootstrap = self.bootstrap(X, y)

    for b_data, b_labels in bootstrap:
        self.forest.append(self.tree(b_data, b_labels))

```

In [4]:

```

# 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 [5]:

```

def plot(model, X_train, X_test, y_train, y_test):

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

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

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

    xx, yy = get_meshgrid(X_train)

```

```

mesh_predictions = np.array(model.predict(np.c_[xx.ravel(), yy.ravel()])).reshape(xx.shape)

# график обучающей выборки
plt.subplot(1,2,1)
plt.pcolormesh(xx, yy, mesh_predictions, cmap = light_colors)
plt.scatter(X_train[:, 0], X_train[:, 1], c = y_train, cmap = colors)
plt.title('Train')

# график тестовой выборки
plt.subplot(1,2,2)
plt.pcolormesh(xx, yy, mesh_predictions, cmap = light_colors)
plt.scatter(X_test[:, 0], X_test[:, 1], c = y_test, cmap = colors)
plt.title('Test')

```

In [6]:

```

def print_results(model, X_train, X_test, y_train, y_test):

    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    print(f'Количество деревьев: {model.n_estimators_}')

    train_accuracy = accuracy_metric(y_train, y_train_pred)
    print(f'Точность на обучающей выборке: {train_accuracy:.3f}')

    test_accuracy = accuracy_metric(y_test, y_test_pred)
    print(f'Точность на тестовой выборке: {test_accuracy:.3f}\n')

    plot(model, X_train, X_test, y_train, y_test)

```

In [7]:

```

# генерируем данные, представляющие собой 500 объектов с 5-ю признаками
classification_data, classification_labels = make_classification(
    n_samples=100,
    n_features=2,
    n_informative=2,
    n_classes=2,
    n_redundant=0,
    n_clusters_per_class=1,
    random_state=23
)

```

In [8]:

```

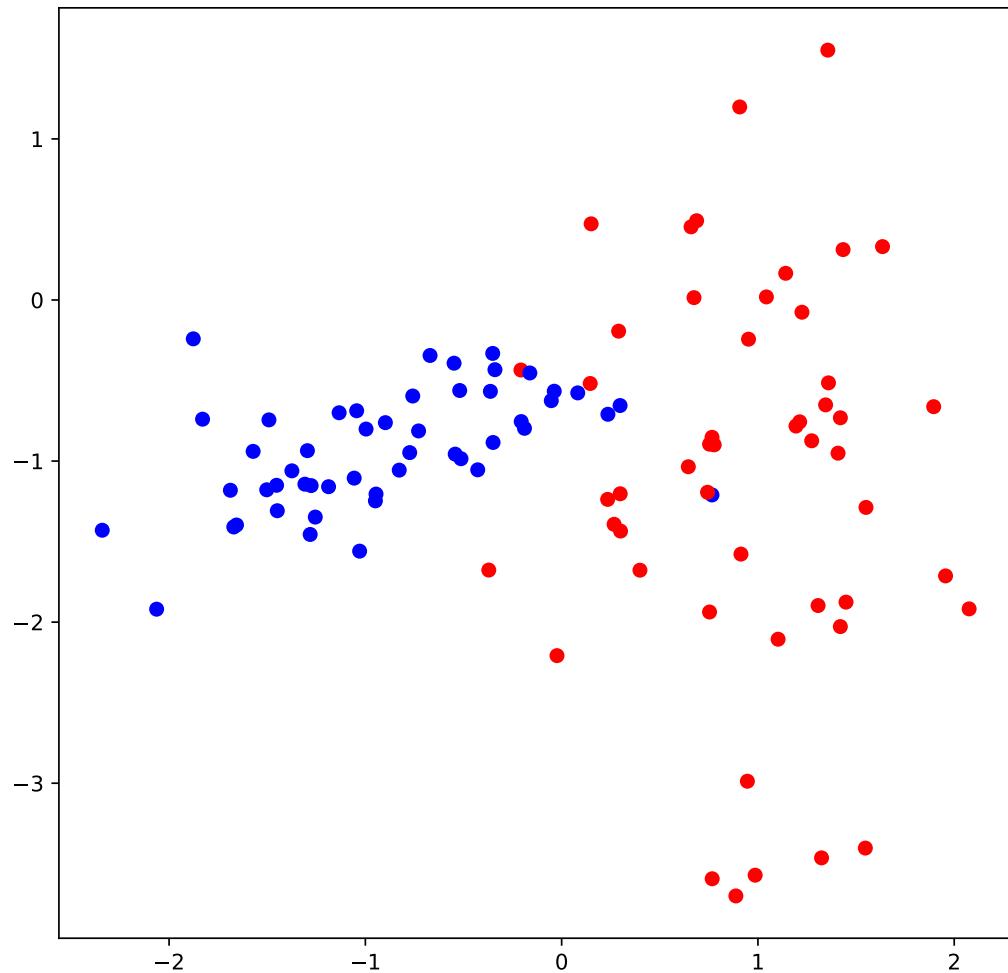
# визуализируем сгенерированные данные
colors = ListedColormap(['red', 'blue'])
light_colors = ListedColormap(['lightcoral', 'lightblue'])

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
)

```

Out[8]:

```
<matplotlib.collections.PathCollection at 0x1a65bbca388>
```



In [9]:

```
# Разобьем выборку на обучающую и тестовую
X_train, X_test, y_train, y_test = train_test_split(
    classification_data,
    classification_labels,
    test_size=0.3,
    random_state=1,
)

parameters = {
    'max_depth': 5,
```

```
'min_leaf': 1,  
'random_state': 42,  
'criterion': 'gini',  
}  
}
```

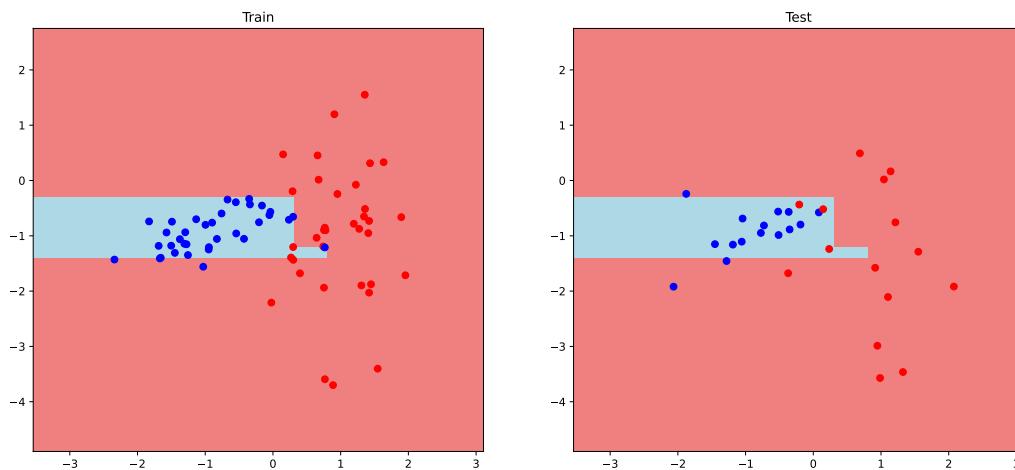
In [10]:

```
clf_1 = RandomForest(  
    n_trees=1,  
    **parameters  
)  
clf_1.fit(X_train, y_train)  
  
print_results(clf_1, X_train, X_test, y_train, y_test)
```

Количество деревьев: 1

Точность на обучающей выборке: 97.143

Точность на тестовой выборке: 80.000



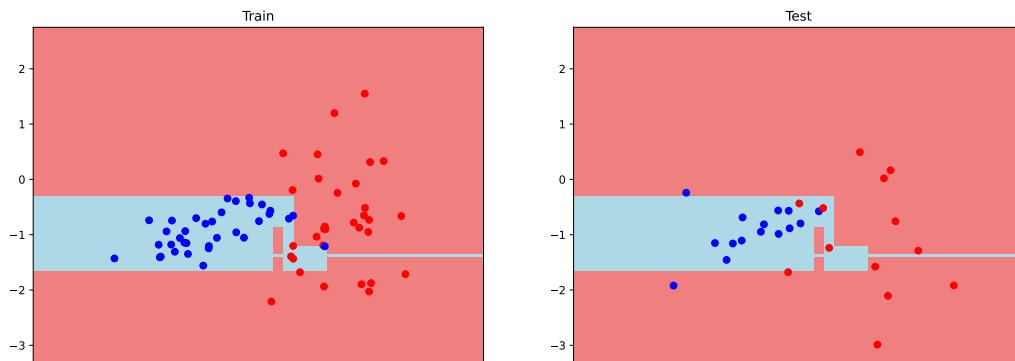
In [11]:

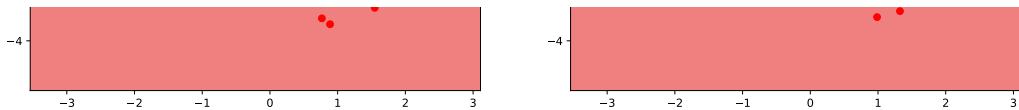
```
clf_3 = RandomForest(  
    n_trees=3,  
    **parameters  
)  
clf_3.fit(X_train, y_train)  
  
print_results(clf_3, X_train, X_test, y_train, y_test)
```

Количество деревьев: 3

Точность на обучающей выборке: 97.143

Точность на тестовой выборке: 80.000





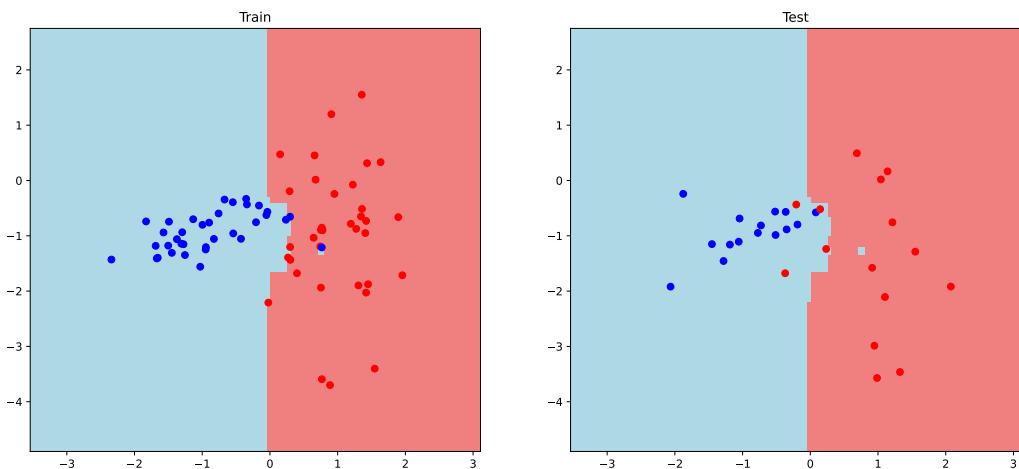
In [12]:

```
clf_10 = RandomForest(  
    n_trees=10,  
    **parameters  
)  
clf_10.fit(X_train, y_train)  
  
print_results(clf_10, X_train, X_test, y_train, y_test)
```

Количество деревьев: 10

Точность на обучающей выборке: 100.000

Точность на тестовой выборке: 86.667



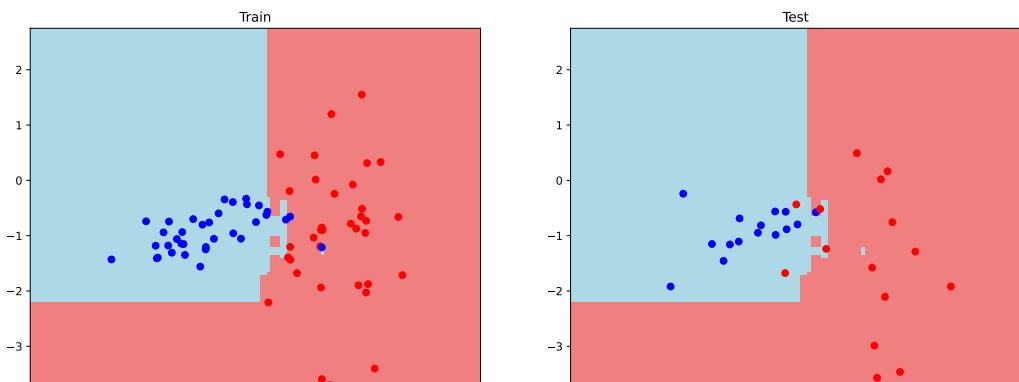
In [13]:

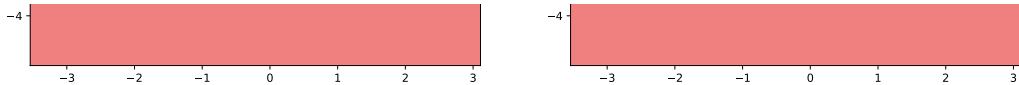
```
clf_50 = RandomForest(  
    n_trees=50,  
    **parameters  
)  
clf_50.fit(X_train, y_train)  
  
print_results(clf_50, X_train, X_test, y_train, y_test)
```

Количество деревьев: 50

Точность на обучающей выборке: 100.000

Точность на тестовой выборке: 86.667





## n\_trees=10 и критерий информативности Шэннона

In [14]:

```
clf = RandomForest(
    n_trees=10,
    max_depth=5,
    min_leaf=1,
    random_state=42,
    criterion='entropy',
)
clf.fit(X_train, y_train)

print_results(clf, X_train, X_test, y_train, y_test)
```

Количество деревьев: 10

Точность на обучающей выборке: 100.000

Точность на тестовой выборке: 86.667

