Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт информационные системы и анализ данных

Центр программной инженерии

**ОТЧЁТ**

к лабораторной работе №1 по дисциплине:

|  |
| --- |
| «Нейросетевые технологии» |
| Применение нейронных сетей для задач классификации и кластеризации |
| наименование темы |

Выполнилстудент ИСТб-21-1 Д.И. Морозов

номер группы подпись И. О. Фамилия

дата

Проверил Доцент Е.А. Осипова

Должность подпись И. О. Фамилия

дата

Иркутск – 2023 г.

# 1 Постановка задачи

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Сгенерировать случайные числа по указанному закону распределения. Из них создается матрица из M строк (наблюдений) и P столбцов (признаков). Сгенерированные строки должны принадлежать одному из N классов за счет прибавления к сгенерированным числам положительной или отрицательной добавки (функция)
2. Все признаки должны быть числовыми и нормализованными (находиться в диапазоне [0,1]). Дополнительно создается классификационный признак.
3. Разделить данные варианта на обучающую и тестирующую выборки.
4. Персептрон. Так как персептрон способен делить объекты только на два класса, то организуем работу итерационно:

4.1. Создать персептрон для отделения 1 класса от всех остальных. Протестировать персептрон по обучающей и тестирующей выборкам и определить процент правильно классифицированных объектов 1 класса.

4.2. Создать персептрон для отделения 2 класса от оставшихся. Протестировать персептрон по обучающей и тестирующей выборкам и определить процент правильно классифицированных объектов 2 класса

4.3. Продолжить, пока не будут разделены все объекты

4.4. Построить графики разделяющей поверхности и визуальное представление результатов тестирования

1. Сеть Кохонена (Kohonen). Вероятностная нейронная сеть (PNN). Сеть с прямым распрост­ра­нением сигнала и обрат­ным распространением ошибки (MLP). Осуществить кластеризацию с помощью конкурирующей сети (слой Кохонена), а затем классификацию с помощью двух трехслойных сетей: вероятностной нейронной сети (PNN) и сети с прямым распрост­ра­нением сигнала и обрат­ным распространением ошибки (MLP). Для каждого вида сети выполнить следующие этапы:

5.1 Создание сети

5.2 Обучение сети на обучающей выборке

5.3 Протестировать сеть отдельно по обучающей и по тестирующей выборке

5.4 Построить визуальное представление результатов тестирования

5.5 Оценить качество классификации в виде матриц

5.6 Выдать число итераций обучения

1. Оформить результаты в виде таблицы

# 2 Ход работы

# 2.1 Подготовка данных

Импорт библиотек для дальнейшей работы.

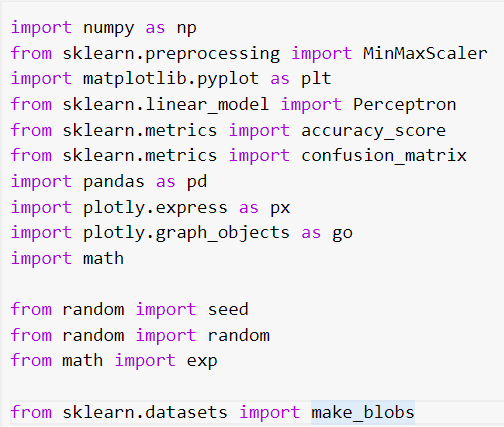


Рисунок 1 – Импорт библиотек

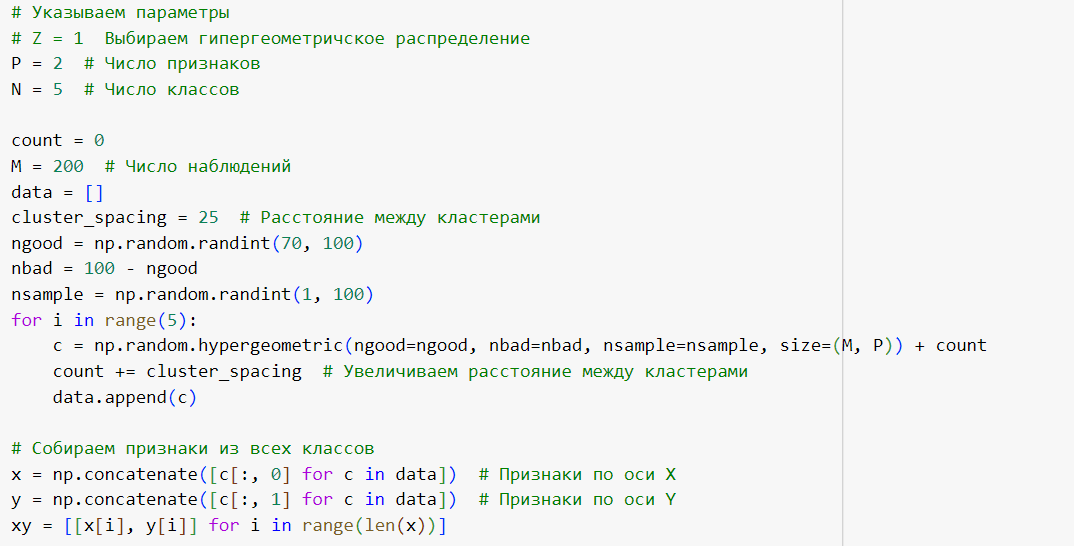


Рисунок 2 – Создание выборки

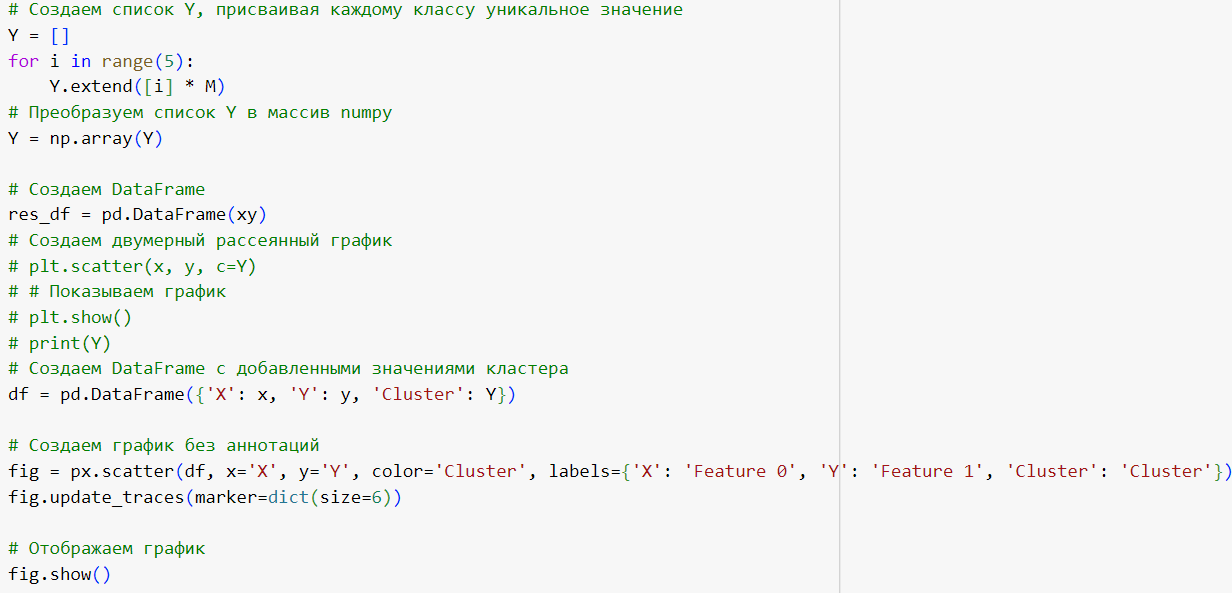


Рисунок 3 – Код для вывода графика значений с их метками

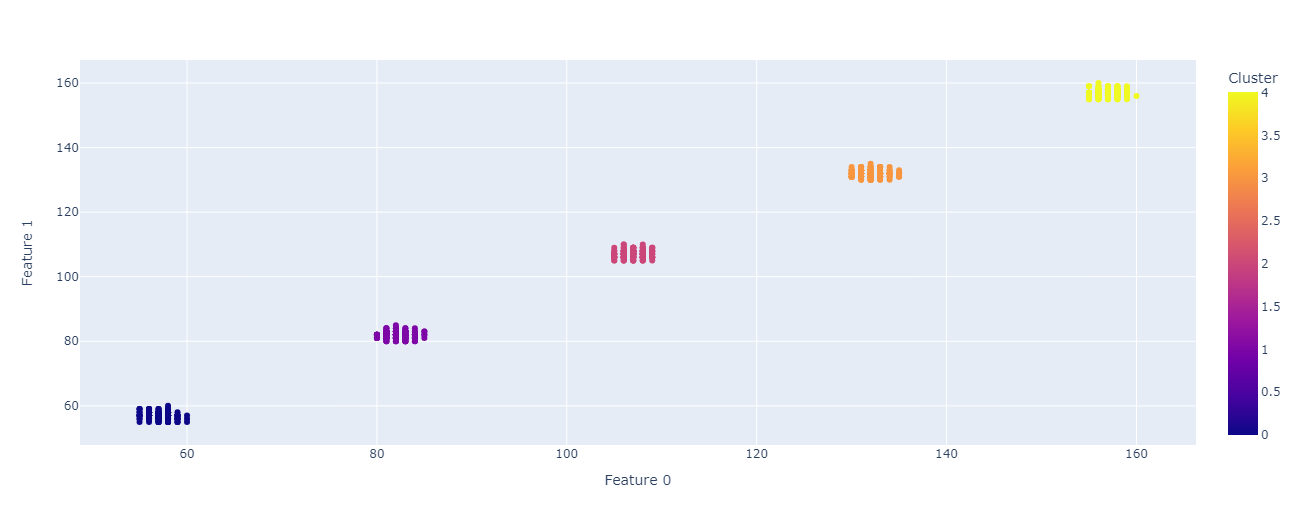


Рисунок 4 – Графики с метками классов

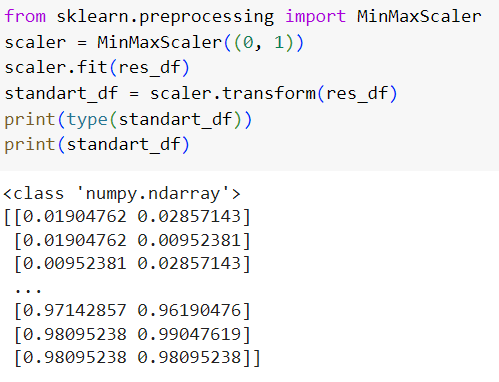


Рисунок 5 – Стандартизация данных, выводом значений и типом данных

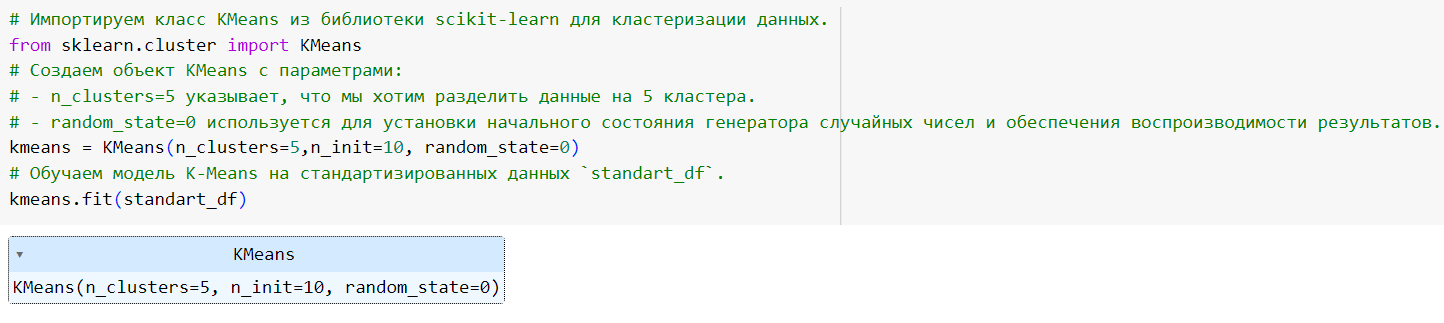


Рисунок 6 – Кластеризация значений и обучение модели

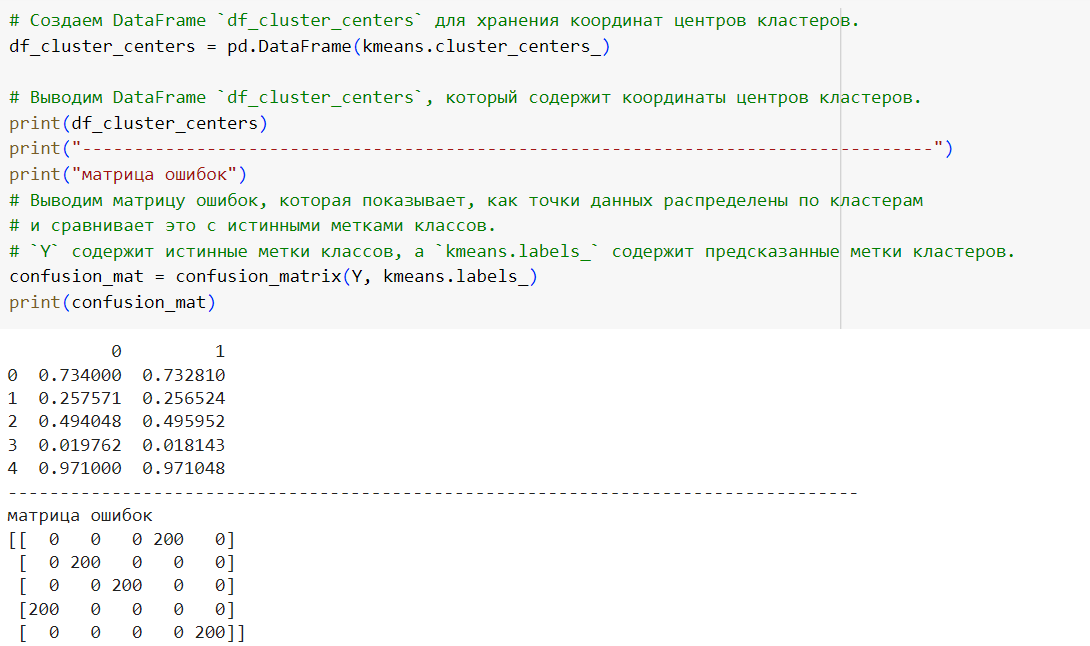


Рисунок 7 – Матрица ошибок и координаты центров кластеров

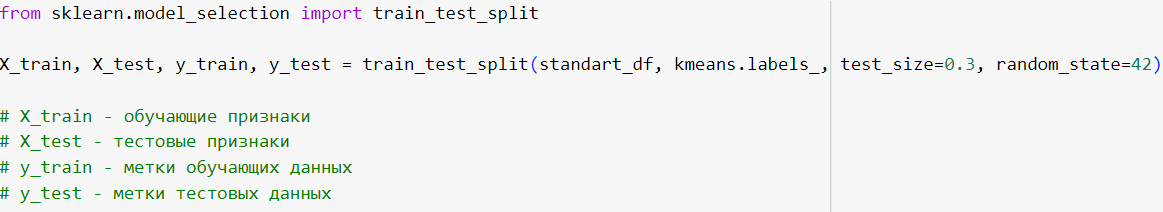


Рисунок 8 – Обучение данных

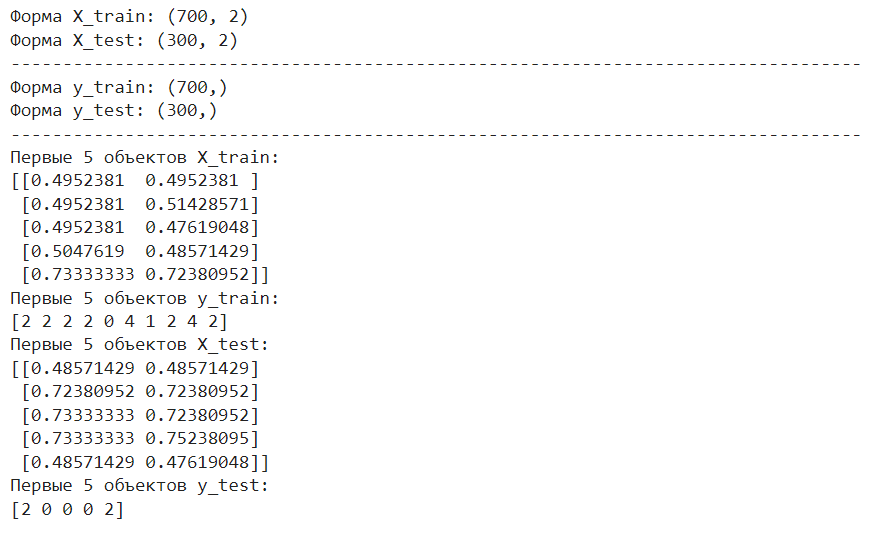


Рисунок 9 – Проверка созданных данных

# 3 Создание персептрона и его работа

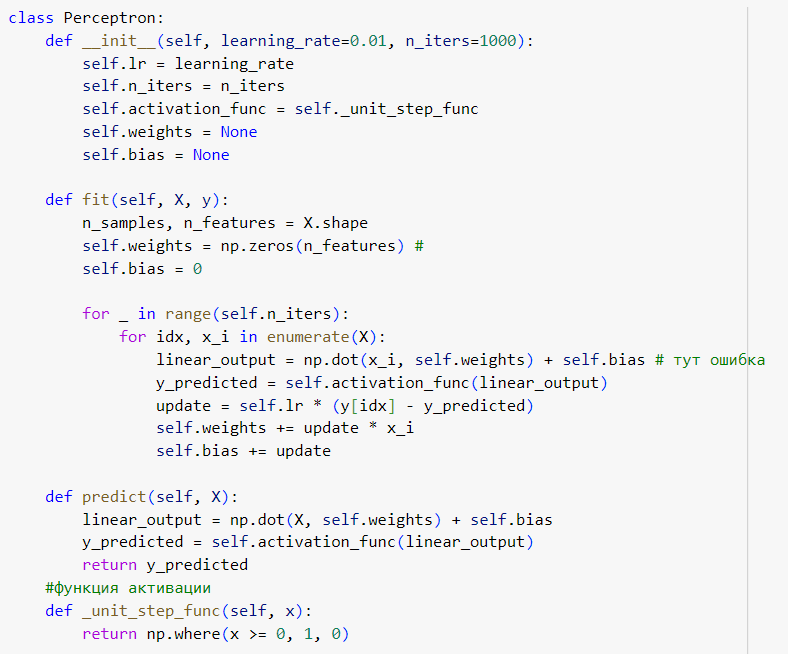


Рисунок 10 – Реализация класса персептрон

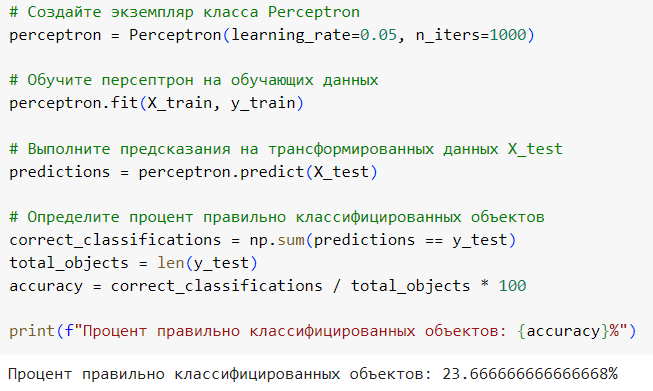
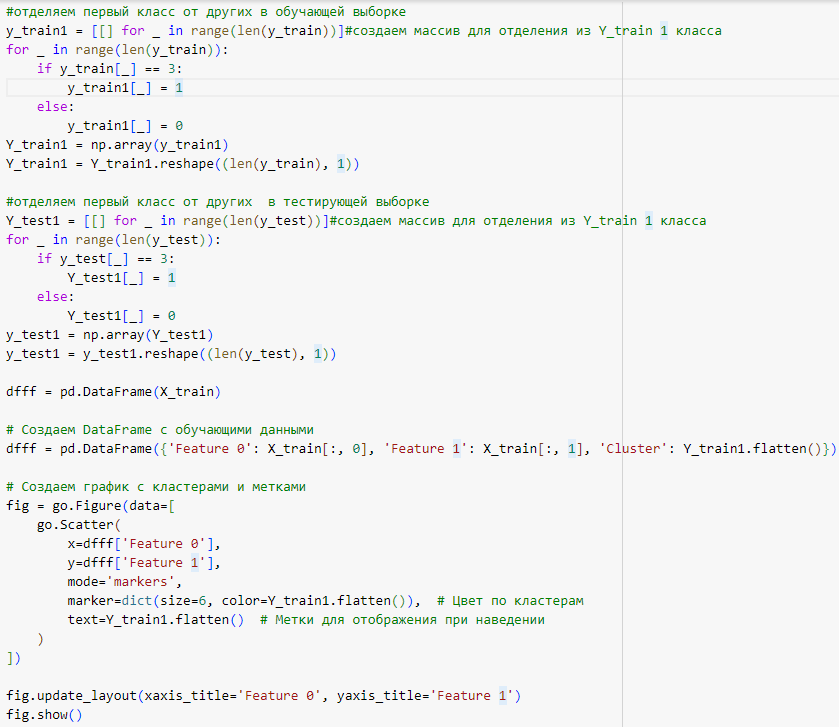


Рисунок 11 – Обучение персептрона на обучающих данных

Рисунок 12 – Отделение класса

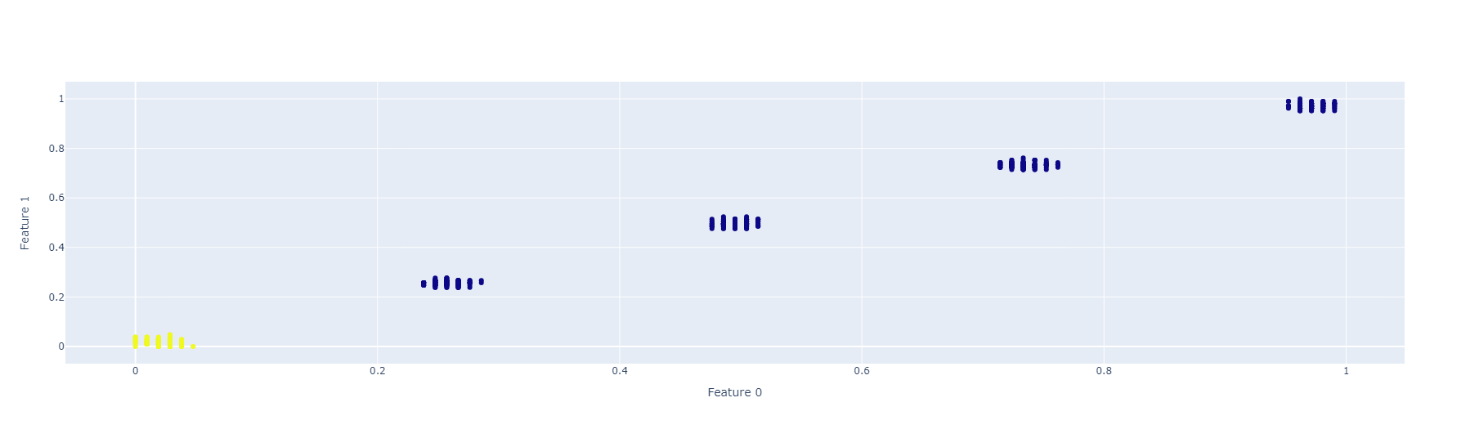


Рисунок 13 – Результат отделения класса

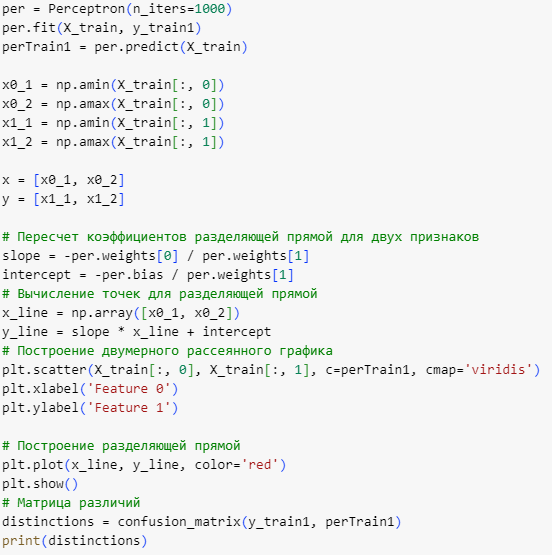


Рисунок 14 – Создание персептрона для удаления класса и построения разделяющей прямой

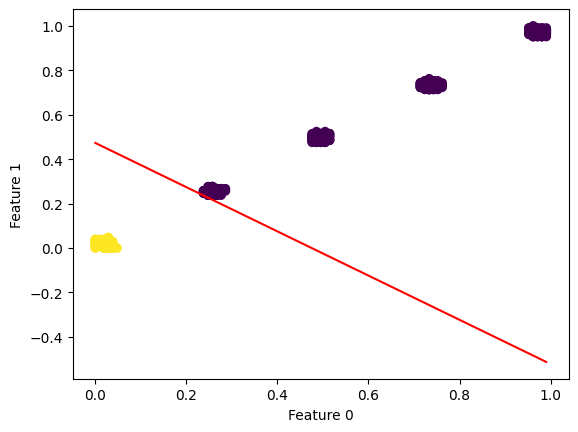


Рисунок 15 –Результат отделения кластера

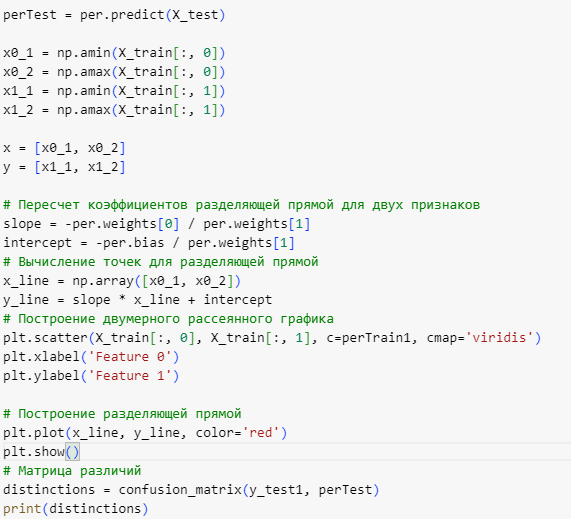


Рисунок 16 – Перерасчет для тестовых данных

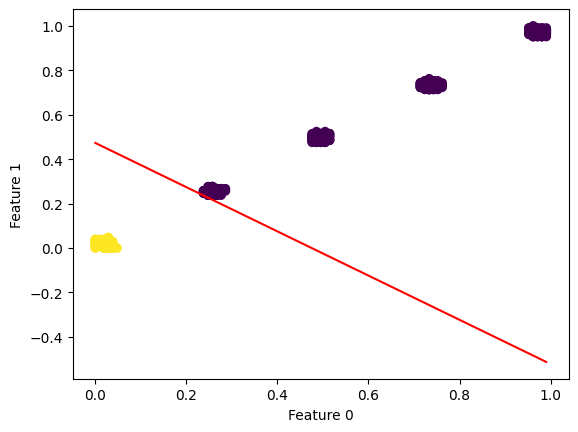


Рисунок 17 – Результат отделения кластера в тестовых данных

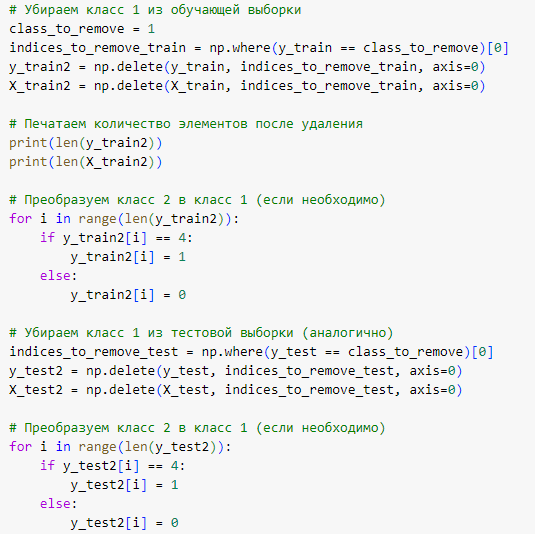


Рисунок 18 – Удаление класса из выборок

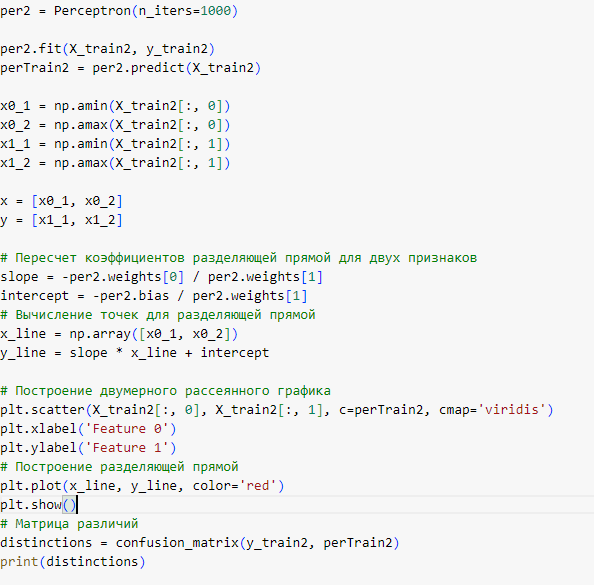


Рисунок 19 – Создание персептрона для 2 кластера

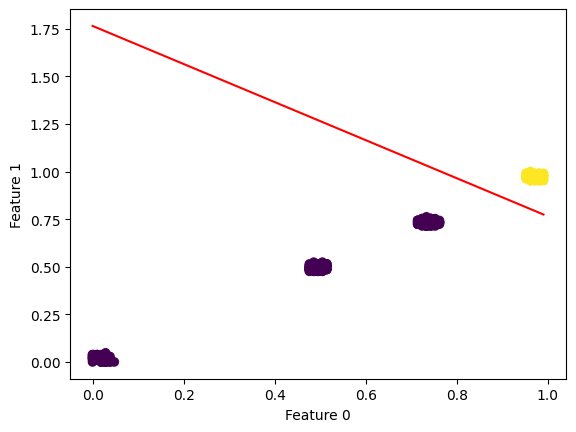


Рисунок 20 – Удаление кластера из выборки

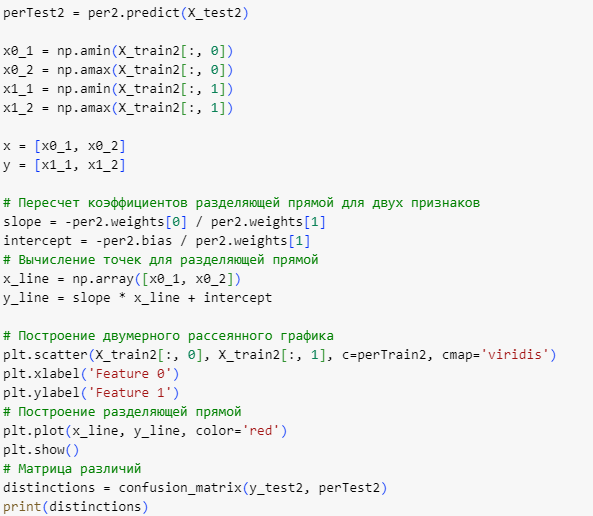


Рисунок 21 –Перерасчёт для тестовой выборки

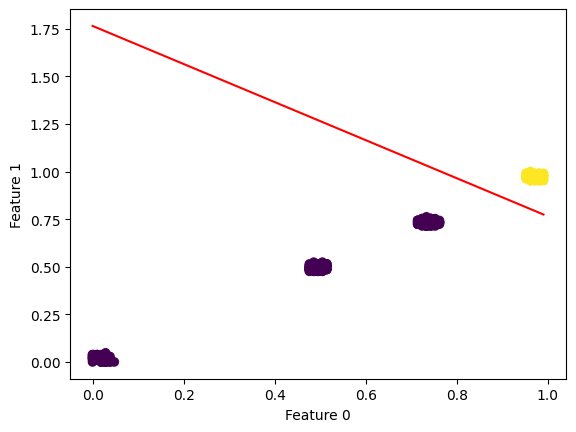


Рисунок 22 – Удаление кластера из тестовой выборки

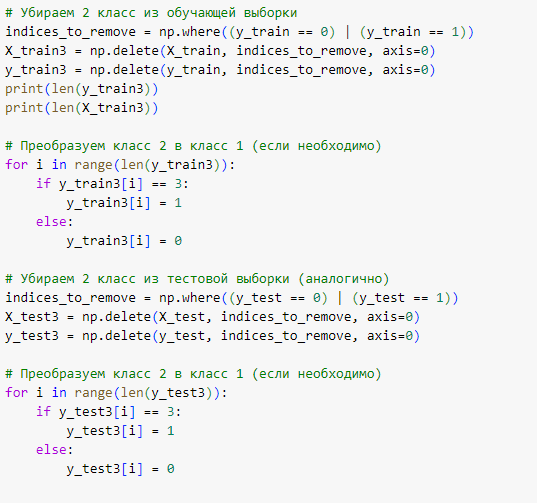


Рисунок 23 –Убираем класс из выборки и преобразуем данные

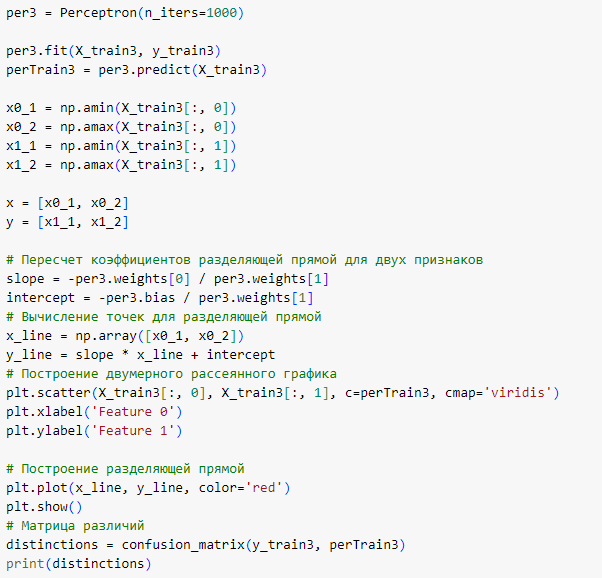


Рисунок 24 – Обучение персептрона для отделения нового кластера

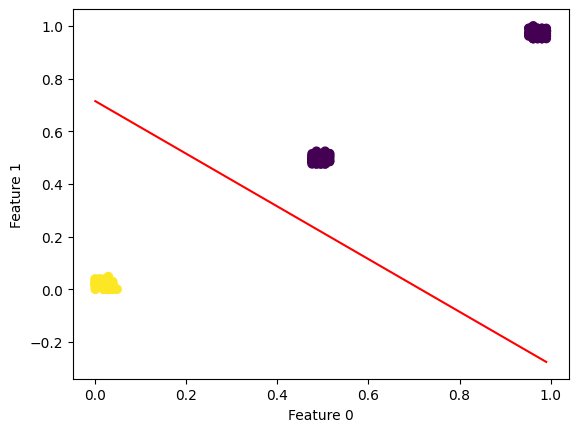


Рисунок 25 – Отделение кластера

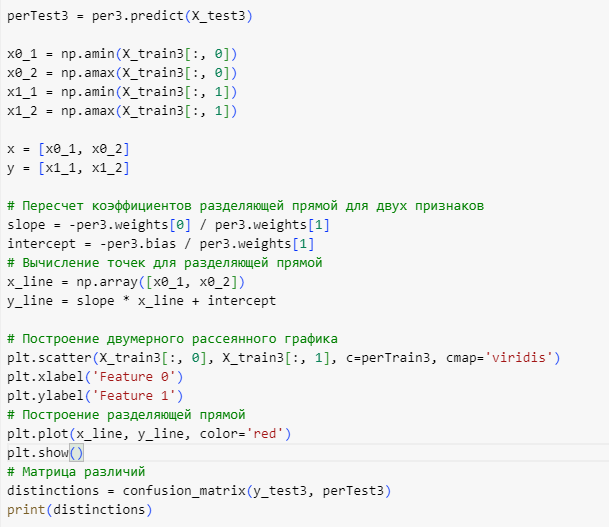


Рисунок 26 – Перерасчёт коэффициентов и построение разделяющей прямой для тестовой выборки

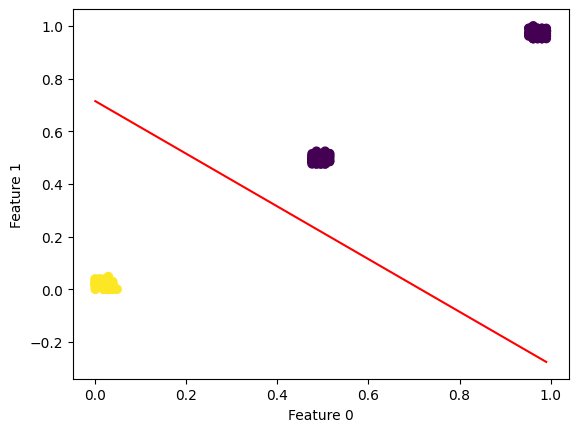


Рисунок 27 – Удаление кластера

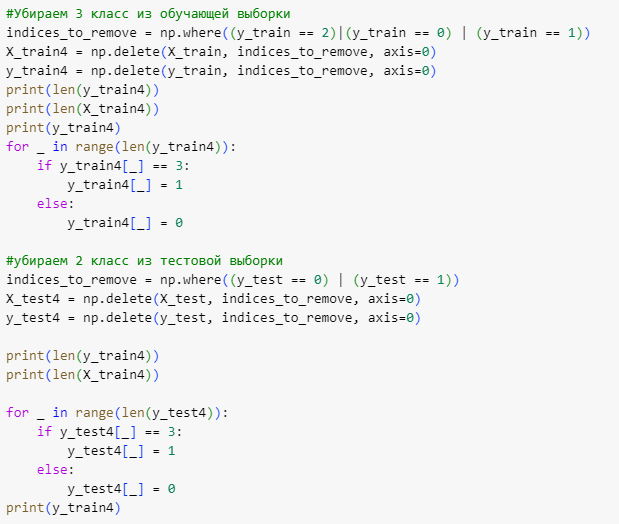


Рисунок 28 – Убираем третий класс из выборки

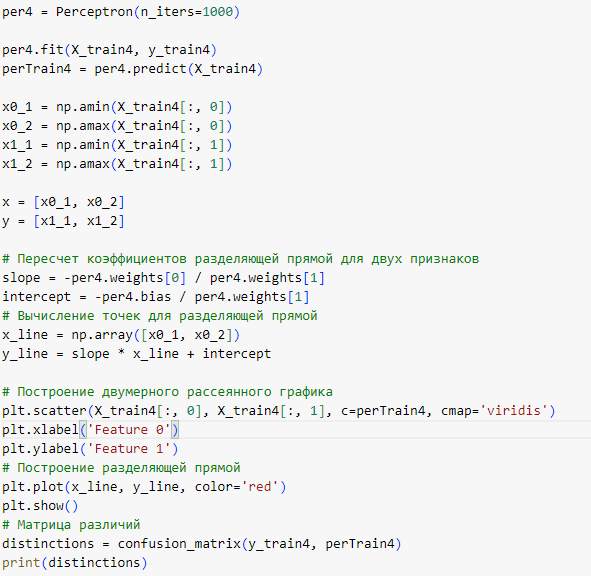


Рисунок 29 – Создание персептрона для отделения последнего класса

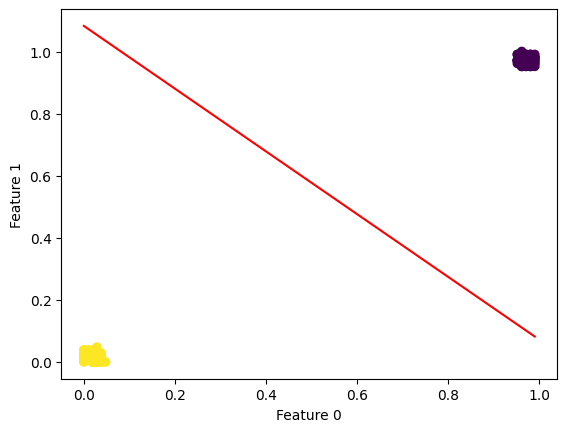


Рисунок 30 – Отделение третьего класса

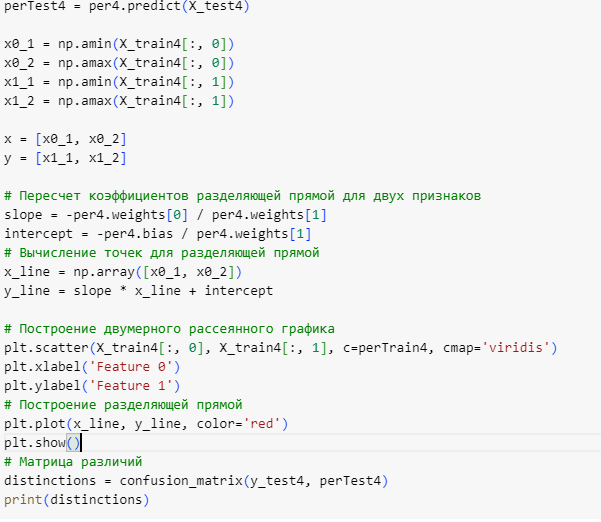


Рисунок 31 – Перерасчёт коэффициентов прямой

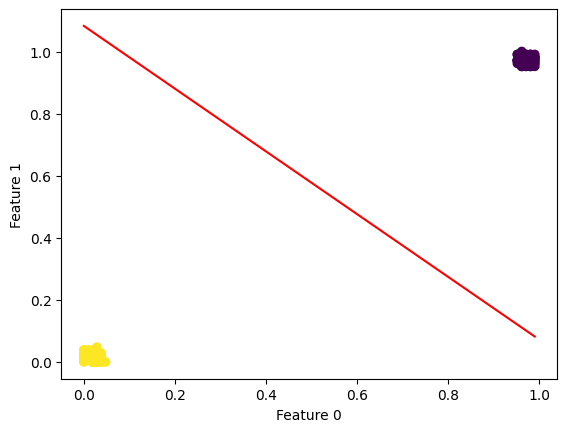


Рисунок 32 – Отделение класса от тестовой выборки

# 4 Создание сети Кохонена и её работа

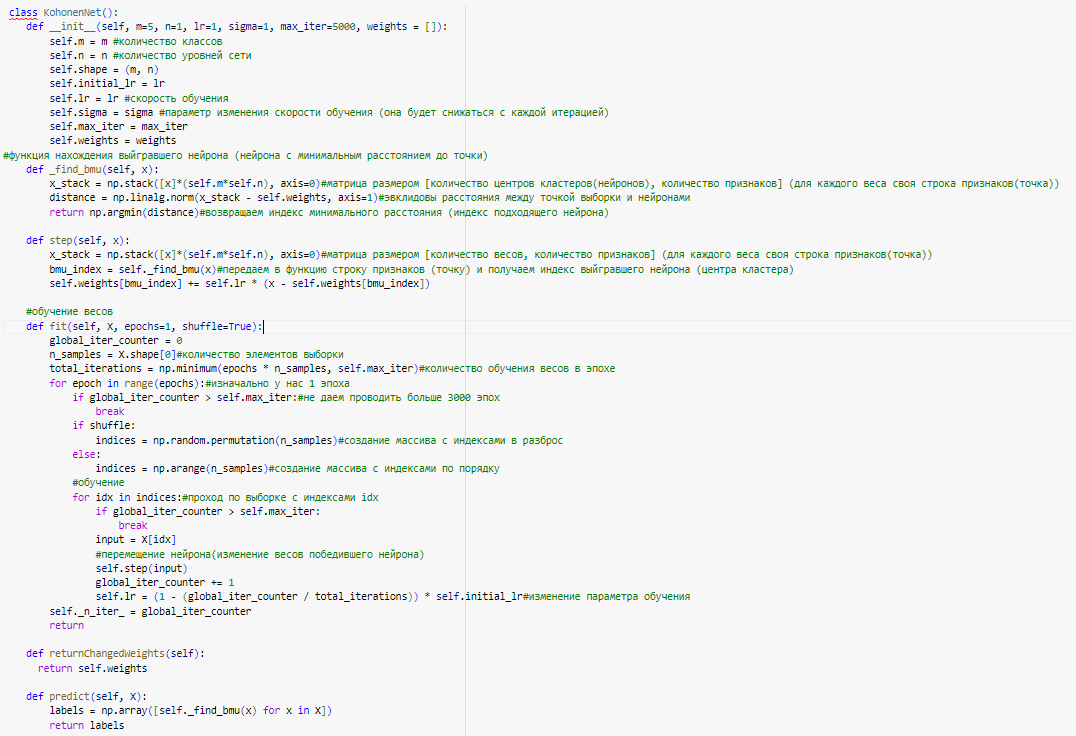


Рисунок 33 – Реализация класса сети Кохонена

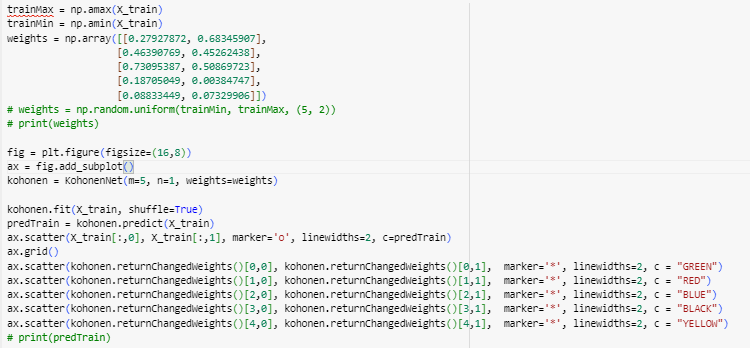


Рисунок 34 – Создание класса Кохонена и весов для работы его алгоритма

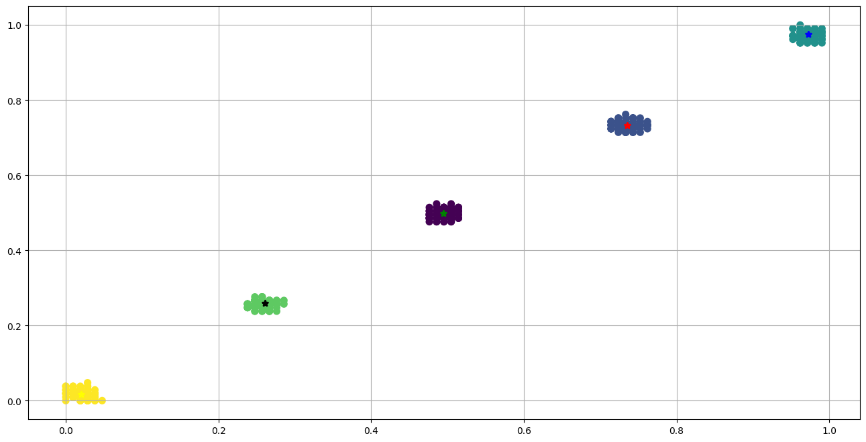


Рисунок 35 – Результат работы сети Кохонена

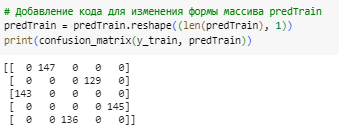


Рисунок 36 – Матрица ошибок

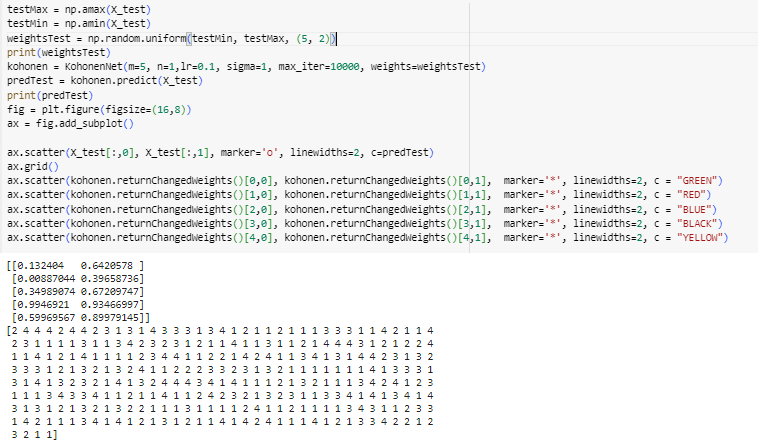


Рисунок 37 – Обучение сети на тестовых данных и вывод весов с метками кластеров

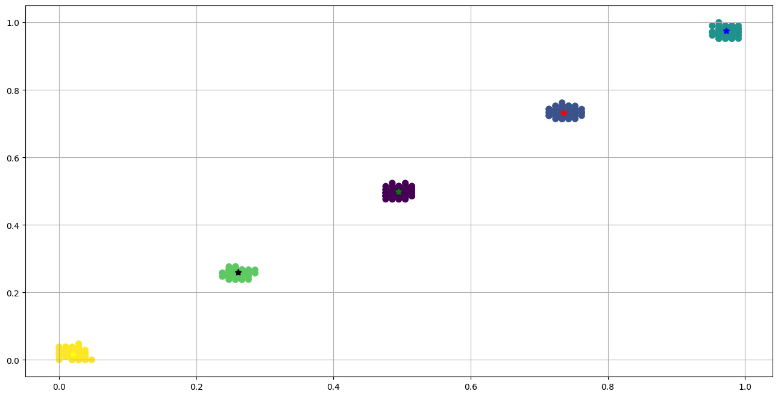


Рисунок 38 – Вывод классов после работы сети Кохонена

# 5 Вероятностная нейронная сеть (PNN)

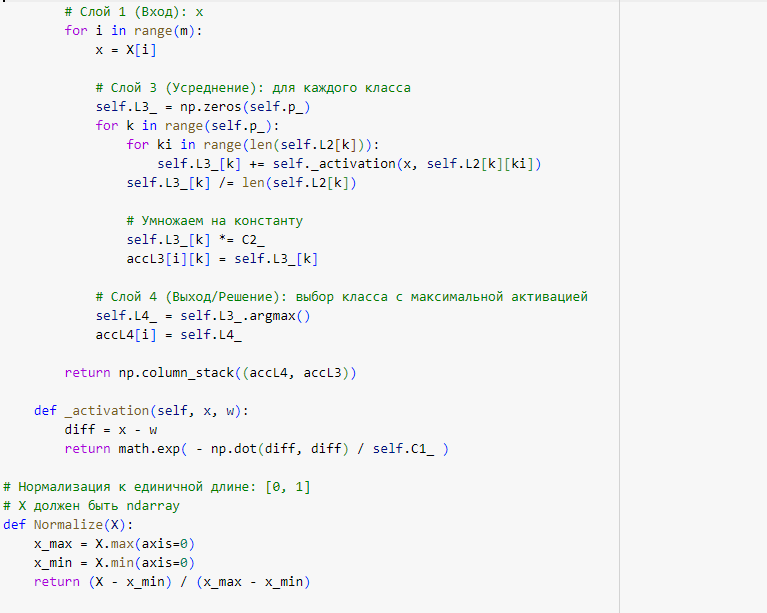
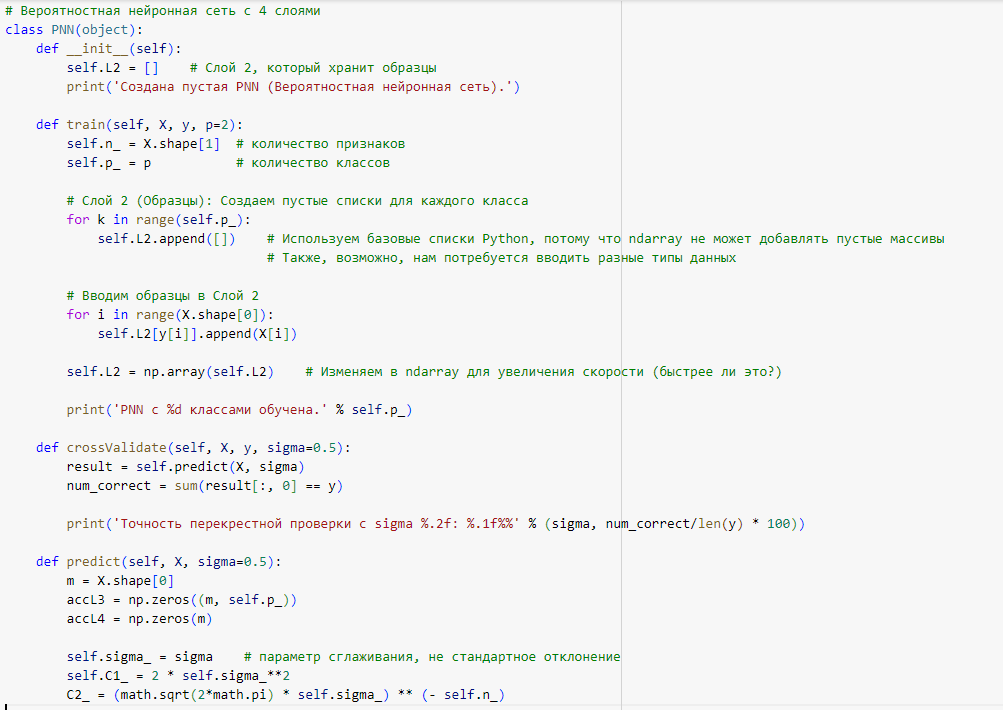


Рисунок 39 – Создание класса PNN сети

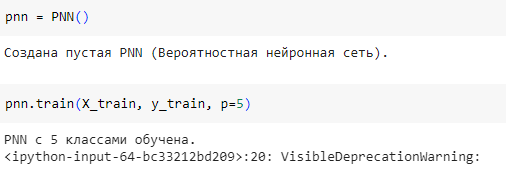


Рисунок 40 – Создание сети и обучение на выборке

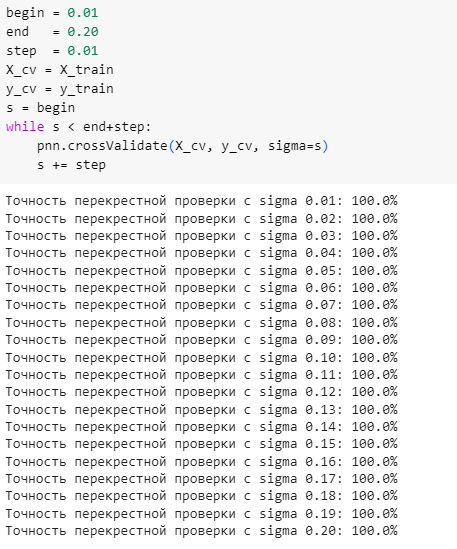


Рисунок 40 – Точность проверки

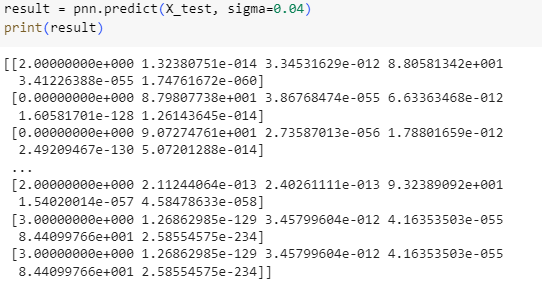
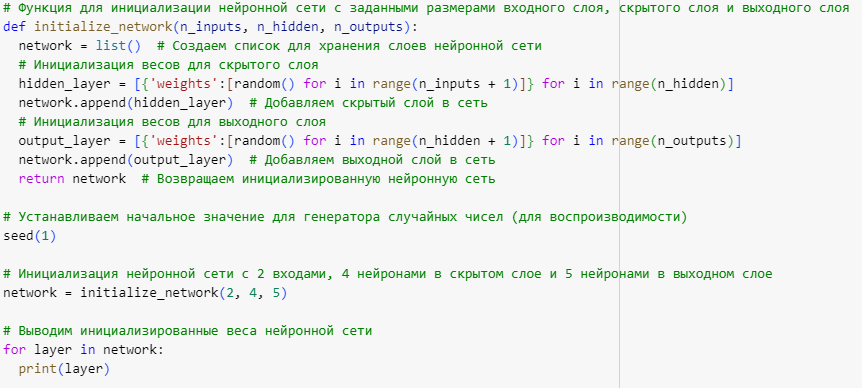


Рисунок 40 – Предсказание данных и их вывод

# 6 Сеть с прямым распрост­ра­нением сигнала и обрат­ным распространением ошибки (MLP)

Рисунок 41 – Создание размеров нейронной сети

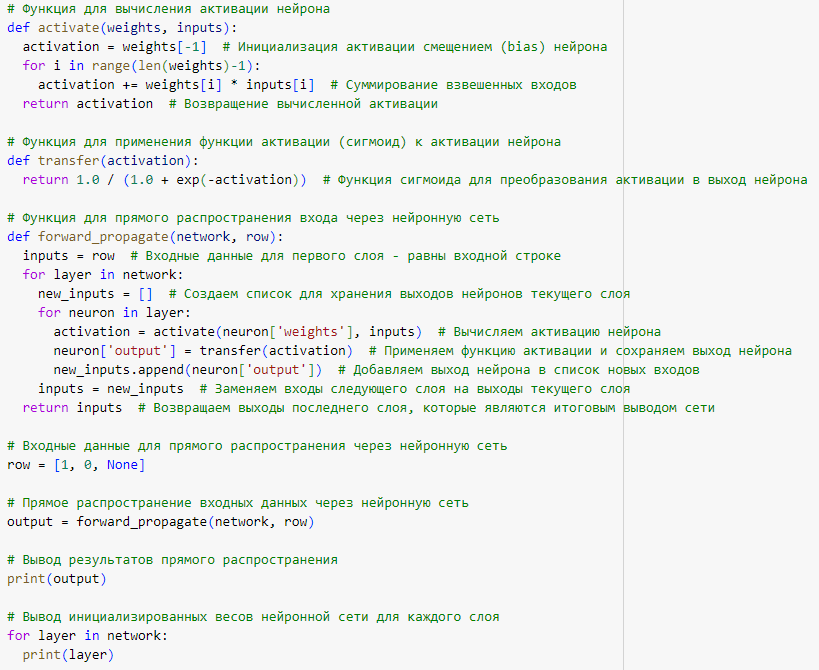
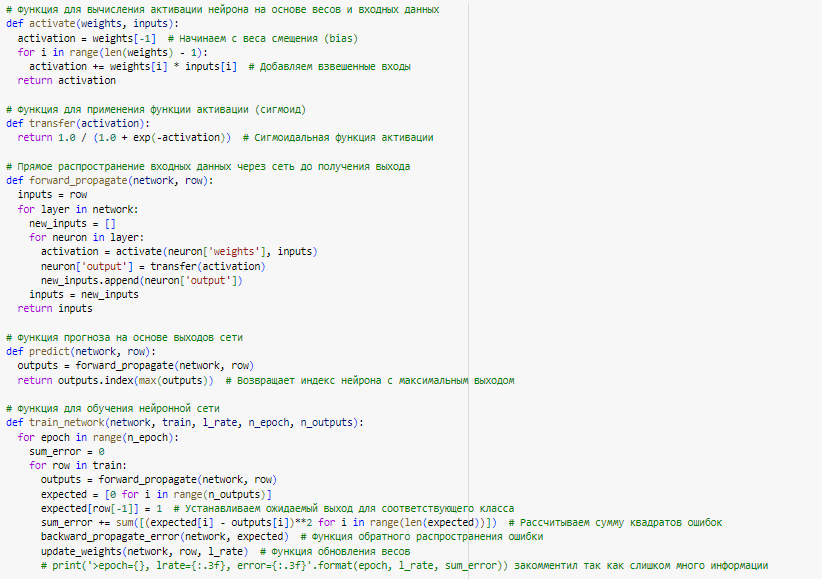
Рисунок 42 – Функции для работы сети



Рисунок 43 – Вторая часть функций сети

Рисунок 44 – Третья часть функций для работы сети

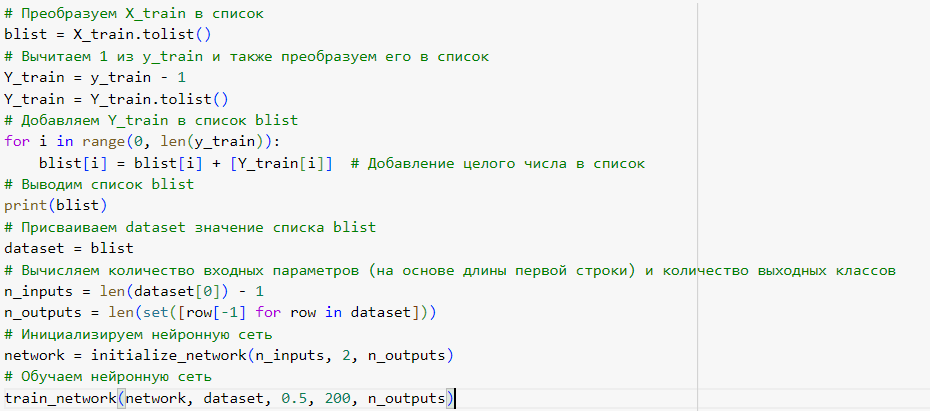


Рисунок 45 – Создание и обучение экземпляра класса сети

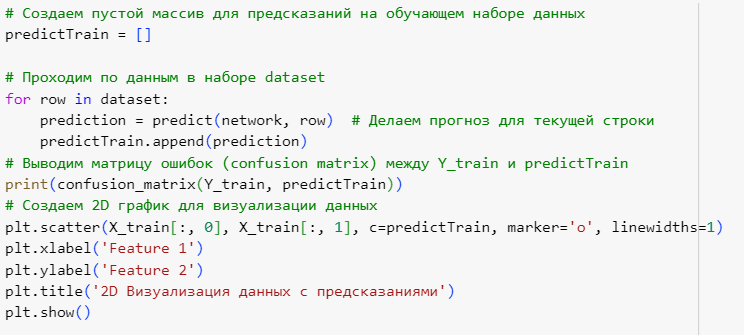


Рисунок 46 – Код вывода матрицы ошибок и визуализация данных

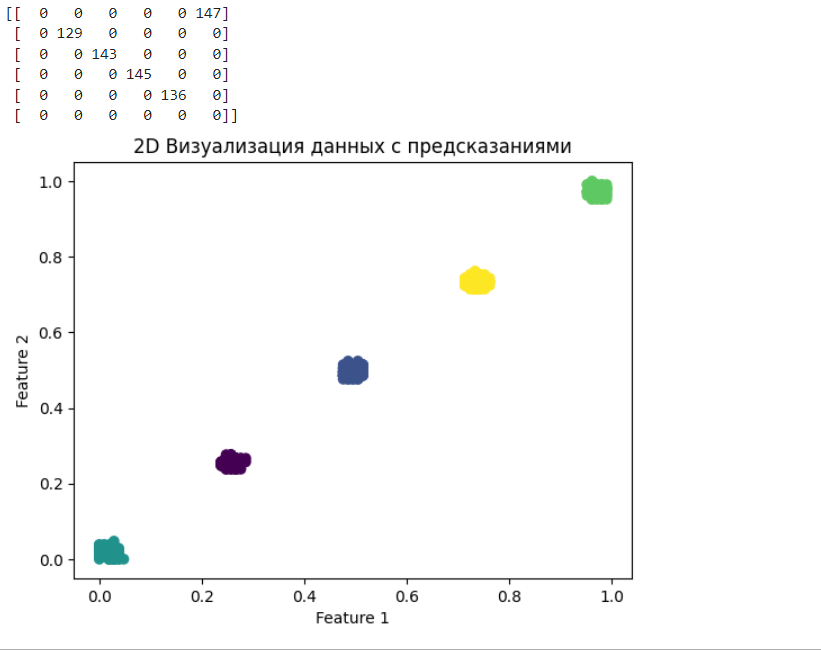


Рисунок 47 – Матрица ошибок и визуализированные данные

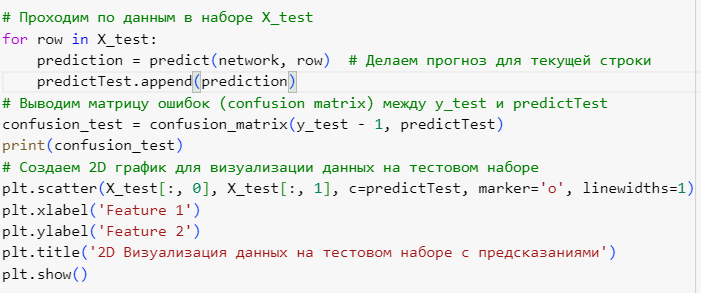


Рисунок 48 – Код вывода матрицы ошибок и визуализация данных для тестовой выборки

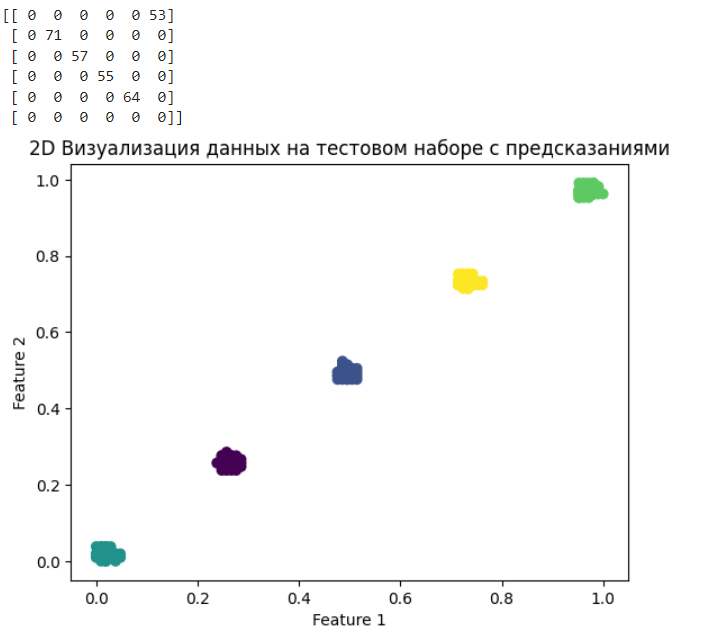


Рисунок 49 – Матрица ошибок и визуализированные данные для тестовой выборки

3 Результаты работы

В ходе выполненной работы было проведено исследование, а также реализация и тестирование разнообразных видов нейронных сетей, включая конкурирующую сеть (сеть Кохонена), персептрон, нейронную сеть для задачи распознавания, а также сеть прямого распространения сигнала и обратного распространения ошибки. Результаты этого исследования подтвердили, что все сети успешно справились с задачей классификации объектов, достигнув невероятной точности в 100%.

# Приложение

import numpy as np

from sklearn.preprocessing import MinMaxScaler

import matplotlib.pyplot as plt

from sklearn.linear\_model import Perceptron

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

import pandas as pd

import plotly.express as px

import plotly.graph\_objects as go

import math

from random import seed

from random import random

from math import exp

from sklearn.datasets import make\_blobs

# Указываем параметры

# Z = 1  Выбираем гипергеометричское распределение

P = 2  # Число признаков

N = 5  # Число классов

count = 0

M = 200  # Число наблюдений

data = []

cluster\_spacing = 25  # Расстояние между кластерами

ngood = np.random.randint(70, 100)

nbad = 100 - ngood

nsample = np.random.randint(1, 100)

for i in range(5):

    c = np.random.hypergeometric(ngood=ngood, nbad=nbad, nsample=nsample, size=(M, P)) + count

    count += cluster\_spacing  # Увеличиваем расстояние между кластерами

    data.append(c)

# Собираем признаки из всех классов

x = np.concatenate([c[:, 0] for c in data])  # Признаки по оси X

y = np.concatenate([c[:, 1] for c in data])  # Признаки по оси Y

xy = [[x[i], y[i]] for i in range(len(x))]

# Создаем список Y, присваивая каждому классу уникальное значение

Y = []

for i in range(5):

    Y.extend([i] \* M)

# Преобразуем список Y в массив numpy

Y = np.array(Y)

# Создаем DataFrame

res\_df = pd.DataFrame(xy)

# Создаем двумерный рассеянный график

# plt.scatter(x, y, c=Y)

# # Показываем график

# plt.show()

# print(Y)

# Создаем DataFrame с добавленными значениями кластера

df = pd.DataFrame({'X': x, 'Y': y, 'Cluster': Y})

# Создаем график без аннотаций

fig = px.scatter(df, x='X', y='Y', color='Cluster', labels={'X': 'Feature 0', 'Y': 'Feature 1', 'Cluster': 'Cluster'})

fig.update\_traces(marker=dict(size=6))

# Отображаем график

fig.show()

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler((0, 1))

scaler.fit(res\_df)

standart\_df = scaler.transform(res\_df)

print(type(standart\_df))

print(standart\_df)

# Импортируем класс KMeans из библиотеки scikit-learn для кластеризации данных.

from sklearn.cluster import KMeans

# Создаем объект KMeans с параметрами:

# - n\_clusters=5 указывает, что мы хотим разделить данные на 5 кластера.

# - random\_state=0 используется для установки начального состояния генератора случайных чисел и обеспечения воспроизводимости результатов.

kmeans = KMeans(n\_clusters=5,n\_init=10, random\_state=0)

# Обучаем модель K-Means на стандартизированных данных `standart\_df`.

kmeans.fit(standart\_df)

# Создаем DataFrame `df\_cluster\_centers` для хранения координат центров кластеров.

df\_cluster\_centers = pd.DataFrame(kmeans.cluster\_centers\_)

# Выводим DataFrame `df\_cluster\_centers`, который содержит координаты центров кластеров.

print(df\_cluster\_centers)

print("----------------------------------------------------------------------------------")

print("матрица ошибок")

# Выводим матрицу ошибок, которая показывает, как точки данных распределены по кластерам

# и сравнивает это с истинными метками классов.

# `Y` содержит истинные метки классов, а `kmeans.labels\_` содержит предсказанные метки кластеров.

confusion\_mat = confusion\_matrix(Y, kmeans.labels\_)

print(confusion\_mat)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(standart\_df, kmeans.labels\_, test\_size=0.3, random\_state=42)

# X\_train - обучающие признаки

# X\_test - тестовые признаки

# y\_train - метки обучающих данных

# y\_test - метки тестовых данных

print("Форма X\_train:", X\_train.shape)

print("Форма X\_test:", X\_test.shape)

assert len(X\_train) + len(X\_test) == len(standart\_df), "Выборки пересекаются или неправильно разделены."

print("----------------------------------------------------------------------------------")

print("Форма y\_train:", y\_train.shape)

print("Форма y\_test:", y\_test.shape)

assert len(y\_train) + len(y\_test) == len(standart\_df), "Выборки пересекаются или неправильно разделены."

print("----------------------------------------------------------------------------------")

# Вывести первые несколько объектов данных (первые 5, например)

print("Первые 5 объектов X\_train:")

print(X\_train[:5])

print("Первые 5 объектов y\_train:")

print(y\_train[:10])

print("Первые 5 объектов X\_test:")

print(X\_test[:5])

print("Первые 5 объектов y\_test:")

print(y\_test[:5])

class Perceptron:

    def \_\_init\_\_(self, learning\_rate=0.01, n\_iters=1000):

        self.lr = learning\_rate

        self.n\_iters = n\_iters

        self.activation\_func = self.\_unit\_step\_func

        self.weights = None

        self.bias = None

    def fit(self, X, y):

        n\_samples, n\_features = X.shape

        self.weights = np.zeros(n\_features) #

        self.bias = 0

        for \_ in range(self.n\_iters):

            for idx, x\_i in enumerate(X):

                linear\_output = np.dot(x\_i, self.weights) + self.bias # тут ошибка

                y\_predicted = self.activation\_func(linear\_output)

                update = self.lr \* (y[idx] - y\_predicted)

                self.weights += update \* x\_i

                self.bias += update

    def predict(self, X):

        linear\_output = np.dot(X, self.weights) + self.bias

        y\_predicted = self.activation\_func(linear\_output)

        return y\_predicted

    #функция активации

    def \_unit\_step\_func(self, x):

        return np.where(x >= 0, 1, 0)

# Создайте экземпляр класса Perceptron

perceptron = Perceptron(learning\_rate=0.05, n\_iters=1000)

# Обучите персептрон на обучающих данных

perceptron.fit(X\_train, y\_train)

# Выполните предсказания на трансформированных данных X\_test

predictions = perceptron.predict(X\_test)

# Определите процент правильно классифицированных объектов

correct\_classifications = np.sum(predictions == y\_test)

total\_objects = len(y\_test)

accuracy = correct\_classifications / total\_objects \* 100

print(f"Процент правильно классифицированных объектов: {accuracy}%")

#отделяем первый класс от других в обучающей выборке

y\_train1 = [[] for \_ in range(len(y\_train))]#создаем массив для отделения из Y\_train 1 класса

for \_ in range(len(y\_train)):

    if y\_train[\_] == 3:

        y\_train1[\_] = 1

    else:

        y\_train1[\_] = 0

Y\_train1 = np.array(y\_train1)

Y\_train1 = Y\_train1.reshape((len(y\_train), 1))

#отделяем первый класс от других  в тестирующей выборке

Y\_test1 = [[] for \_ in range(len(y\_test))]#создаем массив для отделения из Y\_train 1 класса

for \_ in range(len(y\_test)):

    if y\_test[\_] == 3:

        Y\_test1[\_] = 1

    else:

        Y\_test1[\_] = 0

y\_test1 = np.array(Y\_test1)

y\_test1 = y\_test1.reshape((len(y\_test), 1))

dfff = pd.DataFrame(X\_train)

# Создаем DataFrame с обучающими данными

dfff = pd.DataFrame({'Feature 0': X\_train[:, 0], 'Feature 1': X\_train[:, 1], 'Cluster': Y\_train1.flatten()})

# Создаем график с кластерами и метками

fig = go.Figure(data=[

    go.Scatter(

        x=dfff['Feature 0'],

        y=dfff['Feature 1'],

        mode='markers',

        marker=dict(size=6, color=Y\_train1.flatten()),  # Цвет по кластерам

        text=Y\_train1.flatten()  # Метки для отображения при наведении

    )

])

fig.update\_layout(xaxis\_title='Feature 0', yaxis\_title='Feature 1')

fig.show()

per = Perceptron(n\_iters=1000)

per.fit(X\_train, y\_train1)

perTrain1 = per.predict(X\_train)

x0\_1 = np.amin(X\_train[:, 0])

x0\_2 = np.amax(X\_train[:, 0])

x1\_1 = np.amin(X\_train[:, 1])

x1\_2 = np.amax(X\_train[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per.weights[0] / per.weights[1]

intercept = -per.bias / per.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=perTrain1, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_train1, perTrain1)

print(distinctions)

perTest = per.predict(X\_test)

x0\_1 = np.amin(X\_train[:, 0])

x0\_2 = np.amax(X\_train[:, 0])

x1\_1 = np.amin(X\_train[:, 1])

x1\_2 = np.amax(X\_train[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per.weights[0] / per.weights[1]

intercept = -per.bias / per.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=perTrain1, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_test1, perTest)

print(distinctions)

# Убираем класс 1 из обучающей выборки

class\_to\_remove = 1

indices\_to\_remove\_train = np.where(y\_train == class\_to\_remove)[0]

y\_train2 = np.delete(y\_train, indices\_to\_remove\_train, axis=0)

X\_train2 = np.delete(X\_train, indices\_to\_remove\_train, axis=0)

# Печатаем количество элементов после удаления

print(len(y\_train2))

print(len(X\_train2))

# Преобразуем класс 2 в класс 1 (если необходимо)

for i in range(len(y\_train2)):

    if y\_train2[i] == 4:

        y\_train2[i] = 1

    else:

        y\_train2[i] = 0

# Убираем класс 1 из тестовой выборки (аналогично)

indices\_to\_remove\_test = np.where(y\_test == class\_to\_remove)[0]

y\_test2 = np.delete(y\_test, indices\_to\_remove\_test, axis=0)

X\_test2 = np.delete(X\_test, indices\_to\_remove\_test, axis=0)

# Преобразуем класс 2 в класс 1 (если необходимо)

for i in range(len(y\_test2)):

    if y\_test2[i] == 4:

        y\_test2[i] = 1

    else:

        y\_test2[i] = 0

per2 = Perceptron(n\_iters=1000)

per2.fit(X\_train2, y\_train2)

perTrain2 = per2.predict(X\_train2)

x0\_1 = np.amin(X\_train2[:, 0])

x0\_2 = np.amax(X\_train2[:, 0])

x1\_1 = np.amin(X\_train2[:, 1])

x1\_2 = np.amax(X\_train2[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per2.weights[0] / per2.weights[1]

intercept = -per2.bias / per2.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train2[:, 0], X\_train2[:, 1], c=perTrain2, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_train2, perTrain2)

print(distinctions)

perTest2 = per2.predict(X\_test2)

x0\_1 = np.amin(X\_train2[:, 0])

x0\_2 = np.amax(X\_train2[:, 0])

x1\_1 = np.amin(X\_train2[:, 1])

x1\_2 = np.amax(X\_train2[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per2.weights[0] / per2.weights[1]

intercept = -per2.bias / per2.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train2[:, 0], X\_train2[:, 1], c=perTrain2, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_test2, perTest2)

print(distinctions)

# Убираем 2 класс из обучающей выборки

indices\_to\_remove = np.where((y\_train == 0) | (y\_train == 1))

X\_train3 = np.delete(X\_train, indices\_to\_remove, axis=0)

y\_train3 = np.delete(y\_train, indices\_to\_remove, axis=0)

print(len(y\_train3))

print(len(X\_train3))

# Преобразуем класс 2 в класс 1 (если необходимо)

for i in range(len(y\_train3)):

    if y\_train3[i] == 3:

        y\_train3[i] = 1

    else:

        y\_train3[i] = 0

# Убираем 2 класс из тестовой выборки (аналогично)

indices\_to\_remove = np.where((y\_test == 0) | (y\_test == 1))

X\_test3 = np.delete(X\_test, indices\_to\_remove, axis=0)

y\_test3 = np.delete(y\_test, indices\_to\_remove, axis=0)

# Преобразуем класс 2 в класс 1 (если необходимо)

for i in range(len(y\_test3)):

    if y\_test3[i] == 3:

        y\_test3[i] = 1

    else:

        y\_test3[i] = 0

per3 = Perceptron(n\_iters=1000)

per3.fit(X\_train3, y\_train3)

perTrain3 = per3.predict(X\_train3)

x0\_1 = np.amin(X\_train3[:, 0])

x0\_2 = np.amax(X\_train3[:, 0])

x1\_1 = np.amin(X\_train3[:, 1])

x1\_2 = np.amax(X\_train3[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per3.weights[0] / per3.weights[1]

intercept = -per3.bias / per3.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train3[:, 0], X\_train3[:, 1], c=perTrain3, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_train3, perTrain3)

print(distinctions)

perTest3 = per3.predict(X\_test3)

x0\_1 = np.amin(X\_train3[:, 0])

x0\_2 = np.amax(X\_train3[:, 0])

x1\_1 = np.amin(X\_train3[:, 1])

x1\_2 = np.amax(X\_train3[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per3.weights[0] / per3.weights[1]

intercept = -per3.bias / per3.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train3[:, 0], X\_train3[:, 1], c=perTrain3, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_test3, perTest3)

print(distinctions)

#Убираем 3 класс из обучающей выборки

indices\_to\_remove = np.where((y\_train == 2)|(y\_train == 0) | (y\_train == 1))

X\_train4 = np.delete(X\_train, indices\_to\_remove, axis=0)

y\_train4 = np.delete(y\_train, indices\_to\_remove, axis=0)

print(len(y\_train4))

print(len(X\_train4))

print(y\_train4)

for \_ in range(len(y\_train4)):

    if y\_train4[\_] == 3:

        y\_train4[\_] = 1

    else:

        y\_train4[\_] = 0

#убираем 2 класс из тестовой выборки

indices\_to\_remove = np.where((y\_test == 0) | (y\_test == 1))

X\_test4 = np.delete(X\_test, indices\_to\_remove, axis=0)

y\_test4 = np.delete(y\_test, indices\_to\_remove, axis=0)

print(len(y\_train4))

print(len(X\_train4))

for \_ in range(len(y\_test4)):

    if y\_test4[\_] == 3:

        y\_test4[\_] = 1

    else:

        y\_test4[\_] = 0

print(y\_train4)

per4 = Perceptron(n\_iters=1000)

per4.fit(X\_train4, y\_train4)

perTrain4 = per4.predict(X\_train4)

x0\_1 = np.amin(X\_train4[:, 0])

x0\_2 = np.amax(X\_train4[:, 0])

x1\_1 = np.amin(X\_train4[:, 1])

x1\_2 = np.amax(X\_train4[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per4.weights[0] / per4.weights[1]

intercept = -per4.bias / per4.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train4[:, 0], X\_train4[:, 1], c=perTrain4, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_train4, perTrain4)

print(distinctions)

perTest4 = per4.predict(X\_test4)

x0\_1 = np.amin(X\_train4[:, 0])

x0\_2 = np.amax(X\_train4[:, 0])

x1\_1 = np.amin(X\_train4[:, 1])

x1\_2 = np.amax(X\_train4[:, 1])

x = [x0\_1, x0\_2]

y = [x1\_1, x1\_2]

# Пересчет коэффициентов разделяющей прямой для двух признаков

slope = -per4.weights[0] / per4.weights[1]

intercept = -per4.bias / per4.weights[1]

# Вычисление точек для разделяющей прямой

x\_line = np.array([x0\_1, x0\_2])

y\_line = slope \* x\_line + intercept

# Построение двумерного рассеянного графика

plt.scatter(X\_train4[:, 0], X\_train4[:, 1], c=perTrain4, cmap='viridis')

plt.xlabel('Feature 0')

plt.ylabel('Feature 1')

# Построение разделяющей прямой

plt.plot(x\_line, y\_line, color='red')

plt.show()

# Матрица различий

distinctions = confusion\_matrix(y\_test4, perTest4)

print(distinctions)

 class KohonenNet():

    def \_\_init\_\_(self, m=5, n=1, lr=1, sigma=1, max\_iter=5000, weights = []):

        self.m = m #количество классов

        self.n = n #количество уровней сети

        self.shape = (m, n)

        self.initial\_lr = lr

        self.lr = lr #скорость обучения

        self.sigma = sigma #параметр изменения скорости обучения (она будет снижаться с каждой итерацией)

        self.max\_iter = max\_iter

        self.weights = weights

#функция нахождения выйгравшего нейрона (нейрона с минимальным расстоянием до точки)

    def \_find\_bmu(self, x):

        x\_stack = np.stack([x]\*(self.m\*self.n), axis=0)#матрица размером [количество центров кластеров(нейронов), количество признаков] (для каждого веса своя строка признаков(точка))

        distance = np.linalg.norm(x\_stack - self.weights, axis=1)#эвклидовы расстояния между точкой выборки и нейронами

        return np.argmin(distance)#возвращаем индекс минимального расстояния (индекс подходящего нейрона)

    def step(self, x):

        x\_stack = np.stack([x]\*(self.m\*self.n), axis=0)#матрица размером [количество весов, количество признаков] (для каждого веса своя строка признаков(точка))

        bmu\_index = self.\_find\_bmu(x)#передаем в функцию строку признаков (точку) и получаем индекс выйгравшего нейрона (центра кластера)

        self.weights[bmu\_index] += self.lr \* (x - self.weights[bmu\_index])

    #обучение весов

    def fit(self, X, epochs=1, shuffle=True):

        global\_iter\_counter = 0

        n\_samples = X.shape[0]#количество элементов выборки

        total\_iterations = np.minimum(epochs \* n\_samples, self.max\_iter)#количество обучения весов в эпохе

        for epoch in range(epochs):#изначально у нас 1 эпоха

            if global\_iter\_counter > self.max\_iter:#не даем проводить больше 3000 эпох

                break

            if shuffle:

                indices = np.random.permutation(n\_samples)#создание массива с индексами в разброс

            else:

                indices = np.arange(n\_samples)#создание массива с индексами по порядку

            #обучение

            for idx in indices:#проход по выборке с индексами idx

                if global\_iter\_counter > self.max\_iter:

                    break

                input = X[idx]

                #перемещение нейрона(изменение весов победившего нейрона)

                self.step(input)

                global\_iter\_counter += 1

                self.lr = (1 - (global\_iter\_counter / total\_iterations)) \* self.initial\_lr#изменение параметра обучения

        self.\_n\_iter\_ = global\_iter\_counter

        return

    def returnChangedWeights(self):

      return self.weights

    def predict(self, X):

        labels = np.array([self.\_find\_bmu(x) for x in X])

        return labels

trainMax = np.amax(X\_train)

trainMin = np.amin(X\_train)

weights = np.array([[0.27927872, 0.68345907],

                   [0.46390769, 0.45262438],

                   [0.73095387, 0.50869723],

                   [0.18705049, 0.00384747],

                   [0.08833449, 0.07329906]])

# weights = np.random.uniform(trainMin, trainMax, (5, 2))

# print(weights)

fig = plt.figure(figsize=(16,8))

ax = fig.add\_subplot()

kohonen = KohonenNet(m=5, n=1, weights=weights)

kohonen.fit(X\_train, shuffle=True)

predTrain = kohonen.predict(X\_train)

ax.scatter(X\_train[:,0], X\_train[:,1], marker='o', linewidths=2, c=predTrain)

ax.grid()

ax.scatter(kohonen.returnChangedWeights()[0,0], kohonen.returnChangedWeights()[0,1],  marker='\*', linewidths=2, c = "GREEN")

ax.scatter(kohonen.returnChangedWeights()[1,0], kohonen.returnChangedWeights()[1,1],  marker='\*', linewidths=2, c = "RED")

ax.scatter(kohonen.returnChangedWeights()[2,0], kohonen.returnChangedWeights()[2,1],  marker='\*', linewidths=2, c = "BLUE")

ax.scatter(kohonen.returnChangedWeights()[3,0], kohonen.returnChangedWeights()[3,1],  marker='\*', linewidths=2, c = "BLACK")

ax.scatter(kohonen.returnChangedWeights()[4,0], kohonen.returnChangedWeights()[4,1],  marker='\*', linewidths=2, c = "YELLOW")

# print(predTrain)

# Добавление кода для изменения формы массива predTrain

predTrain = predTrain.reshape((len(predTrain), 1))

print(confusion\_matrix(y\_train, predTrain))

testMax = np.amax(X\_test)

testMin = np.amin(X\_test)

weightsTest = np.random.uniform(testMin, testMax, (5, 2))

print(weightsTest)

kohonen = KohonenNet(m=5, n=1,lr=0.1, sigma=1, max\_iter=10000, weights=weightsTest)

predTest = kohonen.predict(X\_test)

print(predTest)

fig = plt.figure(figsize=(16,8))

ax = fig.add\_subplot()

ax.scatter(X\_test[:,0], X\_test[:,1], marker='o', linewidths=2, c=predTest)

ax.grid()

ax.scatter(kohonen.returnChangedWeights()[0,0], kohonen.returnChangedWeights()[0,1],  marker='\*', linewidths=2, c = "GREEN")

ax.scatter(kohonen.returnChangedWeights()[1,0], kohonen.returnChangedWeights()[1,1],  marker='\*', linewidths=2, c = "RED")

ax.scatter(kohonen.returnChangedWeights()[2,0], kohonen.returnChangedWeights()[2,1],  marker='\*', linewidths=2, c = "BLUE")

ax.scatter(kohonen.returnChangedWeights()[3,0], kohonen.returnChangedWeights()[3,1],  marker='\*', linewidths=2, c = "BLACK")

ax.scatter(kohonen.returnChangedWeights()[4,0], kohonen.returnChangedWeights()[4,1],  marker='\*', linewidths=2, c = "YELLOW")

predTest = predTest.reshape((len(predTest),1))

print(confusion\_matrix(y\_test, predTest))

# Вероятностная нейронная сеть с 4 слоями

class PNN(object):

    def \_\_init\_\_(self):

        self.L2 = []    # Слой 2, который хранит образцы

        print('Создана пустая PNN (Вероятностная нейронная сеть).')

    def train(self, X, y, p=2):

        self.n\_ = X.shape[1]  # количество признаков

        self.p\_ = p           # количество классов

        # Слой 2 (Образцы): Создаем пустые списки для каждого класса

        for k in range(self.p\_):

            self.L2.append([])    # Используем базовые списки Python, потому что ndarray не может добавлять пустые массивы

                                  # Также, возможно, нам потребуется вводить разные типы данных

        # Вводим образцы в Слой 2

        for i in range(X.shape[0]):

            self.L2[y[i]].append(X[i])

        self.L2 = np.array(self.L2)    # Изменяем в ndarray для увеличения скорости (быстрее ли это?)

        print('PNN с %d классами обучена.' % self.p\_)

    def crossValidate(self, X, y, sigma=0.5):

        result = self.predict(X, sigma)

        num\_correct = sum(result[:, 0] == y)

        print('Точность перекрестной проверки с sigma %.2f: %.1f%%' % (sigma, num\_correct/len(y) \* 100))

    def predict(self, X, sigma=0.5):

        m = X.shape[0]

        accL3 = np.zeros((m, self.p\_))

        accL4 = np.zeros(m)

        self.sigma\_ = sigma    # параметр сглаживания, не стандартное отклонение

        self.C1\_ = 2 \* self.sigma\_\*\*2

        C2\_ = (math.sqrt(2\*math.pi) \* self.sigma\_) \*\* (- self.n\_)

        # Слой 1 (Вход): x

        for i in range(m):

            x = X[i]

            # Слой 3 (Усреднение): для каждого класса

            self.L3\_ = np.zeros(self.p\_)

            for k in range(self.p\_):

                for ki in range(len(self.L2[k])):

                    self.L3\_[k] += self.\_activation(x, self.L2[k][ki])

                self.L3\_[k] /= len(self.L2[k])

                # Умножаем на константу

                self.L3\_[k] \*= C2\_

                accL3[i][k] = self.L3\_[k]

            # Слой 4 (Выход/Решение): выбор класса с максимальной активацией

            self.L4\_ = self.L3\_.argmax()

            accL4[i] = self.L4\_

        return np.column\_stack((accL4, accL3))

    def \_activation(self, x, w):

        diff = x - w

        return math.exp( - np.dot(diff, diff) / self.C1\_ )

# Нормализация к единичной длине: [0, 1]

# X должен быть ndarray

def Normalize(X):

    x\_max = X.max(axis=0)

    x\_min = X.min(axis=0)

    return (X - x\_min) / (x\_max - x\_min)

pnn = PNN()

pnn.train(X\_train, y\_train, p=5)

begin = 0.01

end   = 0.20

step  = 0.01

X\_cv = X\_train

y\_cv = y\_train

s = begin

while s < end+step:

    pnn.crossValidate(X\_cv, y\_cv, sigma=s)

    s += step

result = pnn.predict(X\_test, sigma=0.04)

print(result)

# Функция для инициализации нейронной сети с заданными размерами входного слоя, скрытого слоя и выходного слоя

def initialize\_network(n\_inputs, n\_hidden, n\_outputs):

  network = list()  # Создаем список для хранения слоев нейронной сети

  # Инициализация весов для скрытого слоя

  hidden\_layer = [{'weights':[random() for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]

  network.append(hidden\_layer)  # Добавляем скрытый слой в сеть

  # Инициализация весов для выходного слоя

  output\_layer = [{'weights':[random() for i in range(n\_hidden + 1)]} for i in range(n\_outputs)]

  network.append(output\_layer)  # Добавляем выходной слой в сеть

  return network  # Возвращаем инициализированную нейронную сеть

# Устанавливаем начальное значение для генератора случайных чисел (для воспроизводимости)

seed(1)

# Инициализация нейронной сети с 2 входами, 4 нейронами в скрытом слое и 5 нейронами в выходном слое

network = initialize\_network(2, 4, 5)

# Выводим инициализированные веса нейронной сети

for layer in network:

  print(layer)

# Функция для вычисления активации нейрона

def activate(weights, inputs):

  activation = weights[-1]  # Инициализация активации смещением (bias) нейрона

  for i in range(len(weights)-1):

    activation += weights[i] \* inputs[i]  # Суммирование взвешенных входов

  return activation  # Возвращение вычисленной активации

# Функция для применения функции активации (сигмоид) к активации нейрона

def transfer(activation):

  return 1.0 / (1.0 + exp(-activation))  # Функция сигмоида для преобразования активации в выход нейрона

# Функция для прямого распространения входа через нейронную сеть

def forward\_propagate(network, row):

  inputs = row  # Входные данные для первого слоя - равны входной строке

  for layer in network:

    new\_inputs = []  # Создаем список для хранения выходов нейронов текущего слоя

    for neuron in layer:

      activation = activate(neuron['weights'], inputs)  # Вычисляем активацию нейрона

      neuron['output'] = transfer(activation)  # Применяем функцию активации и сохраняем выход нейрона

      new\_inputs.append(neuron['output'])  # Добавляем выход нейрона в список новых входов

    inputs = new\_inputs  # Заменяем входы следующего слоя на выходы текущего слоя

  return inputs  # Возвращаем выходы последнего слоя, которые являются итоговым выводом сети

# Входные данные для прямого распространения через нейронную сеть

row = [1, 0, None]

# Прямое распространение входных данных через нейронную сеть

output = forward\_propagate(network, row)

# Вывод результатов прямого распространения

print(output)

# Вывод инициализированных весов нейронной сети для каждого слоя

for layer in network:

  print(layer)

# Функция для вычисления производной функции активации (сигмоид)

def transfer\_derivative(output):

  return output \* (1.0 - output)  # Производная сигмоида

# Функция для обратного распространения ошибки через нейронную сеть

def backward\_propagate\_error(network, expected):

  for i in reversed(range(len(network))):

    layer = network[i]

    errors = list()  # Создаем список для хранения ошибок нейронов текущего слоя

    if i != len(network) - 1:

      for j in range(len(layer)):

        error = 0.0

        for neuron in network[i + 1]:

          error += (neuron['weights'][j] \* neuron['delta'])  # Суммируем ошибки следующего слоя, взвешенные по соответствующим весам

        errors.append(error)

    else:

      for j in range(len(layer)):

        neuron = layer[j]

        errors.append(expected[j] - neuron['output'])  # Рассчитываем ошибку для выходного слоя

    for j in range(len(layer)):

      neuron = layer[j]

      neuron['delta'] = errors[j] \* transfer\_derivative(neuron['output'])  # Вычисляем дельту (ошибку) для нейрона

# Ожидаемые значения для обучения

expected = [0, 1, 2, 3, 4]

# Вызываем функцию обратного распространения ошибки

backward\_propagate\_error(network, expected)

# Выводим обновленные значения для весов нейронной сети

for layer in network:

  print(layer)

# Функция для обновления весов нейронной сети на основе вычисленных ошибок

def update\_weights(network, row, l\_rate):

  for i in range(len(network)):

    inputs = row[:-1]  # Входные данные для текущего слоя

    if i != 0:

      inputs = [neuron['output'] for neuron in network[i - 1]]  # Если не первый слой, берем выходы предыдущего слоя

    for neuron in network[i]:

      for j in range(len(inputs)):

        # Обновление весов нейрона с учетом ошибки и скорости обучения

        neuron['weights'][j] += l\_rate \* neuron['delta'] \* inputs[j]

      neuron['weights'][-1] += l\_rate \* neuron['delta']  # Обновление веса смещения (bias) нейрона

# Функция для вычисления активации нейрона на основе весов и входных данных

def activate(weights, inputs):

  activation = weights[-1]  # Начинаем с веса смещения (bias)

  for i in range(len(weights) - 1):

    activation += weights[i] \* inputs[i]  # Добавляем взвешенные входы

  return activation

# Функция для применения функции активации (сигмоид)

def transfer(activation):

  return 1.0 / (1.0 + exp(-activation))  # Сигмоидальная функция активации

# Прямое распространение входных данных через сеть до получения выхода

def forward\_propagate(network, row):

  inputs = row

  for layer in network:

    new\_inputs = []

    for neuron in layer:

      activation = activate(neuron['weights'], inputs)

      neuron['output'] = transfer(activation)

      new\_inputs.append(neuron['output'])

    inputs = new\_inputs

  return inputs

# Функция прогноза на основе выходов сети

def predict(network, row):

  outputs = forward\_propagate(network, row)

  return outputs.index(max(outputs))  # Возвращает индекс нейрона с максимальным выходом

# Функция для обучения нейронной сети

def train\_network(network, train, l\_rate, n\_epoch, n\_outputs):

  for epoch in range(n\_epoch):

    sum\_error = 0

    for row in train:

      outputs = forward\_propagate(network, row)

      expected = [0 for i in range(n\_outputs)]

      expected[row[-1]] = 1  # Устанавливаем ожидаемый выход для соответствующего класса

      sum\_error += sum([(expected[i] - outputs[i])\*\*2 for i in range(len(expected))])  # Рассчитываем сумму квадратов ошибок

      backward\_propagate\_error(network, expected)  # Функция обратного распространения ошибки

      update\_weights(network, row, l\_rate)  # Функция обновления весов

      # print('>epoch={}, lrate={:.3f}, error={:.3f}'.format(epoch, l\_rate, sum\_error)) закомментил так как слишком много информации

# Преобразуем X\_train в список

blist = X\_train.tolist()

# Вычитаем 1 из y\_train и также преобразуем его в список

Y\_train = y\_train - 1

Y\_train = Y\_train.tolist()

# Добавляем Y\_train в список blist

for i in range(0, len(y\_train)):

    blist[i] = blist[i] + [Y\_train[i]]  # Добавление целого числа в список

# Выводим список blist

print(blist)

# Присваиваем dataset значение списка blist

dataset = blist

# Вычисляем количество входных параметров (на основе длины первой строки) и количество выходных классов

n\_inputs = len(dataset[0]) - 1

n\_outputs = len(set([row[-1] for row in dataset]))

# Инициализируем нейронную сеть

network = initialize\_network(n\_inputs, 2, n\_outputs)

# Обучаем нейронную сеть

train\_network(network, dataset, 0.5, 200, n\_outputs)

# Создаем пустой массив для предсказаний на обучающем наборе данных

predictTrain = []

# Проходим по данным в наборе dataset

for row in dataset:

    prediction = predict(network, row)  # Делаем прогноз для текущей строки

    predictTrain.append(prediction)

# Выводим матрицу ошибок (confusion matrix) между Y\_train и predictTrain

print(confusion\_matrix(Y\_train, predictTrain))

# Создаем 2D график для визуализации данных

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=predictTrain, marker='o', linewidths=1)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('2D Визуализация данных с предсказаниями')

plt.show()

# Создаем пустой массив для предсказаний на тестовом наборе данных

predictTest = []

# Проходим по данным в наборе X\_test

for row in X\_test:

    prediction = predict(network, row)  # Делаем прогноз для текущей строки

    predictTest.append(prediction)

# Выводим матрицу ошибок (confusion matrix) между y\_test и predictTest

confusion\_test = confusion\_matrix(y\_test - 1, predictTest)

print(confusion\_test)

# Создаем 2D график для визуализации данных на тестовом наборе

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=predictTest, marker='o', linewidths=1)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('2D Визуализация данных на тестовом наборе с предсказаниями')

plt.show()