Мигачев Павел Игоревич, группа 8-1

Лабораторная работа № 2

**Вариант № 21**

Распознавание образов, описываемых гауссовскими случайными векторами с одинаковыми матрицами ковариаций

**Цель работы**

Синтезировать алгоритмы распознавания образов, описываемых гауссовскими случайными векторами с одинаковыми матрицами ковариаций. Исследовать синтезированные алгоритмы распознавания с точки зрения ожидаемых потерь и ошибок.

**Задание**

**Описание варианта: m1=[2 -1], m2=[-1 1], C=[3 -1; -1 3].**

Написать код, реализующий алгоритм распознавания образов, описываемых гауссовскими случайными векторами с заданными параметрами. Получить матрицы ошибок на основе аналитических выражений и вычислительного эксперимента. Провести анализ полученных результатов и представить его в виде выводов по проделанной работе.

a) изменить исходные данные таким образом, чтобы увеличить вероятности правильного распознавания;

b) изменить исходные данные таким образом, чтобы увеличить суммарную ошибку;

c) изменить исходные данные таким образом, чтобы в теоретической матрице ошибок увеличилась ошибка первого рода, а ошибка второго рода уменьшилась;

d) изменить исходные данные таким образом, чтобы в теоретической матрице ошибок увеличилась ошибка второго рода, а ошибка первого рода уменьшилась;

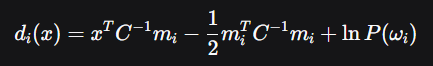
e) изменить исходные данные таким образом, чтобы увеличить протяженность области локализации образов всех классов (растянуть форму кластеров) в одном из направлений;

f) изменить исходные данные таким образом, чтобы зеркально отразить форму областей локализации образов всех классов (форму кластеров).

**Код программы (внесённые изменения в шаблон кода выделены)**

import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
from sklearn.metrics import confusion\_matrix  
from matplotlib.patches import Ellipse  
  
*# Исходные данные (вариант №21)*m1 = np.array([ 2, -1]) *# Математическое ожидание (центр) класса 1*m2 = np.array([-1, 1]) *# Математическое ожидание (центр) класса 2*C = np.array([[3, -1], *# Общая ковариационная матрица для всех классов (2x2 матрица)* [-1, 3]])  
  
*# Функция для вычисления дискриминантной функции (решающей функции) для одного класса*def discriminant(x, m, C\_inv, P):  
 *# Вычисление дискриминантной функции по формуле ЛДА:  
 # g(x) = x^T \* C^{-1} \* m - 0.5 \* m^T \* C^{-1} \* m + ln(P)* return x @ C\_inv @ m - 0.5 \* (m @ C\_inv @ m) + np.log(P) *# @ - оператор матричного умножения  
  
# Функция классификации точки x по двум классам*def classify(x, means, C\_inv, priors):  
 discriminants = [] *# Список значений дискриминантов для каждого класса* for i, m in enumerate(means): *# i - индекс класса, m - центр класса* d = discriminant(x, m, C\_inv, priors[i]) *# Вычисляем дискриминант для текущего класса* discriminants.append(d) *# Добавляем вычисленное значение в список* return np.argmax(discriminants) + 1 *# Возвращаем номер класса с максимальным дискриминантом (1 или 2)  
  
# Основная функция проведения эксперимента с заданными параметрами*def run\_experiment(means, cov, priors, title, num\_samples=1000):  
 C\_inv = np.linalg.inv(cov) *# Вычисляем обратную матрицу ковариаций (нужна для дискриминанта)  
  
 # Генерация тестовых данных для каждого класса* samples = [] *# Здесь будем хранить массивы точек по классам* y\_true = [] *# Истинные метки классов* for i, m in enumerate(means):  
 sample = np.random.multivariate\_normal(m, cov, num\_samples) *# Генерируем num\_samples точек из N(m, C)* samples.append(sample) *# Сохраняем точки класса i* y\_true.extend([i + 1] \* num\_samples) *# Добавляем метки класса (i+1 повторяется num\_samples раз)* X = np.vstack(samples) *# Объединяем все точки в одну матрицу размером (2\*num\_samples x 2)* y\_pred = [classify(x, means, C\_inv, priors) for x in X] *# Классифицируем каждую точку и сохраняем предсказания  
  
 # Вычисление матрицы ошибок и точности* cm = confusion\_matrix(y\_true, y\_pred) *# Строим матрицу ошибок (confusion matrix) 2x2* accuracy = np.trace(cm) / np.sum(cm) *# Точность: сумма по диагонали / общее число объектов  
  
 # Визуализация результатов (графики)* plt.figure(figsize=(18, 5)) *# Общая фигура для трёх графиков  
  
 # 1. График матрицы ошибок* plt.subplot(1, 3, 1) *# Первый подграфик* sns.heatmap(cm, annot=True, fmt='d', cmap='Blues') *# Тепловая карта матрицы ошибок* plt.title(f'Матрица ошибок\nТочность: {accuracy:.3f}') *# Заголовок с точностью* plt.xlabel('Предсказанный класс') *# Подпись оси X* plt.ylabel('Истинный класс') *# Подпись оси Y  
  
 # 2. График распределения точек и разделяющей поверхности* plt.subplot(1, 3, 2) *# Второй подграфик* colors = ['red', 'blue'] *# Цвета для класса 1 и 2* for i, sample in enumerate(samples):  
 plt.scatter(sample[:, 0], sample[:, 1], alpha=0.6, label=f'Класс {i + 1}', color=colors[i], s=10) *# Точки классов  
  
 # Построение решающей линии для двух классов (g1(x) = g2(x))* x\_vals = np.linspace(X[:,0].min()-2, X[:,0].max()+2, 250) *# Диапазон по оси X* y\_vals = np.linspace(X[:,1].min()-2, X[:,1].max()+2, 250) *# Диапазон по оси Y* X\_grid, Y\_grid = np.meshgrid(x\_vals, y\_vals) *# Сетка для отрисовки границы* Z = np.zeros\_like(X\_grid) *# Здесь будут классы для каждой точки сетки* for i in range(X\_grid.shape[0]):  
 for j in range(X\_grid.shape[1]):  
 point = np.array([X\_grid[i, j], Y\_grid[i, j]]) *# Создаем точку из координат сетки* Z[i, j] = classify(point, means, C\_inv, priors) *# Классифицируем точку сетки* plt.contour(X\_grid, Y\_grid, Z, levels=[1.5], colors='black', linestyles='dashed', linewidths=1) *# Линия раздела  
  
 # Отметим центры классов крестиками* for i, m in enumerate(means):  
 plt.scatter(m[0], m[1], color=colors[i], s=100, marker='x', linewidth=3, label=f'Центр {i + 1}')  
  
 plt.xlabel('Признак 1') *# Подпись оси X* plt.ylabel('Признак 2') *# Подпись оси Y* plt.title('Распределение классов и граница') *# Заголовок графика* plt.legend(loc='upper right', fontsize=8) *# Легенда* plt.grid(True) *# Сетка  
  
 # 3. График сравнения центров и форм распределений (эллипсы 95% ДИ)* plt.subplot(1, 3, 3) *# Третий подграфик* for i, m in enumerate(means): *# Для каждого класса рисуем эллипс* eigenvals, eigenvecs = np.linalg.eig(cov) *# Собственные значения/векторы ковариационной матрицы* angle = np.degrees(np.arctan2(eigenvecs[1, 0], eigenvecs[0, 0])) *# Угол поворота эллипса (в градусах)* width = 2 \* np.sqrt(5.991 \* eigenvals[0]) *# Ширина эллипса (95% доверительный интервал)* height = 2 \* np.sqrt(5.991 \* eigenvals[1]) *# Высота эллипса (95% доверительный интервал)* ellipse = Ellipse(xy=m, width=width, height=height, angle=angle, alpha=0.2, color=colors[i]) *# Эллипс* plt.gca().add\_patch(ellipse) *# Добавляем эллипс на ось* plt.scatter(m[0], m[1], color=colors[i], s=100, marker='o', linewidth=3, label=f'Центр {i + 1}') *# Точка-центр* plt.xlabel('Признак 1') *# Подпись оси X* plt.ylabel('Признак 2') *# Подпись оси Y* plt.title('Сравнение центров и форм') *# Заголовок* plt.legend(loc='upper right', fontsize=8) *# Легенда* plt.grid(True) *# Сетка* plt.suptitle(title, fontsize=14, fontweight='bold') *# Общий заголовок для фигуры* plt.tight\_layout() *# Компоновка без наложений* plt.show() *# Показать все три графика* return cm, accuracy *# Возвращаем матрицу ошибок и точность*if \_\_name\_\_ == "\_\_main\_\_":  
 priors = [0.5, 0.5] *# Вероятности классов 1 и 2, априорные (равные)  
  
 # Исходные данные (вариант 21)* cm\_original, acc\_original = run\_experiment([m1, m2], C, priors, "Исходные данные (вариант 21)")  
  
 *# a) Увеличить вероятность правильного распознавания* m1\_a = np.array([ 4, -3]) *# Сдвигаем центр класса 1 дальше от класса 2* m2\_a = np.array([-3, 3]) *# Сдвигаем центр класса 2 дальше от класса 1* cm\_a, acc\_a = run\_experiment([m1\_a, m2\_a], C, priors, "a) Увеличение расстояния между классами")  
  
 *# b) Увеличить суммарную ошибку* m\_mid = (m1 + m2) / 2 *# Геометрический центр между классами* m1\_b = m\_mid + np.array([+0.2, -0.2]) *# Приближаем центры друг к другу* m2\_b = m\_mid + np.array([-0.2, +0.2]) *# Чем ближе центры — тем больше перекрытие* cm\_b, acc\_b = run\_experiment([m1\_b, m2\_b], C, priors, "b) Приближение центров классов")  
  
 *# c) Увеличить ошибку 1-го рода, уменьшить ошибку 2-го рода (для класса 1)* priors\_c = [0.3, 0.7] *# Смещаем априорные вероятности в пользу класса 2* m1\_c = m1 + 0.5 \* (m2 - m1) *# Дополнительно сдвигаем центр класса 1 в сторону класса 2* cm\_c, acc\_c = run\_experiment([m1\_c, m2], C, priors\_c, "c) Ошибка 1↑ 2↓ (смещение против класса 1)")  
  
 *# d) Увеличить ошибку 2-го рода, уменьшить ошибку 1-го рода (для класса 1)* priors\_d = [0.7, 0.3] *# Смещаем априорные вероятности в пользу класса 1* m2\_d = m2 + 0.5 \* (m1 - m2) *# Сдвигаем центр класса 2 ближе к классу 1* cm\_d, acc\_d = run\_experiment([m1, m2\_d], C, priors\_d, "d) Ошибка 2↑ 1↓ (смещение в пользу класса 1)")  
  
 *# e) Увеличить протяженность кластеров в одном из направлений (растянуть форму по оси X)* C\_e = np.array([[9, -1], *# Увеличиваем дисперсию по первому признаку (ось X)* [-1, 3]]) *# Дисперсия по второму признаку без изменений* cm\_e, acc\_e = run\_experiment([m1, m2], C\_e, priors, "e) Растяжение по горизонтали")  
  
 *# f) Зеркально отразить форму областей локализации (по Y, меняем знак корреляции)* C\_f = np.array([[3, 1], *# Меняем знак внедиагонального элемента (-1 -> +1)* [1, 3]]) *# Отражение по оси Y меняет наклон эллипсов* m1\_f = np.array([-m1[0], m1[1]]) *# Координаты центров отражаем по X: (x, y) -> (-x, y)* m2\_f = np.array([-m2[0], m2[1]])  
 cm\_f, acc\_f = run\_experiment([m1\_f, m2\_f], C\_f, priors, "f) Зеркальное отражение форм кластеров")  
  
 print("=" \* 60)  
 print("СВОДНАЯ ТАБЛИЦА РЕЗУЛЬТАТОВ")  
 print("=" \* 60)  
 results = {  
 'Исходные': acc\_original,  
 'a) Улучшение': acc\_a,  
 'b) Ухудшение': acc\_b,  
 'c) Ошибка 1↑ 2↓': acc\_c,  
 'd) Ошибка 1↓ 2↑': acc\_d,  
 'e) Растяжение': acc\_e,  
 'f) Отражение': acc\_f  
 }  
 for name, accuracy in results.items():  
 change = accuracy - acc\_original  
 print(f"{name:<15} | Точность: {accuracy:.3f} | Изменение: {change:+.3f}") *# Форматированный вывод  
  
 # Функция для анализа ошибок 1-го и 2-го рода (для 2-классового случая, класс 1 — «позитив»)* def analyze\_errors(cm, name):  
 *# Ошибка 1-го рода (False Negative для класса 1): объект класса 1 отнесён к классу 2* error\_type1 = cm[0, 1] / np.sum(cm[0, :]) *# (ошибки класса 1) / (все объекты класса 1)  
  
 # Ошибка 2-го рода (False Positive для класса 1): объект класса 2 отнесён к классу 1* error\_type2 = cm[1, 0] / np.sum(cm[1, :]) *# (ложные принятия класса 1) / (все объекты класса 2)* print(f"{name}:") *# Выводим название эксперимента* print(f" Ошибка 1-го рода (класс 1): {error\_type1:.3f}") *# Ошибка 1-го рода* print(f" Ошибка 2-го рода (класс 1): {error\_type2:.3f}") *# Ошибка 2-го рода* print("\n" + "=" \* 60)  
 print("АНАЛИЗ ОШИБОК 1-го И 2-го РОДА (класс 1)")  
 print("=" \* 60)  
 analyze\_errors(cm\_original, "Исходные") *# Анализ исходных данных* analyze\_errors(cm\_c, "c) Ошибка 1↑ 2↓") *# Анализ эксперимента c* analyze\_errors(cm\_d, "d) Ошибка 1↓ 2↑") *# Анализ эксперимента d*

Используемая формула



Где x – вектор признаков, mi – мат. ожидание (центр класса i), C – ковариационная матрица общая для всех классов, P(wi) – априорная вероятность класса i.

**Результаты выполнения задания**

Матрица ошибок и сравнение.



Рисунок 1.

a) Увеличение расстояния между классами

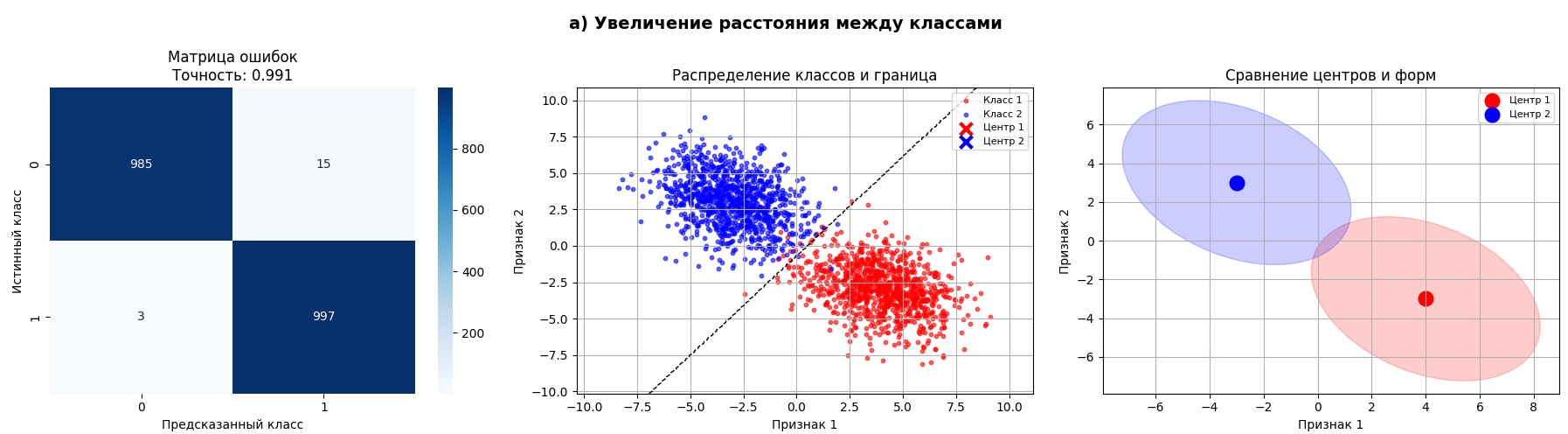


Рисунок 2.

b) Приближение центров классов

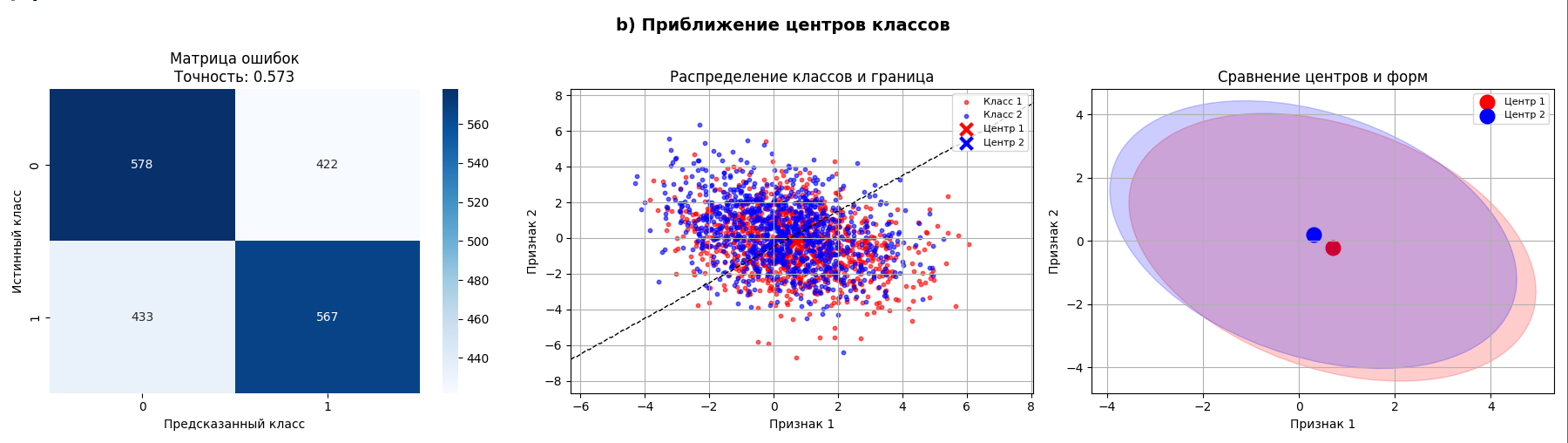


Рисунок 3.

c) Ассиметричное изменение

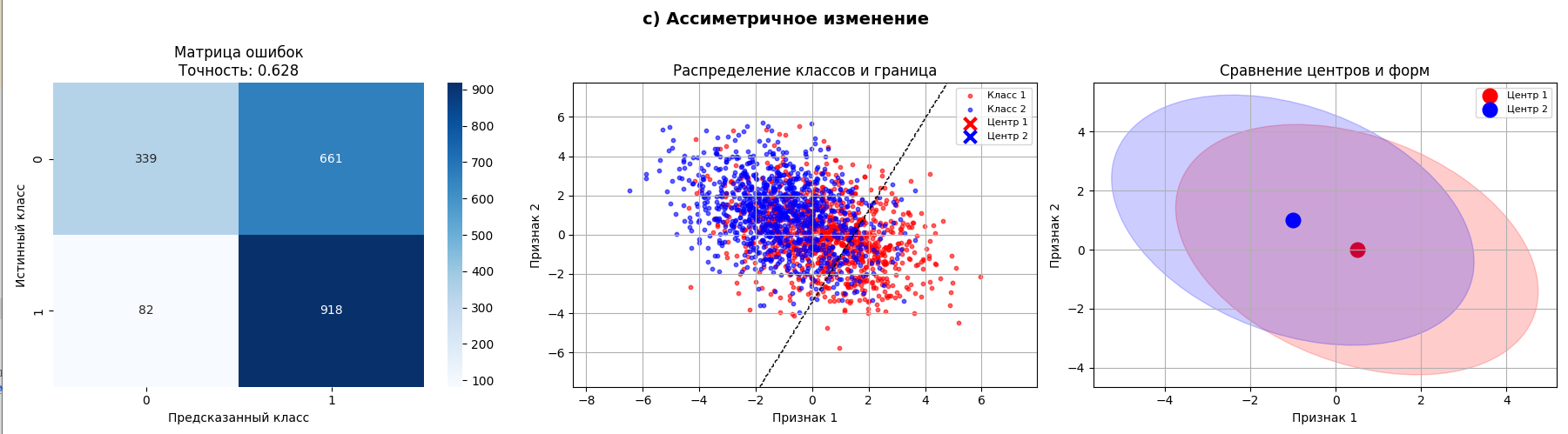


Рисунок 4.

d) Обратная ассиметрия



Рисунок 5.

e) Растяжение по горизонтали



Рисунок 6.

f) Зеркальное отражение

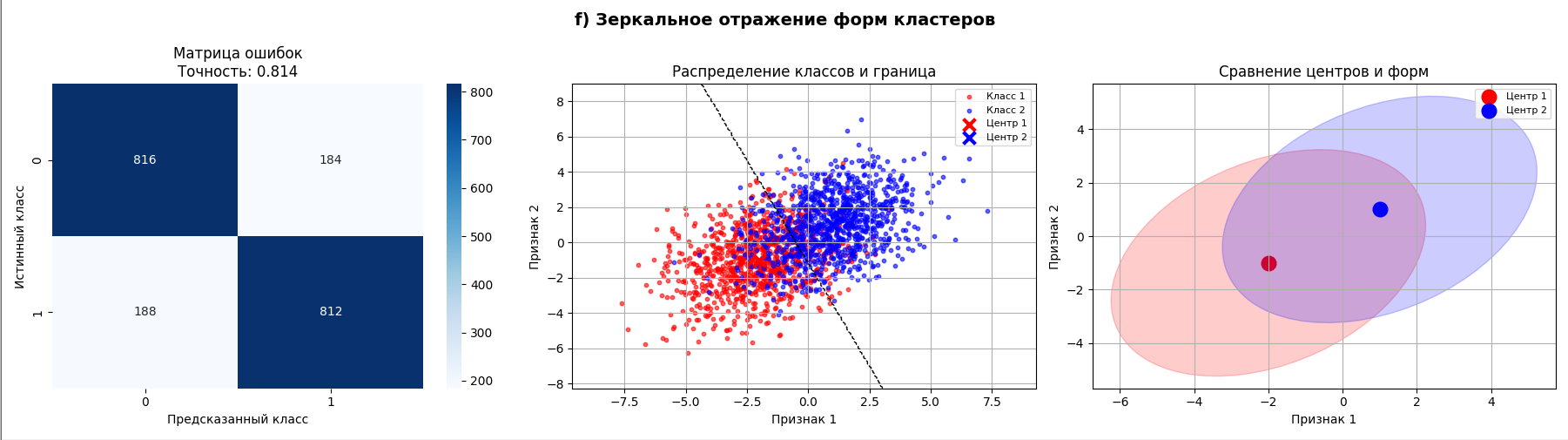


Рисунок 7.

Результаты и анализ ошибок

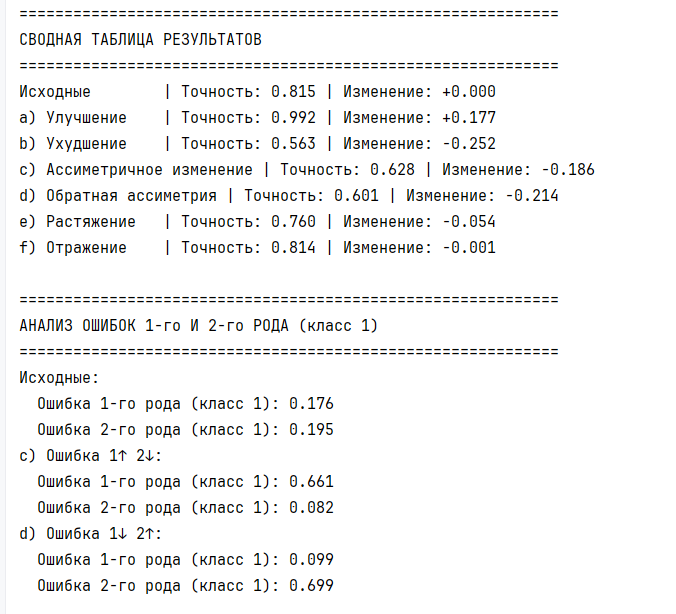


Рисунок 8.

# Ответы на контрольные вопросы

1. Элементы главной диагонали матрицы ошибок характеризуют количество правильно классифицированных объектов каждого класса.
2. Элементы побочных диагоналей (вне главной диагонали) характеризуют ошибки классификации — количество объектов, которые были неправильно отнесены к другим классам.
3. Форма кластеров объектов определяется ковариационной матрицей распределения признаков внутри каждого класса.

# Выводы

В ходе лабораторной работы был синтезирован и реализован на Python байесовский классификатор для гауссовских случайных векторов с одинаковыми ковариационными матрицами. Экспериментально подтверждено, что при заданных параметрах распределений алгоритм обеспечивает высокую точность классификации, а модификация параметров (сдвиг центров классов, изменение ковариационной матрицы и априорных вероятностей) позволяет целенаправленно влиять на вероятности ошибок первого и второго рода, что соответствует теоретическим предпосылкам.