МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Основы машинного обучения»

Тема: Изучение и предобработка данных

Вариант 2

Студентка гр. 1304 Чернякова А.Д.

Преподаватель Жангиров Т.Р.

Санкт-Петербург
2024

Цель работы.

Изучить библиотеки *Pandas, Seaborn, Numpy, Matplotlib* и *Scikit-learn* для предобработки и анализа данных. Протестировать множество методов этих библиотек на нескольких наборах данных.

Задание.

При выполнении лабораторной работы также будет оцениваться количество инструкций, которые были использованы для получения результата.

1. Изучение набора данных iris.csv с использованием Pandas и Seaborn:
   1.1. Загрузить данные из файла как Pandas DataFrame.
   1.2. Вызвав у датафрейма метод head, проверить корректность загруженных данных.
   1.3. Вызвав у датафрейма метод describe, получить характеристики. Опишите полученный результат.
   1.4. Видоизмените полученный датафрейм таким образом, чтобы метка классов были следующими: 0 - Iris-setosa, 1 - Iris-versicolor, 2 - Iris-virginica. Сохраните полученный датафрейм в отдельный файл формата csv.
   1.5. Визуально оцените набор данных, построив изображение, содержащее графики ядерной оценки плотности каждого признака (кроме признака класса), диаграмму рассеяния и двумерную ядерную оценку плотности для каждых признаков. Наблюдения разных классов должны быть выделены отдельным цветом (рекомендуемая палитра 'tab10' или 'Set1'). Пример построения: https://seaborn.pydata.org/examples/pair_grid_with_kde.html. Опишите полученный график, что на нем изображено, какие выводы о данных можно сделать.

1.6. На одном изображении постройте гистограммы распределения для каждого признака (для построения нескольких диаграмм на одном изображении, необходимо создать subplot из matplotlib, и для каждой диаграммы задать параметр ax, указав нужную ячейку. subplot возвращает два параметра: саму фигуру с изображением и список ячеек. Например, изображение с 4 ячейками записанных в ряд: fig, axs = plt.subplots(1,4). Указание ячейки в параметре диаграммы делается следующим образом: ax=axs[0]). Затем последовательно модифицируйте изображение:

1.6.1. Постройте гистограммы для разного количества столбцов: 5,10,15,20,30. Выберите на ваш взгляд такое количество столбцов, который лучшие образом описывает форму распределения признаков.

1.6.2. Сделайте на каждой гистограмме разделение по цвету согласно классу. Проведите это в двух режимах, когда гистограммы накладываются/суммируются и когда пересекаются. Далее используйте режим с пересечением.

1.6.3. Постройте гистограммы, чтобы вместо столбцов изображались ступеньки.

1.6.4. Добавьте на гистограммы график ядерной оценки плотности.

2. Изучение набора данных iris.csv с использованием NumPy:

2.1. Загрузите данные из файла как массив NumPy.

2.2. Выведите первые 10 наблюдений набора данных.

2.3. Рассчитайте характеристики полученные методом describe в п. 1.3 с использованием методов NumPy.

3. Изучение набора данных вашего варианта:

3.1. Оцените и опишите набор данных вашего варианта с использованием методов в п. 1.

4. Преобразование данных:
   4.1. Получите из датафрейма из п. 1.4 столбец с названием классов. Используя LabelEncoder и OneHotEncoder получите различные способы кодирования меток класса. В чем различия полученных кодировок?
   4.2. Для датафрейма из п. 1.4, получите все столбцы признаков (столбцы не содержащие метки классов). Преобразуйте полученные столбцы в массив NumPy.
   4.3. Для массива NumPy из п. 4.2 примените StandardScaler, MinMaxScaler, MaxAbsScaler и RobustScaler. Для каждого из результатов постройте гистограммы по каждому признаку без разделения по классам. В чем различия между такими преобразованиями данных?
   4.4. Согласно варианту, самостоятельно реализуйте StandardScaler или MinMaxScaler с использованием NumPy. Проверьте корректность работы на вашем наборе данных, сравните результаты между вашей реализацией и реализацией из Sklearn, а также рассчитав минимальное, максимальное, среднее значение и дисперсию, после преобразования

Выполнение работы.

1. Изучим набор данных *iris.csv* с использованием *Pandas* и *Seaborn*.

1.1. Загрузим данные из файла как *Pandas DataFrame*. Для этого воспользуемся методом *read_csv* (см. листинг 1.1). Методом *drop* удалим первый столбец, который не несет никакой информации, а просто является нумерацией наблюдений. Это необходимо чтобы в дальнейших пунктах не рисовать лишние графики. Здесь *axis=1* указывает на то, что мы удаляем столбец (*axis=0* используется для удаления строк). *inplace=True* - это параметр, который указывает Pandas модифицировать DataFrame на месте, то есть без необходимости присваивать результат обратно переменной. Если *inplace=False*

или не указано, то метод *drop()* вернет новый *DataFrame* с удаленным столбцом, не изменяя исходный *DataFrame*.

Листинг 1.1 - Загрузка данных из файла как *Pandas DataFrame*

```
1    df = pd.read_csv('iris.csv')
2    df.drop('Unnamed: 0', axis=1, inplace=True)
```

1.2. Вызвав у датафрейма метод *head*, проверим корректность загруженных данных (см. листинг 1.2). Аргументом данного метода является количество строк (наблюдений), которые необходимо вывести начиная сначала. Если аргумента нет, то по умолчанию выводятся первые 5 строк. Вывод метода *head* смотреть в таблице. 1.2.

Листинг 1.2 - Вызов метода *head*

```
1    df.head()
```

Таблица 1.2 - Вывод метода *head*

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Данные загружены корректно.

1.3. Вызвав у датафрейма метод *describe*, получим характеристики. (см. листинг 1.3). Результат работы метода представлен в таблице 1.3. Здесь *count* — это количество заполненных строк в каждом столбце, в нашем случае для всех

150 наблюдений заполнен каждый признак; *mean* — среднее значение по каждому столбцу; *std* — стандартное отклонение, важный статистический показатель, показывающий разброс значений; *min* и *max* — минимальное и максимальное значения по каждому столбцу; *25%, 50%* и *75%* — процентили. Значение процентили - это значение количественной переменной, которое разделяет упорядоченные данные на группы таким образом, что определенный процент наблюдений имеет значения этой количественной переменной меньше значения процентили, а другой процент наблюдений имеет значения этой количественной переменной больше значения процентили. Так например для первого признака *25%* от всех значений меньше, чем 5.1. Метод *describe* работает только с числовыми характеристиками.

Листинг 1.3 - Вызов метода *describe*

```
1    df.describe()
```

Таблица 1.3 - Вывод метода *describe*

Feature	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.84333333	3.05733333	3.75800000	1.19933333	1.00000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.30000000	2.00000000	1.00000000	0.10000000	0.00000000
25%	5.10000000	2.80000000	1.60000000	0.30000000	0.00000000
50%	5.80000000	3.00000000	4.35000000	1.30000000	1.00000000
75%	6.40000000	3.30000000	5.10000000	1.80000000	2.00000000
max	7.90000000	4.40000000	6.90000000	2.50000000	2.00000000

1.4. Видоизменим полученный датафрейм таким образом, чтобы метка

классов были следующими: *0 - Iris-setosa, 1 - Iris-versicolor, 2 - Iris-virginica*. За метки классов отвечает столбец *target*. Для видоизменения использовался словарь, в котором ключ - старая метка класса, значение - соответствующая ей новая метка (см. листинг 1.4). В столбец *target* были записаны новые значения. (см. табл. 1.4). Полученный датафрейм сохранен в отдельный файл формата *csv*.

Листинг 1.4 - Изменение меток класса датафрейма

```
target_change_map = {0: 'Iris-setosa', 1: 'Iris-versicolor',
2:'Iris-virginica'}

# Заменяем метки классов в столбце 'target' согласно словарю
df['target'] = df['target'].map(target_change_map)

# Сохраняем модифицированный DataFrame в отдельный файл
df.to_csv('iris_modified.csv')
df
```

Таблица 1.4 - Видоизмененный датафрейм

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
…	…	…	…	…	…
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica

149	5.9	3.0	5.1	1.8	Iris-virginica

1.5. Визуально оценим набор данных, построив изображение, содержащее графики ядерной оценки плотности каждого признака (кроме признака класса), диаграмму рассеяния и двумерную ядерную оценку плотности для каждых признаков (см. листинг 1.5). Наблюдения разных классов выделены отдельным цветом (палитра *'Set1'*). Объект *PairGrid,* который представляет собой сетку для визуализации парных отношений между различными признаками в *DataFrame df.* Параметр *diag_sharey=False* указывает, что графики на диагонали (для каждого признака отдельно) будут иметь разные шкалы по оси *y.* Параметр *hue="target"* указывает, что данные будут разделены по столбцу *"target"*, и каждый уникальный класс будет отображен разным цветом.

Метод *map_upper* отображает диаграммы рассеивания в верхнем треугольнике сетки графиков (параметр *s=15* указывает размер точек на графике). Метод *map_lower* отображает двумерную ядерную оценку плотности в нижнем треугольнике сетки графиков. Метод *map_diag* отображает графики ядерной оценки плотности на диагонали сетки графиков. Также была добавлена легенда (параметр *lw=2* устанавливает толщину линии для графиков).

Для отображения графиков использовалась библиотека *Seaborn* (см. рисунок 1.5).

Листинг 1.5 - Отображение графиков

```
g = sns.PairGrid(df, diag_sharey=False, hue="target")
# диаграммы рассеяния в верхнем треугольнике сетки графиков
g.map_upper(sns.scatterplot, s=15)
# двумерная ядерная оценка плотности
g.map_lower(sns.kdeplot)
# ядерная оценка плотности
g.map_diag(sns.kdeplot, lw=2)
g.add_legend()
```

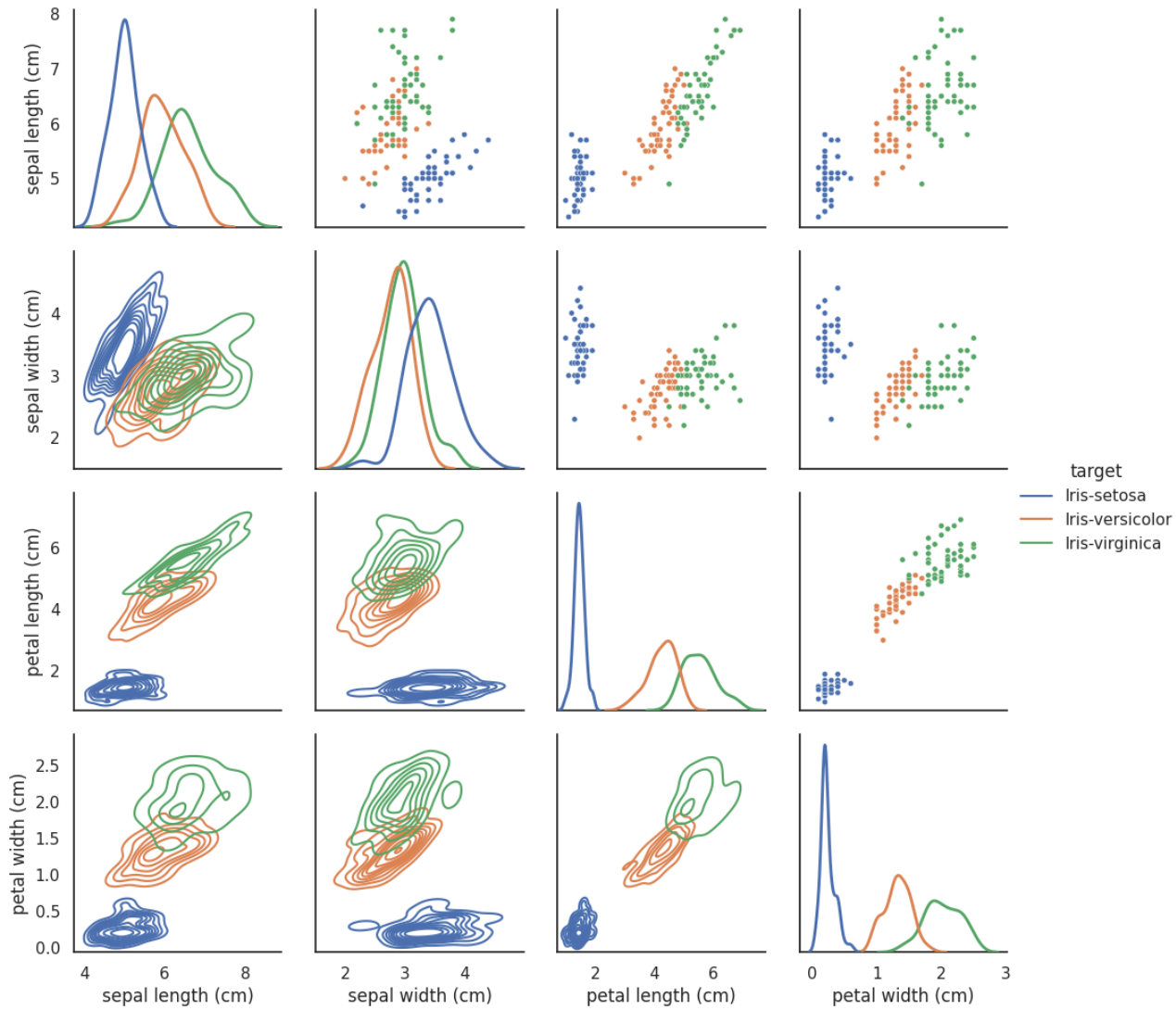

Рисунок 1.5 - Отображение графиков iris.csv

По графикам видно, что происходит смешение классов *iris-setosa* и *iris-versicolor* по всем признакам. У признака *sepal width (cm)* самое сильное смешивание.

1.6. На одном изображении построим гистограммы распределения для каждого признака (для построения нескольких диаграмм на одном изображении создан subplot из matplotlib, и для каждой диаграммы задан параметр ax, указав нужную ячейку. subplot возвращает два параметра: саму фигуру с изображением и список ячеек).

1.6.1. На этом одном изображении построены гистограммы для разного количества столбцов: *5,10,15,20,30* (см. листинг 1.6.1 и рис. 1.6.1).

Листинг 1.6.1 - Построение гистограмм

```
    # Параметр figsize=(20, 15) задает размеры изображения:
    # ширина 20 дюймов и высота 15 дюймов.
    fig, axs = plt.subplots(4, 5, figsize=(20, 15))

    bins_list = [5, 10, 15, 20, 30]

    colors = ['blue', 'salmon', 'green', 'orange', 'red']

    # Перебираем каждый признак и каждое количество столбцов
    # итерируемся по всем признакам, кроме последнего
    for i, feature in enumerate(df.columns[:-1]):
        for j, bins in enumerate(bins_list):
            sns.histplot(df, x = feature, bins=bins, ax=axs[i,
    j], kde=True, color=colors[j]))
```

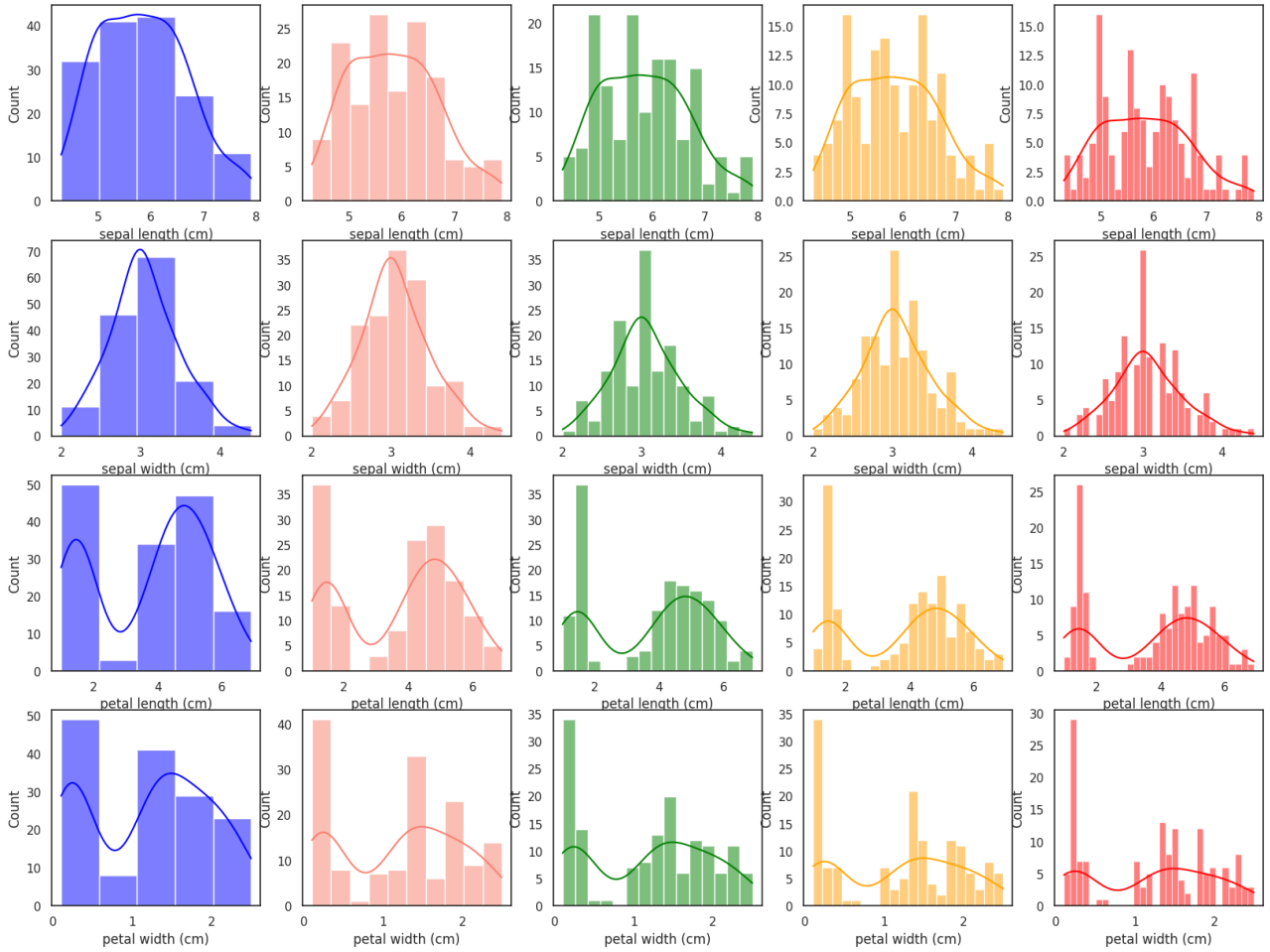

Рисунок 1.6.1 - Отображение гистограмм для *iris.csv*

Наиболее подходящее количество столбцов для наилучшего описания формы распределения признаков - 20 столбцов, так как при меньшем

количестве столбцов слишком большой диапазон значений для одного столбца, а при большем количестве столбцом слишком много пустых столбцов.

Далее для *iris.csv* будем рисовать гистограммы с количеством столбцов, равным 20.

1.6.2. Сделаем на каждой гистограмме разделение по цвету согласно классу. Проведем это в двух режимах, когда гистограммы накладываются/суммируются (см. листинг 1.6.2.1) и когда пересекаются (см. листинг 1.6.2.2). Разделение по цвету согласно классу осуществляется благодаря параметру *hue (*в датафрейме *iris.csv* это *target).* Если передать в метод *histplot* параметр *multiple='stack'*, то каждый следующий набор данных на гистограмме будет размещен сверху предыдущего (см. рис. 1.6.2.1), если вообще не указывать такой параметр, то умолчанию гистограммы будут пересекаться (см. рис. 1.6.2.2).

Листинг 1.6.2.1 - Построение гистограмм с наложением

```
1    # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
2    fig, axs = plt.subplots(1, 4, figsize=(20, 10))
3
4    bins = 20
5
6    # stack': каждый следующий набор данных будет размещен
7    # сверху предыдущего
8    for i, feature in enumerate(df.columns[:-1]):
9           sns.histplot(df, x = feature, hue='target',
10   bins=bins, ax=axs[i], multiple='stack')
```

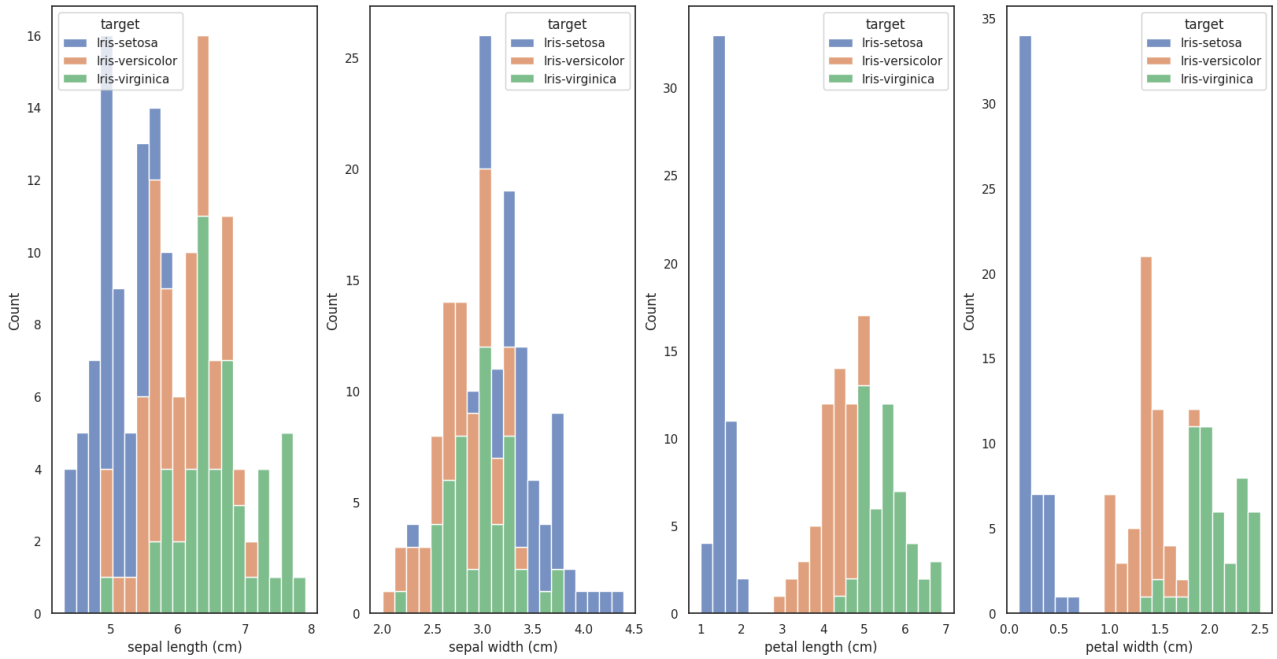

Рисунок 1.6.2.1 - Отображение гистограмм с наложением

Листинг 1.6.2.2 - Построение гистограмм с пересечением

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов
2     fig, axs = plt.subplots(1, 4, figsize=(20, 10))
3
4     bins = 20
5
6     for i, feature in enumerate(df.columns[:-1]):
7             sns.histplot(df, x = feature, hue='target',
8     bins=bins, ax=axs[i])
```

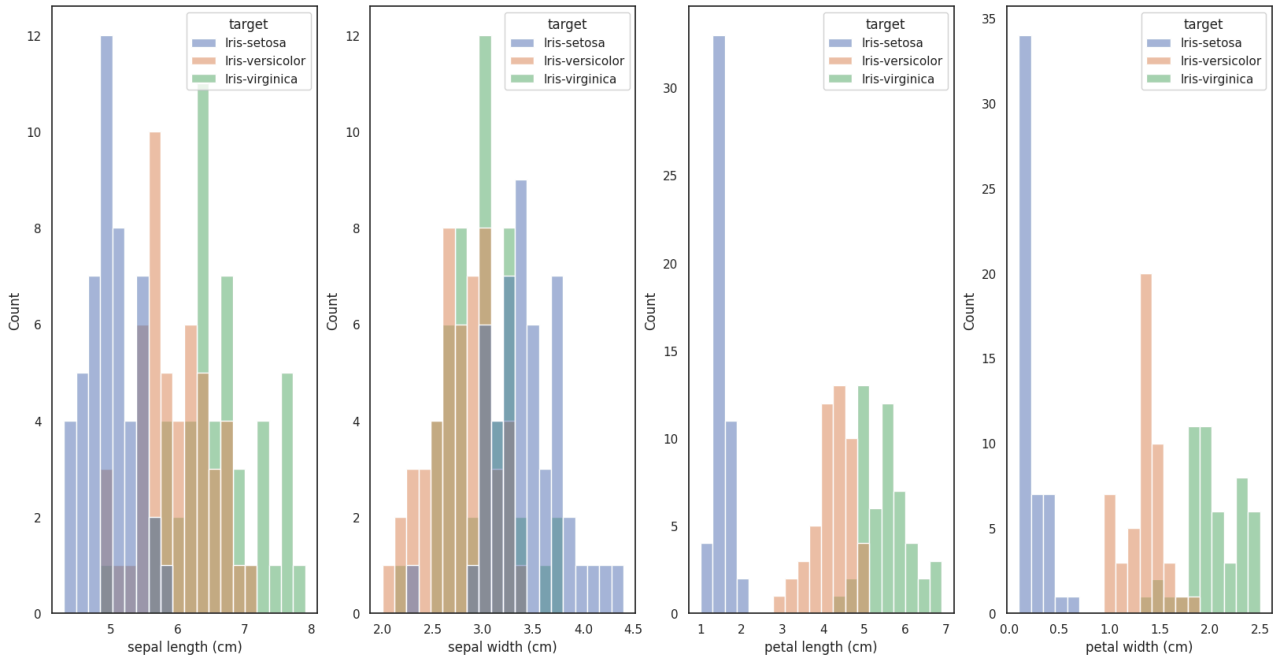

Рисунок 1.6.2.2 - Отображение гистограмм с пересечением

Далее будем использовать режим с пересечением.

1.6.3. Построим гистограммы, чтобы вместо столбцов изображались ступеньки. Единственное изменение по сравнению с предыдущим пунктом - параметр *element* метода *histplot* нужно установить в значение *“step”* (см. листинг 1.6.3). Отображение диаграммы смотреть на рисунке 1.6.3.

Листинг 1.6.3 - Построение гистограмм со ступеньками

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
2     fig, axs = plt.subplots(1, 4, figsize=(20, 10))
3
4     bins = 20
5
6     for i, feature in enumerate(df.columns[:-1]):
7             sns.histplot(df, x = feature, hue='target',
8     bins=bins, ax=axs[i], element = "step")
```

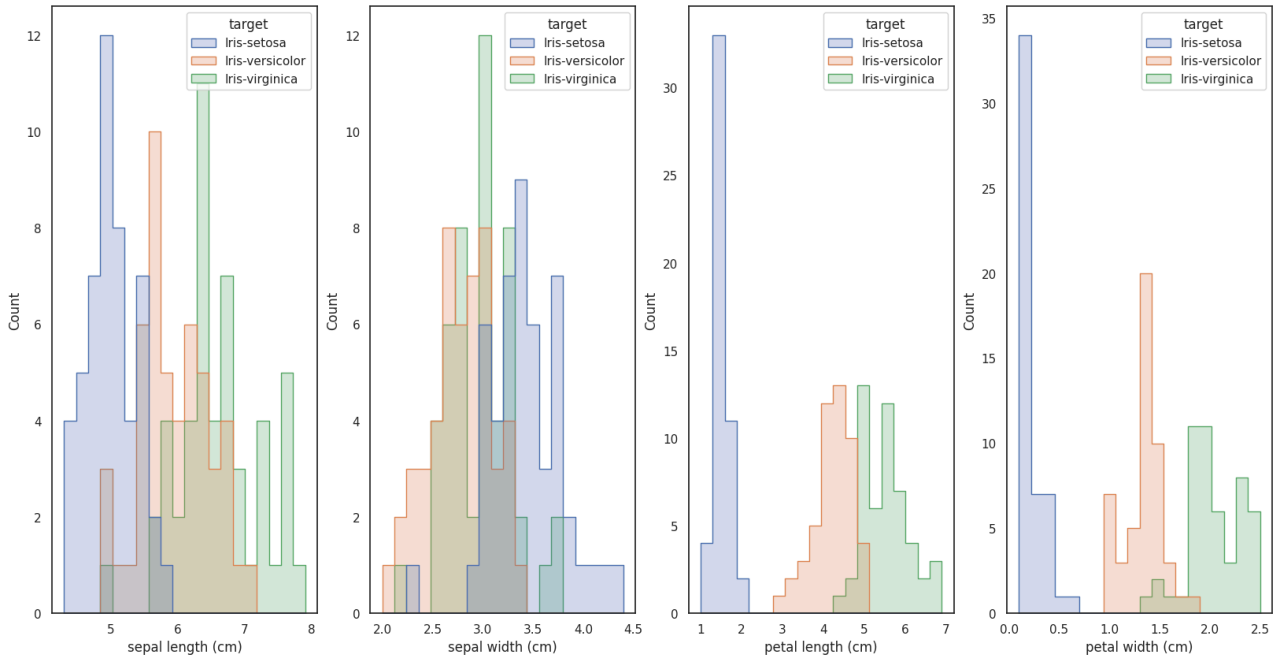

Рисунок 1.6.3- Отображение гистограмм со ступеньками

1.6.4. Добавим на гистограммы график ядерной оценки плотности. Единственное изменение по сравнению с предыдущим пунктом - параметр *kde* метода *histplot* нужно установить в значение *True* (см. листинг 1.6.4). Отображение диаграммы смотреть на рисунке 1.6.4.

Листинг 1.6.4 - Построение гистограмм с ядерной оценкой плотности

```
    # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
    fig, axs = plt.subplots(1, 4, figsize=(20, 10))

    bins = 20

    for i, feature in enumerate(df.columns[:-1]):
            sns.histplot(df, x = feature, hue='target',
    bins=bins, ax=axs[i], element="step", kde = True)
```

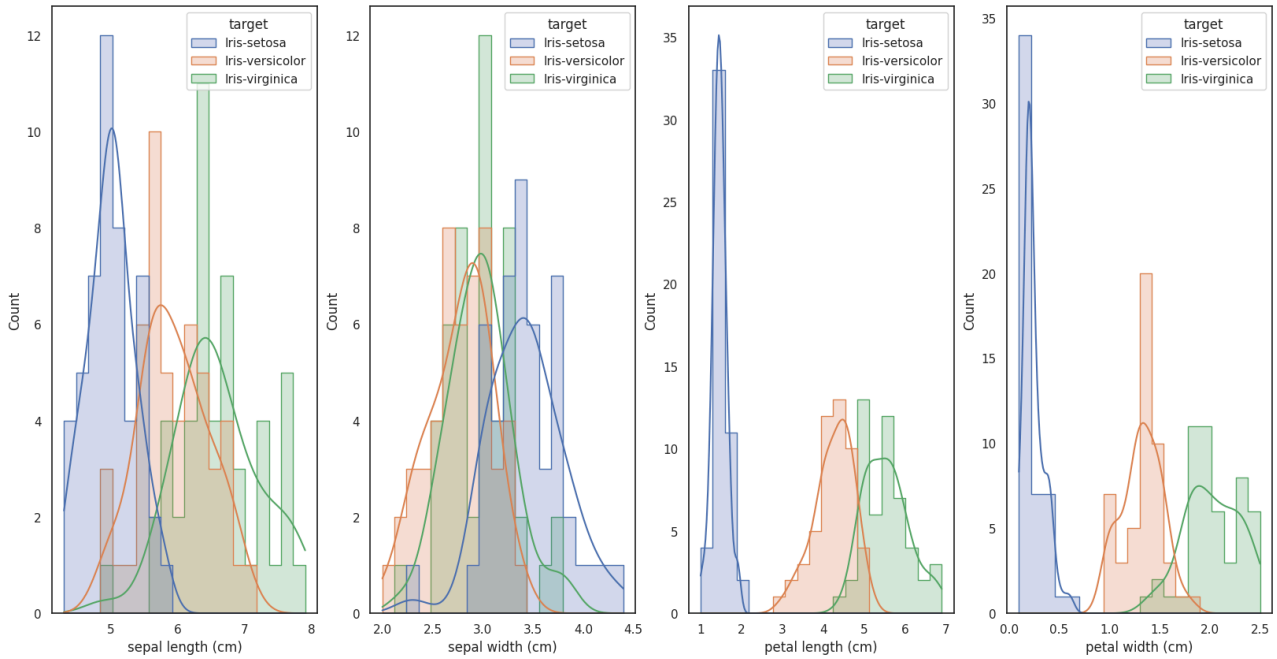

Рисунок 1.6.4- Отображение гистограмм с ядерной оценкой плотности

Графики ядерной оценки плотности на рисунке 1.6.4 и на рисунке 1.5 совпадают.

2. Изучим набор данных *iris.csv* с использованием *NumPy*.

2.1. Загрузим данные из файла *iris.csv* как массив *NumPy* (см. листинг 2.1). Здесь аргумент *delimiter* указывает разделитель, используемый в файле *CSV*. Аргумент *skip_header* указывает, сколько строк в начале файла нужно пропустить перед началом чтения данных (пропускаем первую строку, так как она содержит заголовки столбцов). Аргумент *dtype* определяет тип данных, который мы хотим использовать для хранения загружаемых значений. Аргумент *usecols* указывает, какие столбцы мы хотим загрузить из файла. Значения в скобках - это индексы столбцов (начиная с 0), которые мы хотим загрузить (загружаются столбцы с индексами 1, 2, 3, 4 и 5, так как первый столбец - это просто нумерация). Аргумент *filling_values* определяет, какие значения использовать для заполнения отсутствующих данных (если таковые имеются).

Листинг 2.1 - Загрузка данных из файла *iris.csv* как массив *NumPy*

```
1   data = np.genfromtxt('iris.csv', delimiter=',',
2   skip_header=1,dtype=float, usecols=(1,2,3,4,5),
3   filling_values=0)
```

2.2. Выведем первые 10 наблюдений набора данных (см. листинг 2.2). Результат вывода представлен в таблице 2.2.

Листинг 2.2 - Вывод первых 10 наблюдений

```
1    print("Первые 10 наблюдений:")
2    data[:10]
```

Таблица 2.2 - Вывод первых 10 наблюдений

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0
5	5.4	3.9	1.7	0.4	0.0
6	4.6	3.4	1.4	0.3	0.0
7	5.0	3.4	1.5	0.2	0.0
8	4.4	2.9	1.4	0.2	0.0
9	4.9	3.1	1.5	0.1	0.0

2.3. Рассчитаем характеристики полученные методом *describe* в п. 1.3 с использованием методов *NumPy* (см. листинг 2.3). Значение данных характеристик описано в пункте 1.3. Здесь в сравнении с пунктом 1.3

используются встроенные методы библиотеки *NumPy*: *np.count_nonzero* (*~np.isnan(data)* для создания булевого массива, который имеет значение True для каждого элемента в массиве data, который не является *NaN*), *np.mean, np.std, np.min, np.percentile, np.max*. *axis=0* означает агрегацию по столбцам. Рассчитанные характеристики представлены в таблице 2.3.

Листинг 2.3 - Расчет характеристик с использованием *NumPy*

```
    # axis=0 означает агрегацию по столбцам (по вертикали)

    # Количество ненулевых элементов в массиве.
    count = np.count_nonzero(~np.isnan(data), axis=0)
    # Среднее значение
    mean = np.mean(data, axis=0)
    # Стандартное отклонение
    std = np.std(data, axis=0)
    # Минимальное значение
    min_value = np.min(data, axis=0)
    # 25-й, 50-й (медиана) и 75-й процентили
    percentiles = np.percentile(data, [25, 50, 75], axis=0)
    # Максимальное значение
    max_value = np.max(data, axis=0)

    feature_names = ['sepal length (cm)', 'sepal width (cm)',
    'petal length (cm)', 'petal width (cm)', 'target']

    # Вывод результатов
    print("Feature names:", feature_names)
    print("count:", count)
    print("mean:", mean)
    print("std:", std)
    print("min:", min_value)
    print("25%:", percentiles[0])
    print("50%:", percentiles[1])
    print("75%:", percentiles[2])
    print("max:", max_value)
```

Таблица 2.3 - Рассчитанные характеристик с использованием *NumPy*

Feature	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.84333333	3.05733333	3.75800000	1.19933333	1.00000000

std	0.82530129	0.43441097	1.75940407	0.75969263	0.81649658
min	4.30000000	2.00000000	1.00000000	0.10000000	0.00000000
25%	5.10000000	2.80000000	1.60000000	0.30000000	0.00000000
50%	5.80000000	3.00000000	4.35000000	1.30000000	1.00000000
75%	6.40000000	3.30000000	5.10000000	1.80000000	2.00000000
max	7.90000000	4.40000000	6.90000000	2.50000000	2.00000000

Все рассчитанные характеристики с использованием *NumPy* совпали с рассчитанными значениями методом *describe* в *Pandas (*см. таблицу 1.3 и таблицу 2.3).

3. Изучим набор данных *lab1_var2.csv*.

3.1. Оценим и опишем набор данных *lab1_var2.csv* с использованием методов в п. 1. В данном пункте будем кратко описывать, что означает каждый параметр (так как подробно все описано в п. 1), построим новые графики, рассчитаем значения и проанализируем.

Загрузим данные из файла как *Pandas DataFrame*. Для этого воспользуемся методом *read_csv* (см. листинг 3.1). Методом *drop* удалим первый столбец, который не несет никакой информации, а просто является нумерацией наблюдений. Это необходимо чтобы в дальнейших пунктах не рисовать лишние графики. *inplace=True* - это параметр, который указывает Pandas модифицировать DataFrame на месте, то есть без необходимости присваивать результат обратно переменной.

Листинг 3.1 - Загрузка данных из файла как *Pandas DataFrame*

```
1    df = pd.read_csv('lab1_var2.csv')
2    df.drop('Unnamed: 0', axis=1, inplace=True)
```

3.2. Вызвав у датафрейма метод *head*, проверим корректность

загруженных данных (см. листинг 3.2). Если аргумента нет, то по умолчанию выводятся первые 5 строк. Вывод метода *head* смотреть в табл. 3.2.

Листинг 3.2 - Вызов метода *head*

```
1    df.head()
```

Таблица 3.2 - Вывод метода *head*

	A	B	C	D	E	label
0	3.820369	0.944005	-2.515361	2.691548	4.587330	1
1	2.317692	-4.031270	1.096187	-4.324973	4.395063	1
2	3.031215	2.313363	-3.358096	2.744431	3.922414	0
3	2.880855	-2.467154	2.550000	-2.215556	0.858642	1
4	3.674317	-5.525880	5.199131	1.230003	2.516113	1

Данные загружены корректно.

3.3. Вызвав у датафрейма метод *describe*, получим характеристики. (см. листинг 3.3). Результат работы метода представлен в табл. 3.3. Здесь *count* — это количество заполненных строк в каждом столбце, в нашем случае для всех 200 наблюдений заполнен каждый признак; *mean* — среднее значение по каждому признаку; *std* — стандартное отклонение, важный статистический показатель, показывающий разброс значений; *min* и *max* — минимальное и максимальное значения по каждому признаку; *25%, 50%* и *75%* — процентили.

Листинг 3.3 - Вызов метода *describe*

```
1    df.describe()
```

Таблица 3.3 - Вывод метода *describe*

	A	B	C	D	E	label
count	200.00000	200.00000	200.00000	200.00000	200.00000	200.00000
mean	3.059112	-0.050611	0.043850	-0.089783	3.123704	0.565000
std	1.167444	3.336619	3.126977	3.241638	1.337344	0.497001
min	0.153019	-6.179742	-5.131545	-7.268231	-0.458386	0.000000
25%	2.337088	-2.951520	-3.030783	-3.032036	2.279943	0.000000
50%	3.037541	0.129381	0.374458	1.189169	3.103441	1.000000
75%	3.804794	2.896585	2.984745	2.992634	3.977151	1.000000
max	6.560331	6.426588	5.199131	4.403511	7.304125	1.000000

3.4. Видоизменим полученный датафрейм таким образом, чтобы метка классов были следующими: *0 - zero, 1 - one*. За метки классов в данном наборе данных отвечает столбец *label*. Для видоизменения использовался словарь, в котором ключ - старая метка класса, значение - соответствующая ей новая метка (см. листинг 3.4). В столбец *label* были записаны новые значения. (см. табл. 3.4). Полученный датафрейм сохранен в отдельный файл формата *csv*.

Листинг 3.4 - Изменение меток класса датафрейма

```
label_change_map = {0: 'zero', 1: 'one’}

# Заменяем метки классов в столбце 'target' согласно словарю
df['label'] = df['label'].map(label_change_map)

# Сохраняем модифицированный DataFrame в отдельный файл
df.to_csv('lab1_var2_modified.csv')
df
```

Таблица 3.4 - Видоизмененный датафрейм

	A	B	C	D	E	label
0	3.820369	0.944005	-2.515361	2.691548	4.587330	one
1	2.317692	-4.031270	1.096187	-4.324973	4.395063	one
2	3.031215	2.313363	-3.358096	2.744431	3.922414	zero
3	2.880855	-2.467154	2.550000	-2.215556	0.858642	one
4	3.674317	-5.525880	5.199131	1.230003	2.516113	one
…	…	…	…	…	…	…
195	1.482293	-4.176655	3.859927	-2.859470	4.089271	one
196	4.982931	-6.177010	2.980846	0.937900	1.593777	one
197	2.940962	-2.367159	1.688151	-3.870112	3.095158	one
198	1.828126	-1.905102	3.746216	-4.497843	4.989654	one
199	3.787520	-2.933279	3.880500	-2.238534	1.709943	one

3.5. Визуально оценим набор данных, построив изображение, содержащее графики ядерной оценки плотности каждого признака (кроме признака класса), диаграмму рассеяния и двумерную ядерную оценку плотности для каждых признаков (см. листинг 3.5). Объект *PairGrid,* который представляет собой сетку для визуализации парных отношений между различными признаками в *DataFrame df.* Параметр *diag_sharey=False* указывает, что графики на диагонали (для каждого признака отдельно) будут иметь разные шкалы по оси *y.* Параметр *hue="label"* указывает, что данные будут разделены по столбцу *"label"*, и каждый уникальный класс будет отображен разным цветом.

Метод *map_upper* отображает диаграммы рассеивания в верхнем треугольнике сетки графиков (параметр *s=15* указывает размер точек на графике). Метод *map_lower* отображает оценку плотности в нижнем треугольнике сетки графиков. Метод *map_diag* отображает графики ядерной

оценки плотности на диагонали сетки графиков. Также была добавлена легенда (параметр *lw=2* устанавливает толщину линии для графиков).

Для отображения графиков использовалась библиотека *Seaborn* (см. рисунок 3.5).

Листинг 3.5 - Отображение графиков

```
1    g = sns.PairGrid(df, diag_sharey=False, hue="label")
2    # диаграммы рассеяния в верхнем треугольнике сетки графиков
3    g.map_upper(sns.scatterplot, s=15)
4    # двумерная ядерная оценка плотности
5    g.map_lower(sns.kdeplot)
6    # ядерная оценка плотности
7    g.map_diag(sns.kdeplot, lw=2)
8    g.add_legend()
```

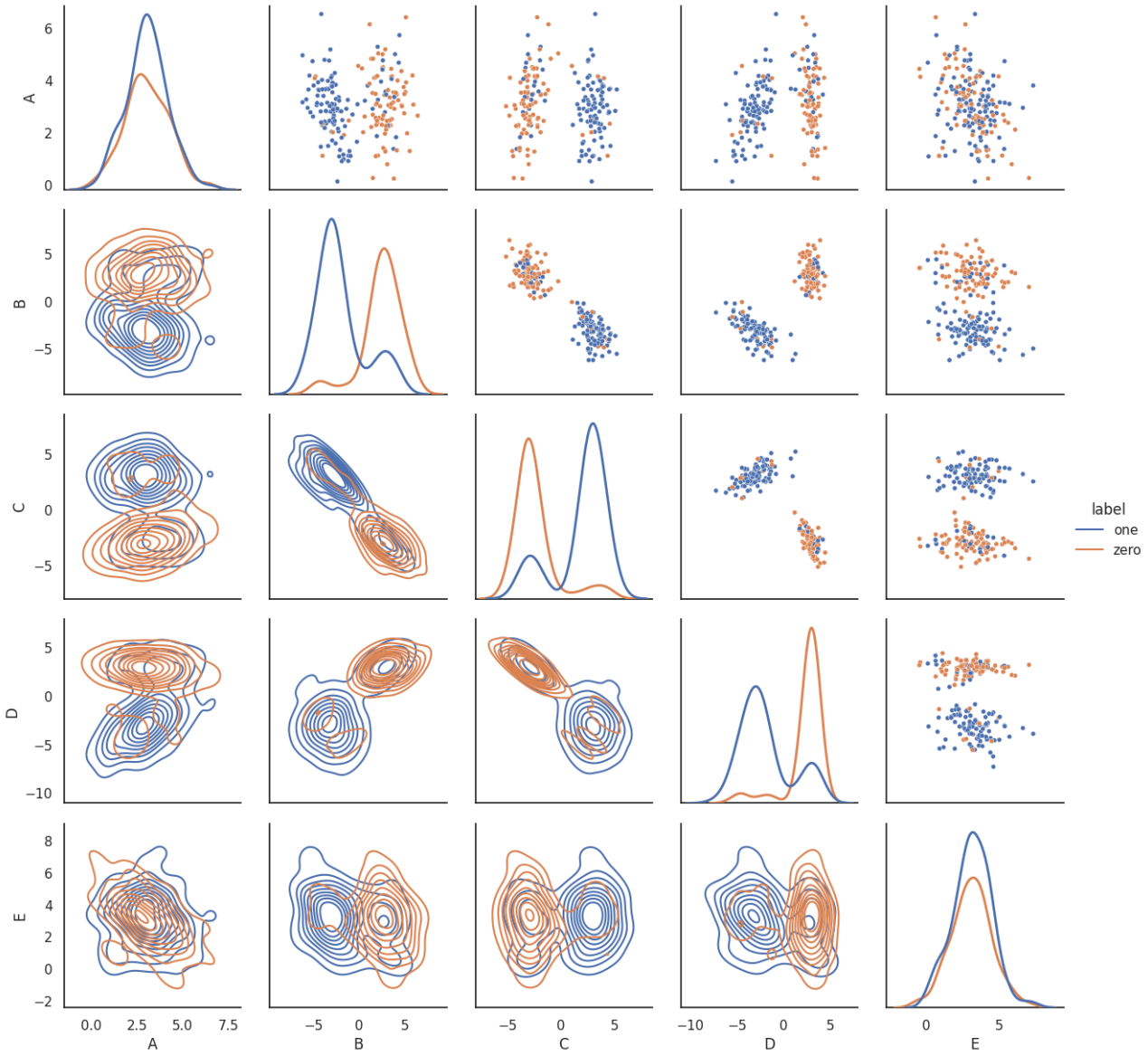

Рисунок 3.5 - Отображение графиков

По графикам видно, что происходит смешение классов *one* и *zero* по всем признакам. У признаков *E* и *A* самое сильное смешивание. Графики ядерной оценки плотности для признака *E* двух классов очень похожи.

3.6. На одном изображении построим гистограммы распределения для каждого признака (для построения нескольких диаграмм на одном изображении создан subplot из matplotlib, и для каждой диаграммы задан параметр ax, указав нужную ячейку. subplot возвращает два параметра: саму фигуру с изображением и список ячеек).

3.6.1. На этом одном изображении построены гистограммы для разного количества столбцов: *5,10,15,20,30* (см. листинг 3.6.1 и рис. 3.6.1).

Листинг 3.6.1 - Построение гистограмм

```
1    # Параметр figsize=(20, 15) задает размеры изображения:
2    # ширина 20 дюймов и высота 15 дюймов.
3    fig, axs = plt.subplots(5, 5, figsize=(20, 15))
4
5    bins_list = [5, 10, 15, 20, 30]
6
7    colors = ['blue', 'salmon', 'green', 'orange', 'red']
8
9    # Перебираем каждый признак и каждое количество столбцов
10   # итерируемся по всем признакам, кроме последнего
11   for i, feature in enumerate(df.columns[:-1]):
12       for j, bins in enumerate(bins_list):
13           sns.histplot(df, x = feature, bins=bins, ax=axs[i,
14   j], kde=True, color=colors[j]))
```

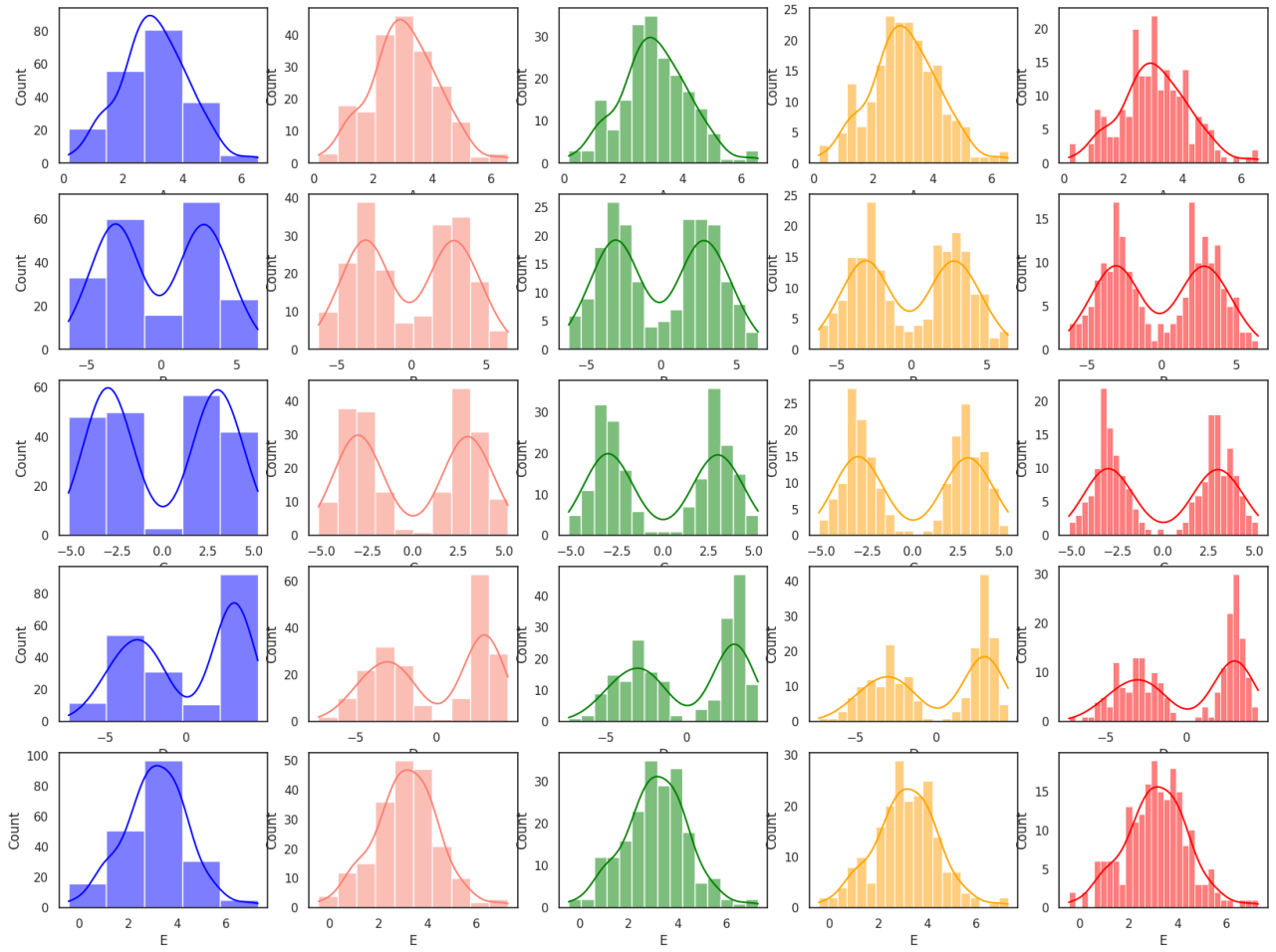

Рисунок 3.6.1 - Отображение гистограмм

Наиболее подходящее количество столбцов для наилучшего описания формы распределения признаков - 20 столбцов, так как при меньшем количестве столбцов слишком большой диапазон значений для одного столбца, а при большем количестве столбцом слишком много пустых столбцов.

Далее будем рисовать гистограммы с количеством столбцов, равным 20.

3.6.2. Сделаем на каждой гистограмме разделение по цвету согласно классу.

Проведем это в двух режимах, когда гистограммы накладываются/суммируются (см. листинг 3.6.2.1) и когда пересекаются (см. листинг 3.6.2.2). Разделение по цвету согласно классу осуществляется благодаря параметру *hue (*в данном датафрейме это *label)*. Если передать в метод *histplot* параметр *multiple='stack'*, то каждый следующий набор данных

на гистограмме будет размещен сверху предыдущего (см. рис. 3.6.2.1), если вообще не указывать такой параметр, то умолчанию гистограммы будут пересекаться (см. рис. 3.6.2.2).

Листинг 3.6.2.1 - Построение гистограмм с наложением

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
2     fig, axs = plt.subplots(1, 5, figsize=(20, 10))
3
4     bins = 20
5
6     # stack': каждый следующий набор данных будет размещен
7     # сверху предыдущего
8     for i, feature in enumerate(df.columns[:-1]):
9             sns.histplot(df, x = feature, hue='label',
10    bins=bins, ax=axs[i], multiple='stack')
```

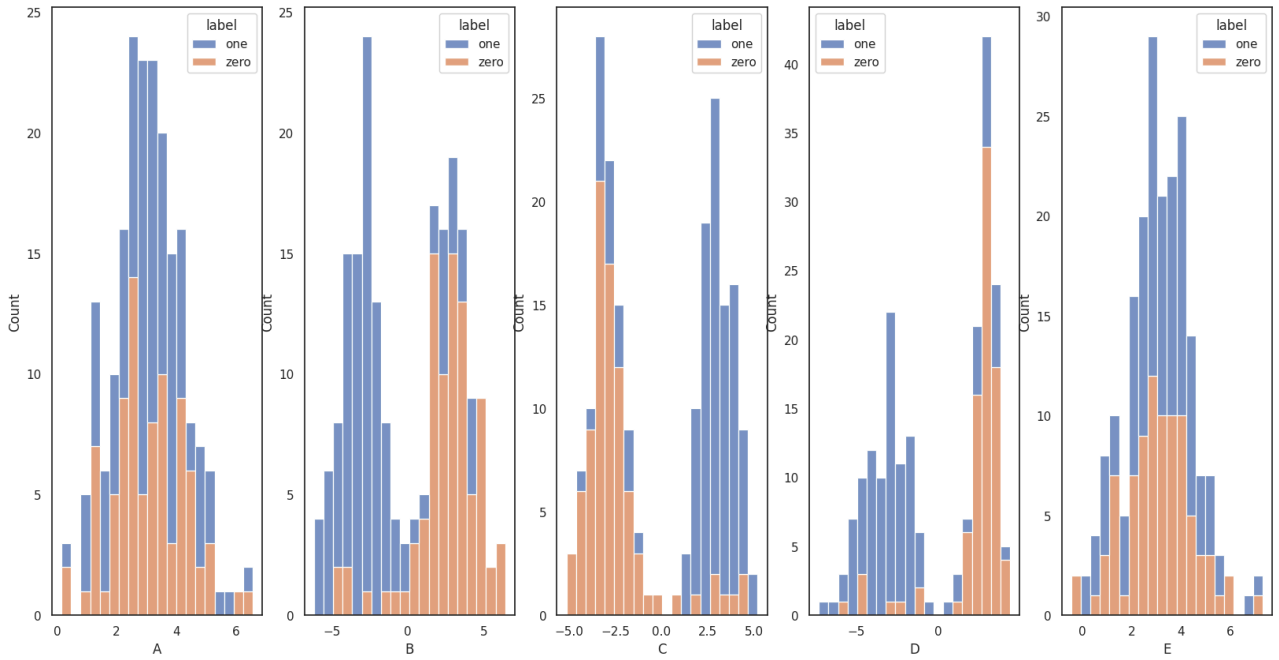

Рисунок 3.6.2.1 - Отображение гистограмм с наложением

Листинг 3.6.2.2 - Построение гистограмм с пересечением

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов
2     fig, axs = plt.subplots(1, 5, figsize=(20, 10))
3
4     bins = 20
5
```

```
6     for i, feature in enumerate(df.columns[:-1]):
7             sns.histplot(df, x = feature, hue='label',
8     bins=bins, ax=axs[i])
```

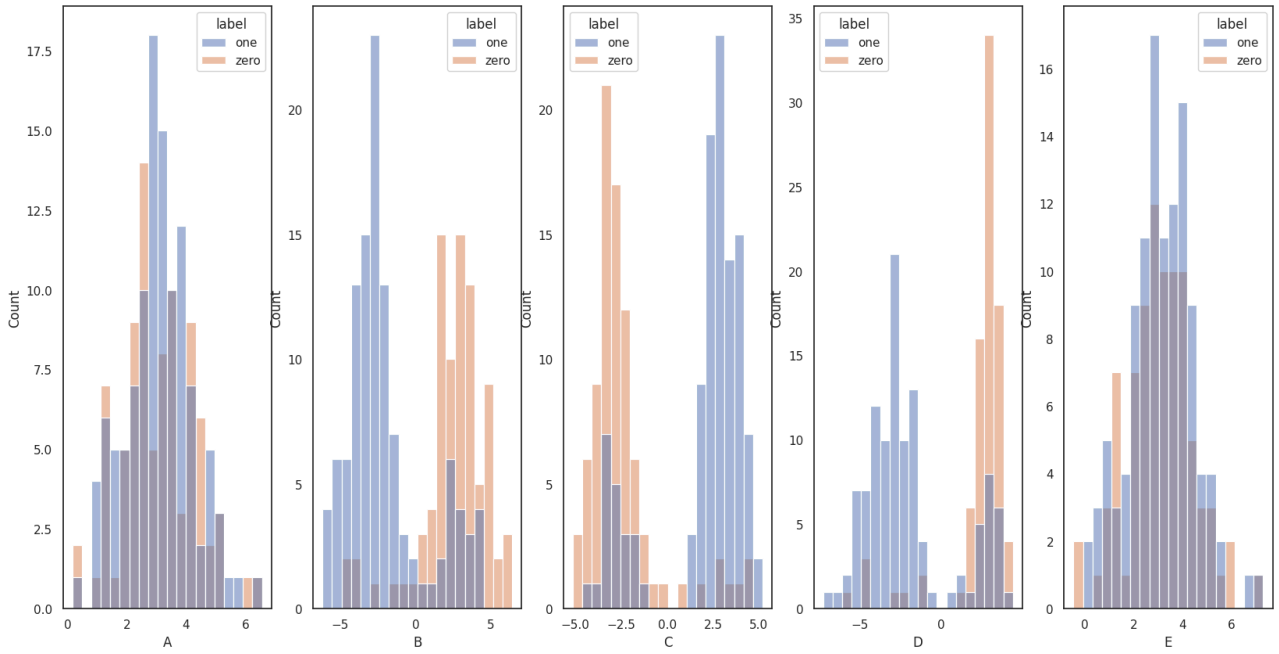

Рисунок 3.6.2.2 - Отображение гистограмм с пересечением

Далее будем использовать режим с пересечением.

3.6.3. Построим гистограммы, чтобы вместо столбцов изображались ступеньки. Единственное изменение по сравнению с предыдущим пунктом - параметр *element* метода *histplot* нужно установить в значение *“step”* (см. листинг 3.6.3). Отображение диаграммы смотреть на рисунке 3.6.3.

Листинг 3.6.3 - Построение гистограмм со ступеньками

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
2     fig, axs = plt.subplots(1, 5, figsize=(20, 10))
3
4     bins = 20
5
6     for i, feature in enumerate(df.columns[:-1]):
7             sns.histplot(df, x = feature, hue='label',
```

```
8     bins=bins, ax=axs[i], element = "step")
```

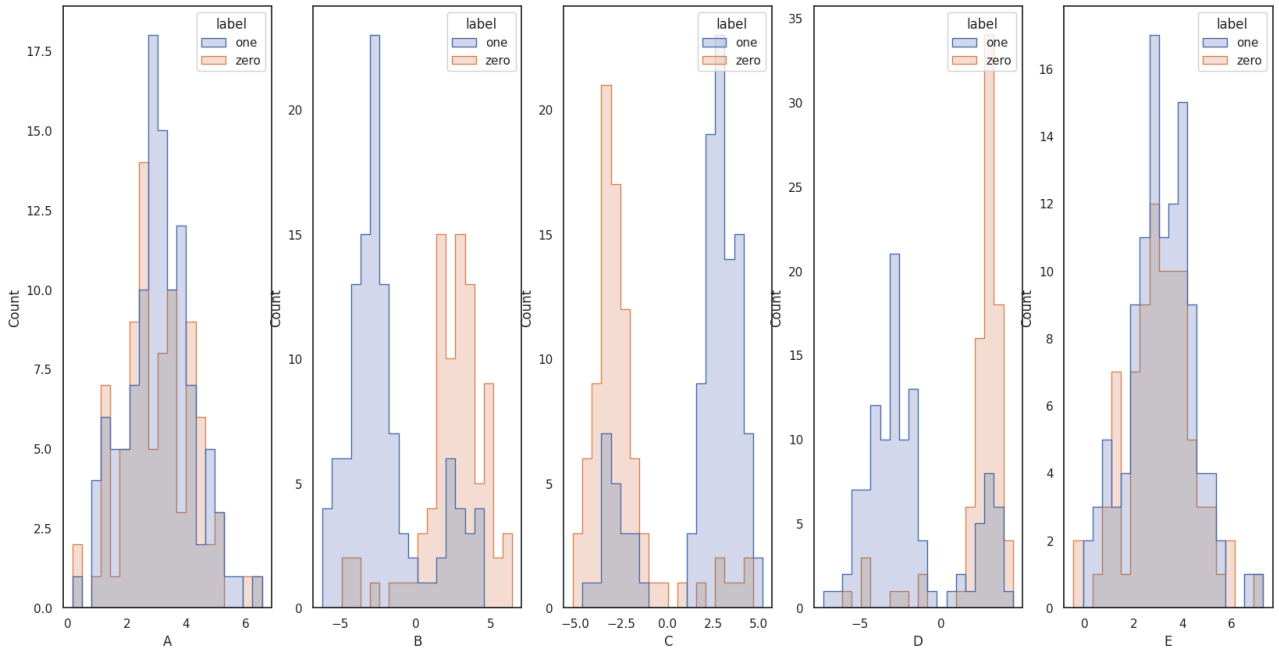

Рисунок 3.6.3- Отображение гистограмм со ступеньками

3.6.4. Добавим на гистограммы график ядерной оценки плотности. Единственное изменение по сравнению с предыдущим пунктом - параметр *kde* метода *histplot* нужно установить в значение *True* (см. листинг 3.6.4). Отображение диаграммы смотреть на рисунке 3.6.4.

Листинг 3.6.4 - Построение гистограмм с ядерной оценкой плотности

```
1     # размеры изображения: ширина 20 дюймов и высота 10 дюймов.
2     fig, axs = plt.subplots(1, 5, figsize=(20, 10))
3
4     bins = 20
5
6     for i, feature in enumerate(df.columns[:-1]):
7           sns.histplot(df, x = feature, hue='label',
8     bins=bins, ax=axs[i], element="step", kde = True)
```

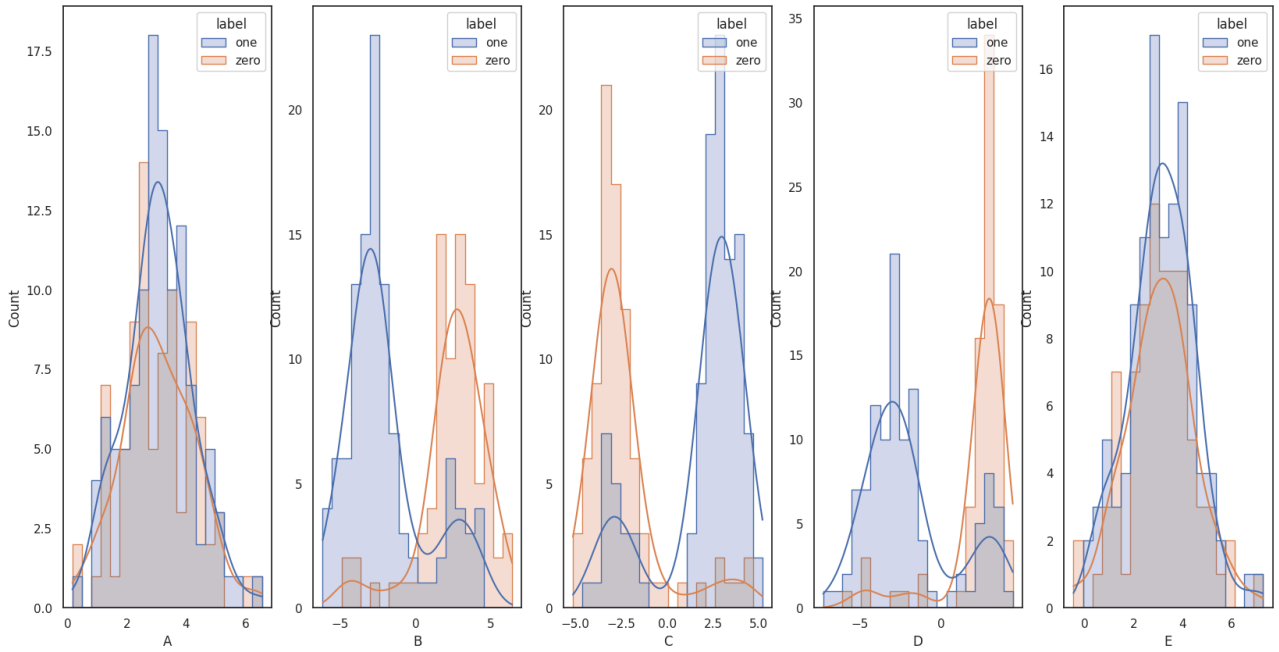

Рисунок 3.6.4- Отображение гистограмм с ядерной оценкой плотности

Графики ядерной оценки плотности на рисунке 3.6.4 и на рисунке 3.5 совпадают.

4.

4.1. Получим из датафрейма из п. 1.4 (*iris_modified.csv*) столбец с названием классов. Используя *LabelEncoder* (см. листинг 4.1.1) и *OneHotEncoder* (см. листинг 4.1.2) получим различные способы кодирования меток класса (см. таблицы 4.1.1 и 4.1.2).

Листинг 4.1.1 - *LabelEncoder*

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

# Преобразование числовых меток классов в датафрейме
df['target'] = le.fit_transform(df['target'])
```

Здесь создается экземпляр *LabelEncoder* и происходит преобразование категориальных меток классов в числовые значения и замена их в столбце *'target'* в *DataFrame 'df'*.

Таблица 4.1.1 - Результат кодирования *LabelEncoder*

target (до кодирования)	target (после кодирования)
Iris-setosa	0
Iris-versicolor	1
Iris-virginica	2

Листинг 4.1.2 - *OneHotEncoder*

```
1   from sklearn.preprocessing import OneHotEncoder
2
3   onehot_encoder = OneHotEncoder(sparse=False)
4
5   onehot_encoded = onehot_encoder.fit_transform(df[['target']])
6
7   encoder_df = pd.DataFrame(onehot_encoded)
8
9   final_df = df.join(encoder_df)
10  final_df.drop('target', axis=1, inplace=True)
```

Здесь создается объект *OneHotEncoder*, затем кодируются значения столбца *target*, происходит преобразование закодированных данных в датафрейм и присоединение к исходному датафрейму. Затем удаляется исходный столбец *target*.

Таблица 4.1.2 - Результат кодирования *OneHotEncoder*

target (до кодирования)	0	1	2
Iris-setosa	1.0	0.0	0.0
Iris-versicolor	0.0	1.0	0.0
Iris-virginica	0.0	0.0	1.0

Можно заметить, что *LabelEncoder* - это унитарное кодирование, каждый класс кодируется целым числом, начиная с 0. *OneHotEncoder* кодировщик берёт

столбец с категориальными данными, который был предварительно закодирован в признак и создает для него несколько новых столбцов. Числа заменяются на единицы и нули, в зависимости от того, какому столбцу какое значение присуще (в каждом столбце 1 - принадлежность признака к классу, 0 - не принадлежность). В нашем примере мы получим три новых столбца с названиями 0, 1, 2 .

LabelEncoder плохо подходит в случае, если классов >2, так как моделям тяжело подстроиться под множество дискретных значений. В зависимости от имеющихся у нас данных мы можем столкнуться с ситуацией, когда после кодирования признаков наша модель запутается, ложно предположив, что данные связаны порядком или иерархией, которого на самом деле нет Например, мы закодировали 0, 1, 2, модель выдает 1.5, что с ее точки зрения означает что-то между 1 и 2, хотя на самом деле вероятность принадлежности классу 0 такая же, как и вероятность принадлежности классам 1 или 2. Чтобы этого избежать используется *OneHotEncoder*. Также такое расширенное кодирование делает возможным легко определить, что вообще не принадлежит никакому классу.

4.2. Для датафрейма из п. 1.4, получим все столбцы признаков (столбцы не содержащие метки классов). Преобразуем полученные столбцы в массив *NumPy* (см. листинг 4.2).

axis=1 - указывает, что мы удаляем столбец, а не строку, *inplace=True* - это параметр, который указывает *Pandas* модифицировать *DataFrame* на месте, то есть без необходимости присваивать результат обратно переменной. Для преобразования использовался метод *to.numpy()*.

Листинг 4.2 - Преобразование датафрейма в массив *NumPy*

```
df = pd.read_csv('iris_modified.csv')
df.drop('Unnamed: 0', axis=1, inplace=True)

```

```
numpy_arr = df[["sepal length (cm)", "sepal width (cm)",
"petal length (cm)", "petal width (cm)"]].to_numpy()
numpy_arr
```

4.3. Для массива *NumPy* из п. 4.2 применим *StandardScaler, MinMaxScaler, MaxAbsScaler* и *RobustScaler* (см. листинг 4.3). Для каждого из результатов построим гистограммы по каждому признаку без разделения по классам (см. рис. 4.3).

Листинг 4.3 - Реализаия *StandardScaler, MinMaxScaler, MaxAbsScaler и RobustScaler*

```
from sklearn.preprocessing import StandardScaler,
MinMaxScaler, MaxAbsScaler, RobustScaler

scalers = [StandardScaler(), MinMaxScaler(), MaxAbsScaler(),
RobustScaler()]
scaler_names = ["StandardScaler", "MinMaxScaler",
"MaxAbsScaler", "RobustScaler"]

fig, axs = plt.subplots(4, len(numpy_arr[0]), figsize=(20,
20))
features = ["sepal length (cm)", "sepal width (cm)", "petal
length (cm)", "petal width (cm)"]

for i, scaler in enumerate(scalers):
    scaled_data = scaler.fit_transform(numpy_arr)
    for j in range(len(numpy_arr[0])):
      # [:, j] означает извлечение всех строк и только j-го
столбца.
        axs[i, j].hist(scaled_data[:, j], bins=20)
        axs[i, j].set_title(f"{scaler_names[i]}:
{features[j]}")

plt.show()
```

Здесь список *scalers* содержит экземпляры каждого из методов предобработки данных. Создание фигуры и отрисовка графиков на сетке осуществляется в с помощью методов *subplots* и *hist*.

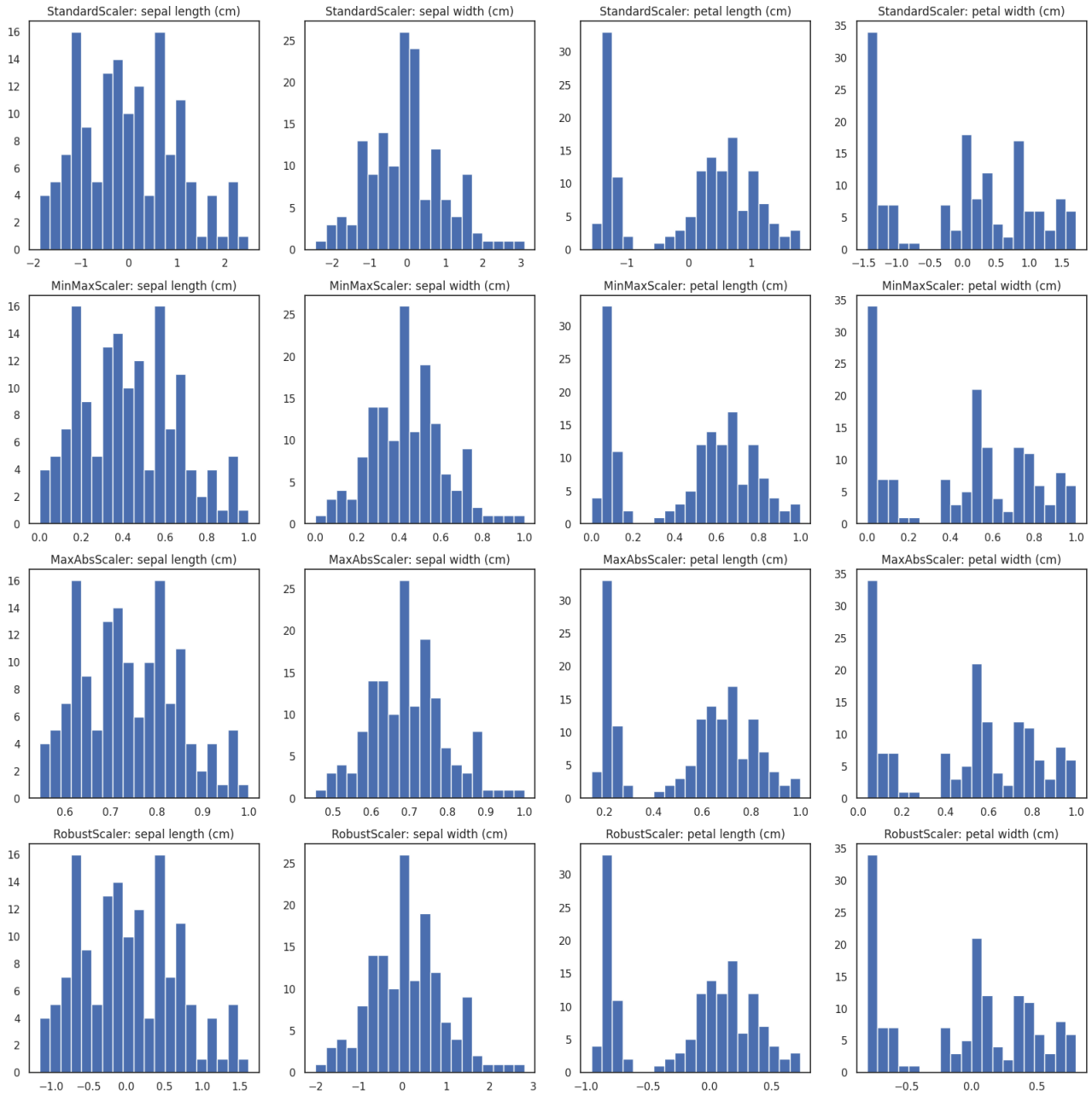

Рисунок 4.3- Отображение гистограмм *StandardScaler, MinMaxScaler, MaxAbsScaler и RobustScaler* по каждому признаку

Рассмотрим все виды предобработки данных.

Первая строчка гистограмм отображает стандартизацию, используется *StandardScaler.* Данная предобработка используется, когда для лучшей работы алгоритма необходимо, чтобы признаки имели одинаковое мат. ожидание и стандартное отклонение. Процедура стандартизации заключается в

центрировании по среднему и делении на стандартное отклонение. Такое преобразование значений можно охарактеризовать формулой 4.3.1.

$$\widehat{X} = \frac{X - EX}{\sigma X}, \qquad (4.3.1)$$

где EX - математическое ожидание, σX - стандартное отклонение.

Вторая строчка гистограмм отображает нормировку, используется *MinMaxScaler*. Разные признаки могут иметь разный порядок величин, из-за чего затрудняется построение модели и последующий анализ. Нормировка используется, чтобы все значения находились в некотором диапазоне и не было больших разбросов значений (чаще всего от 0 до 1). Такое преобразование значений можно охарактеризовать формулой 4.3.2.

$$\widehat{X} = \frac{X - min(X)}{max(x) - min(X)}, \qquad (4.3.2)$$

где $min(X)$ - минимальное значение, $max(X)$ - максимальное значение.

Третья строчка гистограмм отображает нормировку по модулю, используется *MaxAbsScaler*. Нормировка используется, чтобы все значения находились в некотором диапазоне, но иногда необходимо сохранить знак величин и в таких случаях *MinMaxScaler* не подходит, поэтому используют *MaxAbsScaler* (диапазон чаще всего от -1 до 1). Такое преобразование значений можно охарактеризовать формулой 4.3.3.

$$\widehat{X} = \frac{X}{max(|X|)}, \qquad (4.3.3)$$

где $max(|X|)$ - максимальное значение по модулю.

Четвертая строчка гистограмм отображает устойчивую к выбросам нормировку, используется *RobusScaler*. Выбросы данных могут испортить нормировку, и сжать несколько точек в одну. Решением является центрировать относительно медианы и разделить на разницу между 1 и 3 квартилью. Такое преобразование значений можно охарактеризовать формулой 4.3.4.

$$\widehat{X} = \frac{X - Q_2}{Q_3 - Q_1}, \qquad (4.3.4)$$

где Q_1, Q_2, Q_3 - 1-я, 2-я и 3-я квартилии.

Из рисунка 4.3 видно, что предобработка данных не меняет внешний вид гистограмм, меняется лишь ось абсцисс в соответствии с выбранной нормировкой.

4.4. Реализуем свою функцию *StandardScaler* с использованием *NumPy* для *lab1_var2.csv*. Сначала преобразуем датафрейм *lab1_var2.csv* в массив *NumPy (*см. листинг 4.4.1). Реализуем функцию *my_standard_scaler* и вызовем ее, передав в качестве аргумента *lab1_var2.csv* в качестве массива *NumPy (*см. листинг 4.4.2). Первые 10 строк результата работы функции *my_standard_scaler* представлены в таблице 4.4.1.

Листинг 4.4.1 - Преобразование *lab1_var2.csv* в массив *NumPy*

```
1    df = pd.read_csv('lab1_var2.csv')
2    df.drop('Unnamed: 0', axis=1, inplace=True)
3
4    numpy_arr = df[["A", "B", "C", "D", "E"]].to_numpy()
```

Листинг 4.4.2 - Собственная реализация функции *StandardScaler*

```
1   def my_standard_scaler(X):
2       mean = np.mean(X, axis=0)
3       std = np.std(X, axis=0)
4       scaled_X = (X - mean) / std
5       return scaled_X
6
7   scaled_numpy_arr = my_standard_scaler(numpy_arr)
8   scaled_numpy_arr
```

Таблица 4.4.1 - Результат работы функции *my_standard_scaler*

0.65370792	0.29883889	-0.82048335	0.86015492	1.09717429
-0.63667314	-1.19601567	0.33737947	-1.30977553	0.95304529

-0.02395574	0.7102716	-1.09066434	0.87650957	0.59873519
-0.15307289	-0.72606639	0.8034723	-0.65741671	-1.69795204
0.5282898	-1.64508101	1.65278405	0.40815722	-0.45546646
1.23900506	0.78465749	-1.07717626	0.84257701	-0.16229561
-0.33087659	1.94611746	-1.55870853	1.22255994	0.21434658
-0.10616989	-0.39855046	0.66458099	-1.33170926	0.7657284
1.47809421	-0.96599028	0.79320327	-0.41659709	-0.22186199
-1.76102654	-0.48925647	1.13510998	-1.25995768	0.31240354

Вызовем уже готовую реализацию метода *StandardScaler* из *Sklearn* (см. листинг 4.4.3). Первые 10 строк результата работы функции представлены в таблице 4.4.3.

Листинг 4.4.3 - Вызов встроенной функции *StandardScaler*

```
1   scaler = StandardScaler()
2   data = scaler.fit_transform(numpy_arr)
3   data
```

Таблица 4.4.3 - Результат работы встроенной функции *StandardScaler*

0.65370792	0.29883889	-0.82048335	0.86015492	1.09717429
-0.63667314	-1.19601567	0.33737947	-1.30977553	0.95304529
-0.02395574	0.7102716	-1.09066434	0.87650957	0.59873519
-0.15307289	-0.72606639	0.8034723	-0.65741671	-1.69795204
0.5282898	-1.64508101	1.65278405	0.40815722	-0.45546646
1.23900506	0.78465749	-1.07717626	0.84257701	-0.16229561
-0.33087659	1.94611746	-1.55870853	1.22255994	0.21434658
-0.10616989	-0.39855046	0.66458099	-1.33170926	0.7657284

1.47809421	-0.96599028	0.79320327	-0.41659709	-0.22186199
-1.76102654	-0.48925647	1.13510998	-1.25995768	0.31240354

Сравнив две таблицы 4.4.1 и 4.4.3, можно сделать вывод, что они все значения в двух таблицах одинаковы. Собственная реализация функции *StandardScaler* корректна.

Рассчитаем минимальное, максимальное, среднее значение и дисперсию (см. листинг 4.4.4). Результат представлен в таблице 4.4.4.

Листинг 4.4.4 - Расчет характеристик после стандартизации

```
# Минимальное значение
min_value = np.min(scaled_numpy_arr, axis=0)
# Максимальное значение
max_value = np.max(scaled_numpy_arr, axis=0)
# Среднее значение
mean = np.mean(scaled_numpy_arr, axis=0)
# Дисперсия
variance_values = np.var(scaled_numpy_arr, axis=0)

feature_names = ['A', 'B', 'C', 'D', 'E']

# Вывод результатов
print("Feature names:", feature_names)
print("min:", min_value)
print("max:", max_value)
print("mean:", mean)
print("var:", variance_values)
```

Таблица 4.4.4 - Расчет характеристик после стандартизации

	A	B	C	D	E
min	-2.49552467	-1.84153829	-1.65923243	-2.22000809	-2.68523159
max	3.00657182	1.94611746	1.65278405	1.38959693	3.13375724
mean	8.77076189e-17	-2.22044605e-18	-4.44089210e-18	-4.44089210e-18	9.76996262e-17

var	1	1	1	1	1

Вывод.

В ходе выполнения лабораторной работы были исследованы 2 набора данных. Для анализа, исследования, предобработки данных применялись библиотеки *Pandas, Seaborn, Numpy, Matplotlib* и *Scikit-learn.* Корректность значений проверялось методом *head()*, описание характеристик методом *describe()*. Для построения нескольких диаграмм на одном изображении был создан *subplot* из *matplotlib*. Были построены гистограммы с разным количеством столбцов и после анализа выбрано оптимальное количество. Также один и тот же набор исследовался с помощью *Pandas* и с помощью *NumPy*, анализ и все характеристики совпали. Были исследованы способы кодирования (*LabelEncoder* и *OneHotEncoder*) и их различия. Также на практике было рассмотрено 4 способа предобработки данных - StandardScaler, MinMaxScaler, MaxAbsScaler и RobustScaler. После любых из этих преобразований внешний вид гистограммы не менялся, менялась лишь ось абсцисс в соответствии с выбранной нормировкой. Также была реализована собственная функция *StandardScaler* и ее корректность была проверена с помощью встроенной функции *StandardScaler* из *Sklearn*.