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

ОТЧЁТ

по лабораторной работе № 4
по дисциплине «Основы машинного обучения»
Тема: «Классификация»

Студент гр.1304 _____ Поршнев Р.А.
Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2024

Задание

Вариант 4

1. Загрузка данных.

1.1. Загрузите данные вашего варианта. Учтите, что метки классов являются текстом.

1.2. Визуализируйте данные при помощи диаграммы рассеяния с выделением различных классов ней.

1.3. Оцените сбалансированность классов.

1.4. Проведите предобработку данных при необходимости.

1.5. Разделите выборку на обучающую и тестовую. На обучающей проводите обучение модели, на тестовой расчет значений метрик.

2. kNN.

2.1. Проведите классификацию методом k ближайших соседей, подобрав параметры: количество соседей, необходимость взвешивани. Постройте графики зависимости точности (Accuracy) в зависимости от количества соседей (по 1-й линии для взвешенного и не взвешенного метода).

2.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

2.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

2.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

2.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

3. Логистическая регрессия.

3.1. Проведите классификацию методом логистической регрессии при различных параметрах: без регуляризации, с l1 регуляризацией, с l2 регуляризацией. Постройте столбчатую диаграмму зависимости точности (Accuracy) от наличия регуляризации. Дальнейшие пункты 3.* выполняйте для лучшего параметра.

3.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

3.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

3.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

3.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

4. Метод опорных векторов.

4.1. Проведите классификацию методом опорных векторах при различных параметрах ядра: “linear”, “poly” (нужно выбрать степень), “rbf”. Постройте столбчатую диаграмму зависимости точности (Accuracy) от вида ядра. Дальнейшие пункты 4.* выполняйте для лучшего параметра.

4.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

4.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

4.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

4.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

5. Решающие деревья.

5.1. Проведите классификацию используя решающие деревья, подобрав параметры при которых получается лучшее обобщение (максимальная глубина/максимальное количество листьев/метрика загрязнения и т.д.).

5.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

5.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

5.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

5.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

5.6. Визуализируйте полученное дерево решений. Сделайте выводы о правилах в узлах дерева. Сопоставьте их с полученными границами принятия решений.

6. Решающие деревья.

6.1. Постройте таблицу с метриками Precision, Recall, AUC для полученных результатов каждым классификатором.

6.2. Сделайте выводы о том, какие классификаторы лучше всего подходят для вашего набора данных и в каких случаях.

Выполнение работы.

1. Загрузка данных.

1.1. Загрузите данные вашего варианта. Учтите, что метки классов являются текстом.

Для загрузки данных необходимо импортировать библиотеку pandas и воспользоваться методом read_csv, передав в качестве параметра путь датасета. Для проверки корректности загруженных данных достаточно воспользоваться методом head для вывода первых пяти записей датасета. Также была импортирована библиотека numpy для последующего использования её функционала и инициализирована переменная RS для последующего её использования в тех конструкторах, где можно задать значение random_state. Реализация вышеописанных шагов представлена в Листинге 1.1.1, результат проверки загрузки датасета – в Листинге 1.1.2.

Листинг 1.1.1 – Реализация загрузки датасета и вывода первых пяти записей

```
import pandas as pd
import numpy as np

RS = 42
X = pd.read_csv('sample_data/lab4_3.csv')
X.head()
```

Листинг 1.1.2 – Первые пять записей датасета

	X1	X2	Class	Extracurricular Activities
0	0.981214	4.666722	LEFT	
1	0.579113	4.325803	LEFT	
2	0.034151	4.326533	LEFT	
3	0.189180	4.525031	LEFT	
4	0.591636	4.827937	LEFT	

Для того, чтобы установить, какие существуют классы, необходимо воспользоваться методом unique из библотеки numpy, передав в данный метод столбец «Class» искомого датафрейма. Результат выполнения данной команды представлен в Листинге 1.1.3.

Листинг 1.1.3 – Уникальные классовые метки

```
array(['LEFT', 'MID', 'RIGHT'], dtype=object)
```

Реализация кодирования каждого класса непосредственно в самом датафрейме представлена в Листинге 1.1.4.

Листинг 1.1.4 – Кодирование категориальных признаков

```
X['Class'].replace(['LEFT', 'MID', 'RIGHT'], [0, 1, 2], inplace=True)
```

1.2. Визуализируйте данные при помощи диаграммы рассеяния с выделением различных классов ней.

Для визуализации данных необходимо импортировать библиотеки matplotlib и seaborn. Реализация отрисовки диаграммы рассеяния представлена в Листинге 1.2.1, результат отрисовки – на Рисунке 1.2.1.

Листинг 1.2.1 – Реализация отрисовки диаграммы рассеяния

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x=X['X1'], y=X['X2'], hue=X['Class'], palette='tab10').set(
    title='Диаграмма рассеяния'
)
```

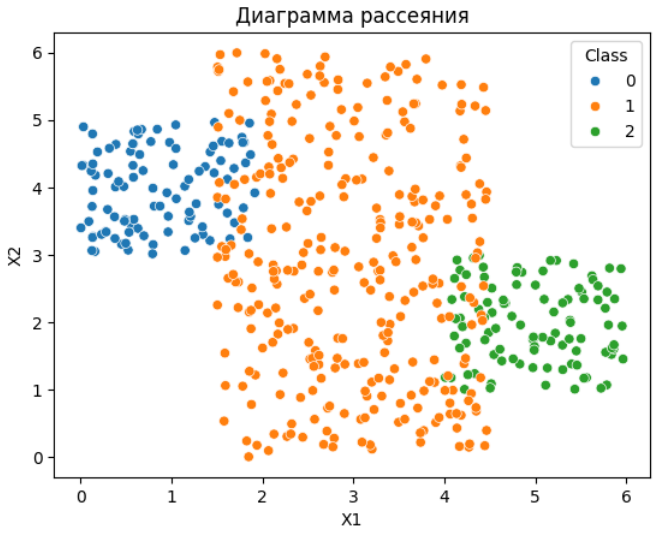

Рисунок 1.2.1 – Диаграмма рассеяния с выделением классов

1.3. Оцените сбалансированность классов.

Для оценки сбалансрованности классов необходимо вывести количество объектов в каждом классе. Реализация представлена в Листинге 1.3.1, результат выполнения данного кода представлен в Листинге 1.3.2.

Листинг 1.3.1 – Реализация оценки мощности каждого класса

```
print(f"Power of #1 class is {len(X[X['Class'] == 0])}")
print(f"Power of #2 class is {len(X[X['Class'] == 1])}")
print(f"Power of #3 class is {len(X[X['Class'] == 2])}")
```

Листинг 1.3.2 – Оценка мощности каждого класса

```
Power of #1 class is 100
Power of #2 class is 300
Power of #3 class is 100
```

Можно сделать вывод, что при оценке параметра accurancy следует использовать сбалансированную оценку, так как классы не сбалансрованы: мощность второго класса втрое больше каждого из оставшихся по отдельности.

1.4. Проведите предобработку данных при необходимости.

Предобработка на данном этапе проводиться не будет, так как эффективнее для каждого метода классификации делать пайплайны, содержащие либо стандартизацию, либо нормализацию, либо отсутствие какой-либо предобработки, а затем внедрять данные пайплайны в кросс-валидацию. Данное решение связано с тем, что в данной работе рассматривается большое количество алгоритмов классификации и вероятность подбора универсальной и оптимальной предобработки уменьшается, следовательно, более правильным решением будет рассмотреть для каждого метода классификации наилучшую предобработку на данном датасете. Данный подход не вызовет проблему с вычислительными ресурсами, так как датасет состоит всего из 500 объектов и имеет всего 2 признака.

Так как практически все алгоримы классификации не работают с датасетами, у которых отсутствуют некоторые значения признаков, следовательно, искомый датасет следует проверить на наличие null-значений. Для этого необходимо при-

менить метод info из библиотеки pandas для данного датасета. Результат вызова данного метода представлен в Листинге 1.4.1.

Листинг 1.4.1 – Проверка датасета на null-значения признаков

```
RangeIndex: 500 entries, 0 to 499
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   X1      500 non-null    float64
 1   X2      500 non-null    float64
 2   Class   500 non-null    int64
dtypes: float64(2), int64(1)
memory usage: 11.8 KB
```

Следовательно, в датасете отсутствуют null-значения признаков.

Также на основании данных на Рисунке 1.2.1, можно сделать вывод, что в датасете отсутствуют выбросы, которые могли бы сильно повлиять на работу таких алгоритмов, как kNN и логистическая регрессия.

1.5. Разделите выборку на обучающую и тестовую. На обучающей проводите обучение модели, на тестовой расчет значений метрик.

Для деления данных на обучающую и тестовую выборки в соотношении 7:3 необходимо воспользоваться функцией train_test_split из sklearn.model_selection. Реализация представдена в Листинге 1.5.1.

Листинг 1.5.1 – Деление данных на тестовую и обучающую выборки

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X.iloc[:, :-1], X['Class'],
                                                    test_size=0.3,
                                                    random_state=RS)
```

2. kNN.

2.1. Проведите классификацию методом k ближайших соседей, подобрав параметры: количество соседей, необходимость взвешивани. Постройте графики зависимости точности (Accuracy) в зависимости от количества соседей (по 1-й линии для взвешенного и не взвешенного метода).

Как упоминалось ранее, для каждого алгоритма классификации реализован пайплайн с подбором предобратки данных, либо её отсутствия. Для kNN при кросс-валидации происходит подбор количества соседей и необходимость во взвешивании. Наилучшая модель выбирается на основании значения сбалансированного значения accurancy. Реализация вышеописанных шагов представлена в Листинге 2.1.1, наилучшие гиперпараметры и предобработка данных– в Листинге 2.1.2.

Листинг 2.1.1 – Подбор предобработки данных и гиперпарамтеров алгоритма kNN

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()
pipe = Pipeline(steps=[("scaler", None), ("knn", knn)])
param_grid = {
    "scaler": [MinMaxScaler(), StandardScaler(), 'passthrough'],
    "knn__n_neighbors": np.arange(1, 2*int(np.sqrt(len(X))), 1),
    "knn__weights": ['distance', 'uniform']
}
search = GridSearchCV(pipe, param_grid, scoring='balanced_accuracy')
search.fit(X_train, y_train)
search.best_params_
```

Листинг 2.1.2 – Наилучшая предобработка данных и гиперпараметры алгоритма kNN

```
{'knn__n_neighbors': 8, 'knn__weights': 'distance', 'scaler': MinMaxScaler()}
```

Можно сделать вывод, что для получения максимального значения balanced_accurancy в алгоритме kNN для данного датасета, необходимо предварительно нормализовать данные, а затем использовать классификатор, для которого установлено число соседей, равное 8, и взвешенное расстояние.

Для отрисовки графика зависимости сбалансированного значения accurancy от числа соседей необходимо для каждого числа соседей обучать классификатор на нормализованных тренеровочных данных, рассчитывать сбалансированное значение accurancy на тестовых данных и записывать в массив для взвешенного и не

взвешенного классификатора. Затем полученные массивы используются непосредственно при отрисовке графиков зависимости. Реализация отрисовки графиков зависимости сбалансированного значения accurancy от числа соседей представлена в Листинге 2.1.3, графики зависимости – на Рисунке 2.1.1.

Листинг 2.1.3 – Реализация отрисовки графиков зависимости сбалансированного значения accurancy от числа соседей для взвешенного и не взвешенного методов

```
from sklearn.metrics import balanced_accuracy_score


X_train_normal = MinMaxScaler().fit_transform(X_train)
X_test_normal = MinMaxScaler().fit_transform(X_test)
b_accur_uniform_list, b_accur_distance_list = [], []
for i in range(1, 40):
    knn_uniform = KNeighborsClassifier(n_neighbors=i, weights='uniform')
    knn_distance = KNeighborsClassifier(n_neighbors=i, weights='distance')
    knn_uniform.fit(X_train_normal, y_train)
    knn_distance.fit(X_train_normal, y_train)
    b_accur_uniform_list.append(balanced_accuracy_score(y_test,
                                        knn_uniform.predict(X_test_normal)))
    b_accur_distance_list.append(balanced_accuracy_score(y_test,
                                        knn_distance.predict(X_test_normal)))
plt.plot(np.arange(1, 40, 1), b_accur_uniform_list, c='blue', label='Не взвешенный метод')
plt.plot(np.arange(1, 40, 1), b_accur_distance_list, c='red', label='Взвешенный метод')
plt.title('Зависимость сбалансированной точности от числа соседей')
plt.xticks(np.arange(1, 40, 2))
plt.grid()
plt.legend()
plt.xlabel('Количество соседей')
plt.xlabel('Значение сбалансированной точности')
plt.show()
```

Рисунок 2.1.1 – Графики зависимости сбалансированного значения accurancy от числа соседей для взвешенного и не взвешенного методов

Исходя из данных на Рисунке 2.1.1, можно сделать вывод, что наибольшее значение используемой метрики достигается при использовании не взвешенного метода и при количестве соседей, равном 6, что не совпадает с данными кросс-валидации, однако кросс-валидацию следует считать более достоверным источником, так как в данном случае она использует 5 тестовых выборок, вместо 1.

2.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

Для выполнения данного задания для каждого метода классификации реализована функция draw_decision_making_boundaries, которая принимает на вход экземпляр классификатора, датасет без классовых меток, обучающая выборка, истинные метки классов и границы отображения по оси OX и OY. Для отрисовки границ на основе обучающей выборки используется функционал DecisionBoundaryDisplay из библиотеки sklearn.inspection. Также отображается легенда и название диаграммы. Реализация данной функции и её вызов для наилучшего kNN классификатора представлена в Листинге 2.2.1, диаграмма рассеяния с границами принятия решения на основе обучающей выборки – на Рисунке 2.2.1.

Листинг 2.2.1 – Реализация функции для отрисовки диаграммы рассеяния и границ принятия решения для kNN

```
from sklearn.inspection import DecisionBoundaryDisplay

def draw_decision_making_boundaries(estimator, X, X_train, y, x_lim, y_lim):
    display = DecisionBoundaryDisplay.from_estimator(
        estimator, X_train, response_method='predict',
        xlabel='X1', ylabel='X2',
        grid_resolution=200, alpha=0.7)
    scatter = display.ax_.scatter(X[:, 0], X[:, 1],
                                  c=y, edgecolors='k', s=15)
    legend = plt.legend(handles=scatter.legend_elements()[0],
                        labels=['0', '1', '2'], title='Class')
    plt.xlim(x_lim)
    plt.ylim(y_lim)
    plt.title('Диаграмма рассеяния с границами принятия решения')
    plt.show()

knn = KNeighborsClassifier(n_neighbors=8, weights='distance')
knn.fit(X_train_normal, y_train)
X_normal = MinMaxScaler().fit_transform(X.iloc[:, :-1])
models_info = {}
models_info['KNN'] = [balanced_accuracy_score(y_test, knn.predict(X_test_normal))]
draw_decision_making_boundaries(knn, X_normal, X_train_normal,
                                X['Class'], (-0.25, 1.25), (-0.25, 1.25))
```

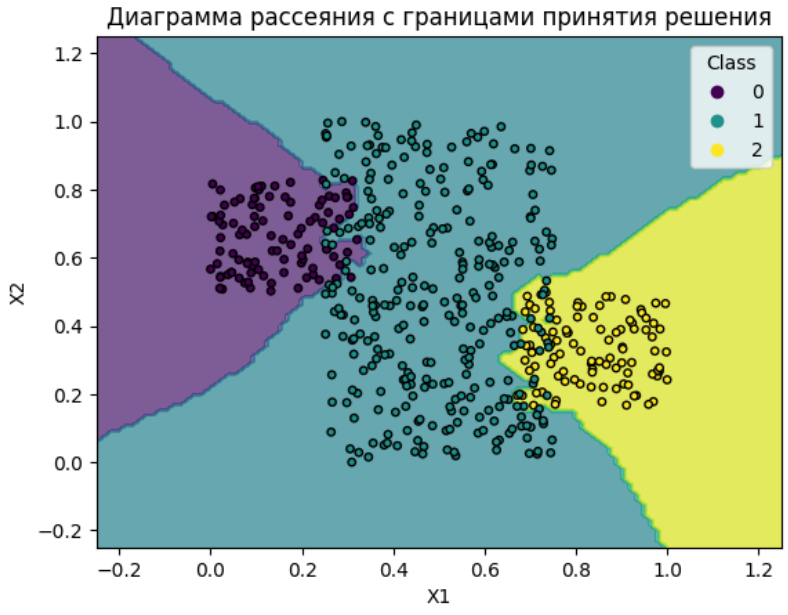

Рисунок 2.2.1 – Диаграмма рассеяния и границы принятия решения для kNN

Исходя из данных на Рисунке 2.2.1 можно сделать вывод о возможном переобучении модели, так как границы принятия решения слишком резкие и извилистые, что влияет на обобщающую способность модели. Вероятнее всего, фиолетвый и жёлтый классы имеют практически прямоугольную форму без глубоких «ям» справа и слева соответственно. Значит, наилучшей моделью будет та, которая игнорирует вышеописанные «ямы», но при этом имеет хорошие метрики.

2.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

Для отрисовки таблицы ошибок использован функционал ConfusionMatrixDisplay из sklearn.metrics, который обёрнут в функцию print_error_matrix, которая принимает на вход экземпляр классификатора, тестовую выборку аргументов и меток классов. Реализация представлена в Листинге 2.3.1, таблица ошибок – на Рисунке 2.3.1.

Листинг 2.3.1 – Реализация функции для отрисовки таблицы ошибок для kNN

```
from sklearn.metrics import ConfusionMatrixDisplay

def print_error_matrix(estimator, X, y):
    ConfusionMatrixDisplay.from_estimator(estimator, X, y, cmap = 'YlGn')
    plt.title('Таблица ошибок')

print_error_matrix(knn, X_test_normal, y_test)
```

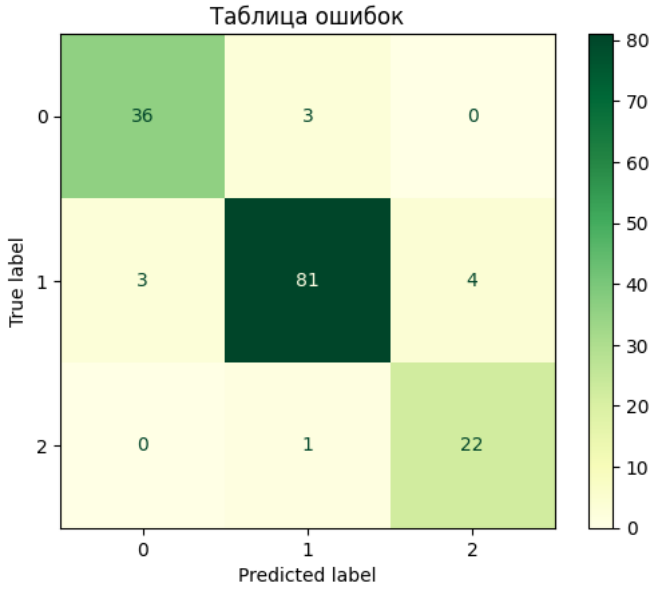

Рисунок 2.3.1 – Таблица ошибок для kNN

Чем зеленее диагональные элементы и белее внедиагональные, тем лучше. Исходя из данных на Рисунке 2.3.1, можно сделать вывод, что качество классификации довольно высокое. Данная модель никогда не классифицирует объекты третьего класса как первого и объекты первого класса как третьего, что совпадает с действительностью: небольшие два класса разделяет один большой.

2.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

Для расчёта данных метрик необходимо воспользоваться функцией precision_recall_fscore_support из sklearn.metrics. Так как данная задача классификации является мультклассовой, следовательно, следует выбрать параметр average. Было выбрано макроусреднение, так как в данной задаче есть один большой класс и два более мелких, каждый из которых втрое меньше большого, а макроусреднению свойственно одинаково учитывать как крупные классы, так и мелкие. Реализация вызова библиотечной функции представлена в Листинге 2.4.1, результат вызова представлен в Листинге 2.4.2.

Листинг 2.4.1 – Реализация вызова библиотечной функции для расчёта Precision, Recall и F1 для kNN

```
from sklearn.metrics import precision_recall_fscore_support

y_pred = knn.predict(X_test_normal)
print(f"(precision, recall, f1score) = {precision_recall_fscore_support(y_test, y_pred,
↪    average='macro')[:-1]}")
```

Листинг 2.4.2 – Рассчитанные значения Precision, Recall и F1 при использовании библиотечной функции для kNN

```
(precision, recall, f1score) = (0.9073906485671192, 0.9333510692206345, 0.9191507639071635)
```

Для расчёта Precision необходимо для каждого диагонального элемента посчитать сумму элементов соответствующего столбца и разделить данный диагональный элемент на рассчитанную сумму, затем просуммировать все такие отношения для каждого диагонального элемента и поделить на 3. Расчёт для Recall аналогичен, но для каждого диагонального элемента сумма считается не по столбцам, а по строкам. Для проверки F1 необходимо для каждого диагонального элемента рассчитать значение F1, затем полученные значения следует просуммировать и поделить на 3. Стоит отметить, что все метрики считались в соответствии с макроусреднением. Релизация проверки на основе таблицы ошибок представлена в Листинге 2.4.3, результат рассчитанных значений – в Листинге 2.4.4.

Листинг 2.4.3 – Реализация расчёта Precision и Recall на основе табличных данных для kNN

```
precision = (36 / (36 + 3) + 81 / (81 + 4) + 22 / (22 + 4)) / 3
recall = (36 / (36 + 3) + 81 / (81 + 7) + 22 / (22 + 1)) / 3
f1_1 = 2 * (36 / (36 + 3)) * (36 / (36 + 3)) / (2 * 36 / (36 + 3))
f1_2 = 2 * (81 / (81 + 4)) * (81 / (81 + 7)) / (81 / (81 + 4) + 81 / (81 + 7))
f1_3 = 2 * (22 / (22 + 4)) * (22 / (22 + 1)) / (22 / (22 + 4) + 22 / (22 + 1))
print(f"precision = {precision}")
print(f"recall = {recall}")
print(f"f1 = {(f1_1 + f1_2 + f1_3)/3}")
models_info = {}
models_info['KNN'] += [precision, recall]
```

Листинг 2.4.4 – Рассчитанные на основе таблицы значения Precision, Recall и Recall для kNN

```
precision = 0.9073906485671192
recall = 0.9333510692206345
f1 = 0.9191507639071635
```

Можно сделать вывод, что табличные данные совпадают с метриками, рассчитанными с помощью библиотечной функции.

2.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

Так как данная задача классификации является многоклассовой, следовательно, строить ROC-кривую нужно для каждого класса по отдельности. При построении будет использована схема OneVsAll. По аналогичной схеме будет производиться расчёт значения AUC. Вышеописанной функционал реализован в виде функции print_roc_curve, которая принимает на вход экземпляр классификатора, тестовую выборку и её метки классов, название классификтора. Реализация представлена в Листинге 2.5.1, ROC-кривые представлены на Рисунке 2.5.1.

Листинг 2.5.1 – Релизация функции отрисовки ROC-кривых в задаче многоклассовой классификации и её вызов для kNN

```
from sklearn.metrics import roc_curve, roc_auc_score

def print_roc_curve(estimator, X_test, y_test, estimator_name):
    fig = plt.Figure(figsize=(5, 5))
    y_score = estimator.predict_proba(X_test)
    colors = ['blue', 'green', 'red', 'orange']
    for i in range(max(y_test)+1):
        fpr, tpr, t = roc_curve(y_test == i, y_score[:, i])
        auc = roc_auc_score(y_test == i, y_score[:, i])
        models_info[estimator_name].append(auc)
        plt.plot(fpr, tpr, c=colors[i], label=f"{i}, AUC = {auc:.3f}")
        plt.xlabel('FPR')
        plt.ylabel('TPR')
        plt.title(f"ROC-кривые для класса каждого класса")
    plt.plot([0, 1], [0, 1], color='black', linestyle='dashed')
    plt.legend(title="Class")
print_roc_curve(knn, X_test_normal, y_test, 'KNN')
```

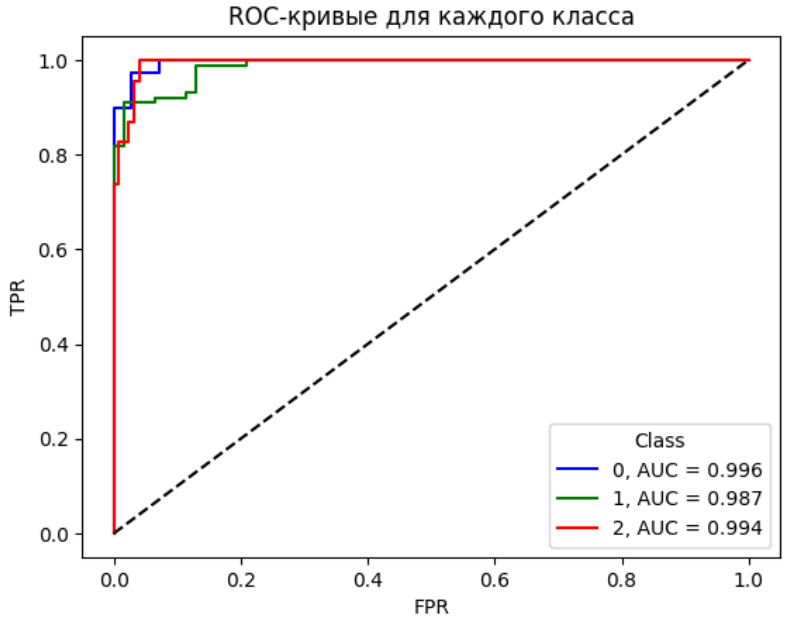

Рисунок 2.5.1 – ROC-кривые в задаче многоклассовой классификации для kNN

Исходя из AUC значений, можно сделать вывод об отличном качестве классификации, так как все значения AUC больше 0.9.

3. Логистическая регрессия.

3.1. Проведите классификацию методом логистической регрессии при различных параметрах: без регуляризации, с l1 регуляризацией, с l2 регуляризацией. Постройте столбчатую диаграмму зависимости точности (Accuracy) от наличия регуляризации. Дальнейшие пункты 3.* выполняйте для лучшего параметра.

Для выполнения данного задания необходимо реализовать функцию, которая будет принимать на вход тип регуляризации, метод подбора коэффициентов регрессии, величину, обратную коэффициенту регуляризации, предобработчики, а также параметр, отвечающий за печать таблицы. Необходимость в реализации данной функции связана с тем, что каждый метод подбора коэффициеннтов регрессии работает с определёнными типами регуляризации. Данная функция содержит в себе пайплайн и кросс-валидацию, на основе которых подбирается наиболее оптимальная комбинация гиперпараметров и предобработчик. Реализация данной функции а также её вызов представлены в Листинге 3.1.1, результат подбора наилучшей модели представлен в Листинге 3.1.2.

Листинг 3.1.1 – Релизация функции для подбора наилучших гиперпараметров модели и её вызов для логистической регрессии

```
from sklearn.linear_model import LogisticRegression

def log_reg_pipeline(penalty, solver, C, scaler, print_table=False):
    log_reg = LogisticRegression(penalty=None, random_state=RS)
    pipe = Pipeline(steps=[("scaler", None), ("log_reg", log_reg)])
    param_grid = {
        "scaler": scaler,
        "log_reg__solver": [solver],
        "log_reg__C": C,
        "log_reg__penalty": penalty
    }
    search = GridSearchCV(pipe, param_grid, scoring='balanced_accuracy')
    search.fit(X_train, y_train)
    if print_table:
        return search.cv_results_
    else:
        return search.best_params_, search.best_score_

penalty_list = [['l2', None], ['l1', 'l2'], ['l2', None],
    ['l2', None],  ['l2', None], ['l1', 'l2', None]]
solver_list =  ['lbfgs', 'liblinear', 'newton-cg',
               'newton-cholesky', 'sag', 'saga']
C = [1e-10]
for i in range(1, 41):
    if i % 2 == 1:
        C.append(C[i-1]*5)
    else:
        C.append(C[i-1]*2)
scalers = [MinMaxScaler(), StandardScaler(), 'passthrough']
log_reg_models_info = np.array([log_reg_pipeline(
    penalty_list[i], solver_list[i], C, scalers) for i in range(len(solver_list))])
print(log_reg_models_info[np.argmax(log_reg_models_info[:, 1])])
```

Листинг 3.1.2 – Наилучшая модель логистической регрессии

```
[{'log_reg__C': 1e-10, 'log_reg__penalty': None, 'log_reg__solver': 'saga', 'scaler':
↪  MinMaxScaler()}
0.8640107760456598]
```

Исходя из полученных результатов, можно сделать, что наилучшее значение сбалансированной точности достигается при предобработке данных с помощью нормализации, при отсутствии регуляризации и при подборе коэффициентов регрессии с помощью вариации градиентного спуска saga.

Для сравнения логистической регрессии с регуляризацией и без неё был создан датафрейм, состоящий из трёх записей: наилучший результат сбалансированной точности при отсутствии регуляризации и при использовании l1 и l2 регуляризаций. Новый датафрейм создавался на основе того, который образуется при преобразовании словаря search.cv_results_в датафрейм при наилучших гиперпараметрах логистической регрессии для каждого типа регуляризации. Реализация формирования данного датафрейма представлена в Листинге 3.1.3, результат – в Листинге 3.1.4.

Листинг 3.1.3 – Реализация формирования таблицы для сравнения сбалансрованной точности при использовании регуляризации и при её отсутствии

```
df = pd.DataFrame(log_reg_pipeline(penalty=['l1', 'l2', None], solver='saga',
                   C=C, scaler=[MinMaxScaler()],
                   print_table=True))
none_df = df[~df['param_log_reg__penalty'].isin(['l1', 'l2'])].sort_values(
    by='mean_test_score', ascending=False).head(1)
l1_df = df[df['param_log_reg__penalty'] == 'l1'].sort_values(
    by='mean_test_score', ascending=False).head(1)
l2_df = df[df['param_log_reg__penalty'] == 'l2'].sort_values(
    by='mean_test_score', ascending=False).head(1)
pd.concat([none_df[['param_log_reg__penalty', 'mean_test_score',
 ↪   'param_log_reg__C']],
        l1_df[['param_log_reg__penalty', 'mean_test_score', 'param_log_reg__C']],
        l2_df[['param_log_reg__penalty', 'mean_test_score', 'param_log_reg__C']]])
```

Листинг 3.1.4 – Сравнение сбалансированной точности при использовании регуляризации и при её отсутствии

	param_log_reg__penalty	mean_test_score	param_log_reg__C
2	None	0.864011	0.0
120	l1	0.864011	10000000000.0
121	l2	0.864011	10000000000.0

Реализация отрисовки зависимости сбалансированной точности от регуляризации представлена в Листинге 3.1.5, результат – на Рисунке 3.1.1.

Листинг 3.1.5 – Реализация отрисовки зависимости сбалансированной точности от регуляризации

```
log_reg_l1 = LogisticRegression(penalty='l1', C=10000000000.0, solver='saga')
    log_reg_l1.fit(X_train_normal, y_train)
    log_reg_l2 = LogisticRegression(penalty='l2', C=10000000000.0, solver='saga')
    log_reg_l2.fit(X_train_normal, y_train)
    log_reg = LogisticRegression(penalty=None, solver='saga')
    log_reg.fit(X_train_normal, y_train)
    b_accur = [balanced_accuracy_score(y_test, log_reg.predict(X_test_normal)),
               balanced_accuracy_score(y_test, log_reg_l1.predict(X_test_normal)),
               balanced_accuracy_score(y_test, log_reg_l2.predict(X_test_normal))]
    names = ['Без регуляризации', 'l1', 'l2']
    plt.bar(names, b_accur)
    for i in range(len(names)):
        plt.text(i, b_accur[i], str(b_accur[i]), ha='center', va='bottom')
    plt.title('Зависимость сбалансированной точности от регуляризации')
    plt.ylabel('Точность')
```

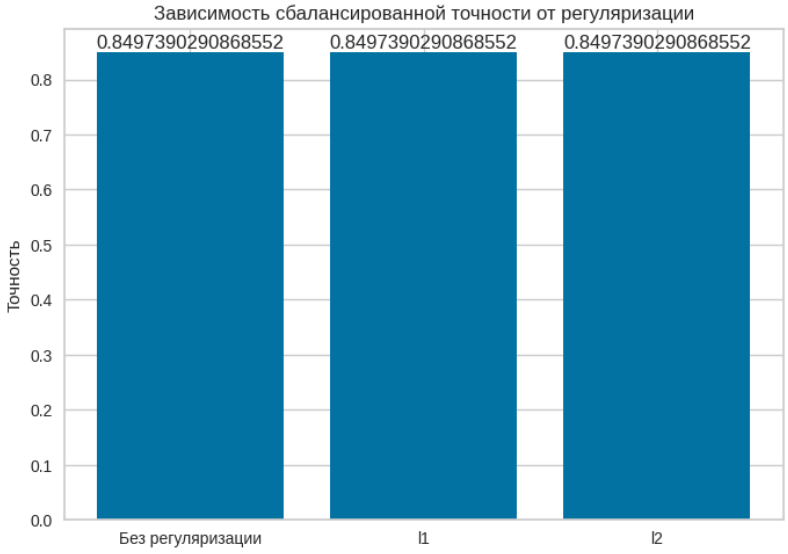

Рисунок 3.1.1 – Зависимость сбалансированной точности от регуляризации

Исходя из полученных данных можно сделать вывод, что модели с наилучшими вариантами регуляризации полностью совпадают с моделью без регуляризации. Следовательно, при использовании логистической регрессии для классификации объектов в данном датасете регуляризация не требуется. Это связано с тем, что обучающая и тестовая выборки качественно похожи.

3.2. Постройте изображение с границами принятия решения отметив на них

точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

Для выполнения данного задания достаточно вызвать функцию draw_decision_making_boundaries, передав ей параметры, аналогичные тем, что передавались в п.2. Реализация вызова представлена в Листинге 3.2.1, результат отрисовки – на Рисунке 3.2.1.

Листинг 3.2.1 – Реализация вызова функции для отрисовки диаграммы рассеяния и границ принятия решения для логистической регрессии

```
models_info['LogisticRegression'] = [balanced_accuracy_score(y_test,
↪   log_reg.predict(X_test_normal))]
draw_decision_making_boundaries(log_reg, X_normal, X_test_normal, X['Class'],
                                (-0.25, 1.25), (-0.25, 1.25))
```

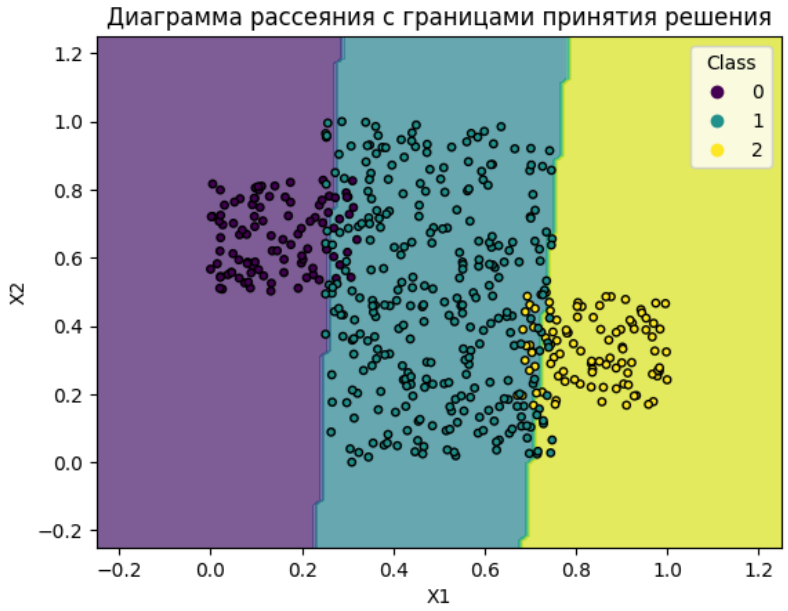

Рисунок 3.2.1 – Диаграмма рассеяния и границы принятия решения для логистической регрессии

Можно сделать вывод, что качество классификации довольно хорошее, но модель не обучилась в полной мере находить объекты фиолетового и жёлтого классов внутри бирюзового.

3.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

Для выполнения данного задания достаточно вызвать функцию print_error_matrix, передав ей параметры, аналогичные тем, что передавались в п.2. Реализация вызова представлена в Листинге 3.3.1, результат отрисовки – на Рисунке 3.3.1.

Листинг 3.3.1 – Реализация вызова функции для отрисовки таблицы ошибок для логистической регрессии

```
print_error_matrix(log_reg, X_test_normal, y_test)
```

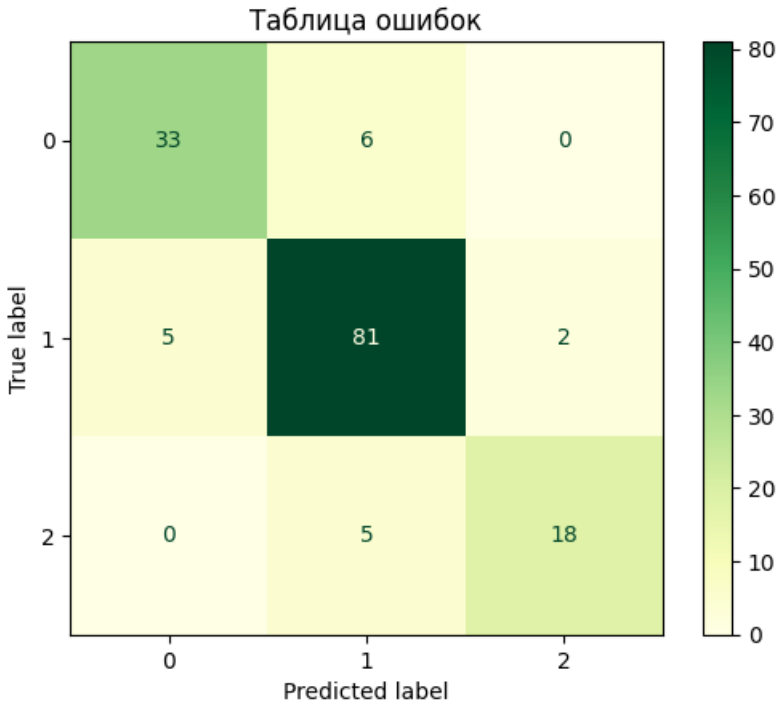

Рисунок 3.3.1 – Таблица ошибок для логистической регрессии

Исходя из матрицы ошибок, можно сделать вывод, что классификация довольно высокого качества. Данная модель никогда не классифицирует объекты третьего класса как первого и объекты первого класса как третьего, что совпадает с действительностью: небольшие два класса разделяет один большой.

3.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

Реализация расчёта метрик представлена в Листинге 3.4.1, результат расчётов – в Листинге 3.4.2.

Листинг 3.4.1 – Реализация вызова библиотечной функции для расчёта Precision, Recall, F1 для логистической регрессии

```
y_pred = log_reg.predict(X_test_normal)
print(f"(precision, recall, f1score) = {precision_recall_fscore_support(y_test, y_pred,
↪   average='macro')[:-1]}")
```

Листинг 3.4.2 – Рассчитанные значения Precision, Recall и F1 при использовании библиотечной функции для логистической регрессии

```
(precision, recall, f1score) = (0.8829519450800914, 0.8497390290868552, 0.8647840531561463)
```

Для расчёта метрик была использована та же логика, что и в п.2. Реализация расчёта метрик представлена в Листинге 3.4.3, результат расчётов – в Листинге 3.4.4.

Листинг 3.4.3 – Реализация расчёта метрик на основе таблицы ошибок для логистической регрессии

```
precision = (33 / (33 + 5) + 81 / (81 + 11) + 18 / (18 + 2)) / 3
recall = (33 / (33 + 6) + 81 / (81 + 7) + 18 / (18 + 5)) / 3
f1_1 = 2 * (33 / (33 + 5)) * (33 / (33 + 6)) / (33 / (33 + 5) + 33 / (33 + 6))
f1_2 = 2 * (81 / (81 + 11)) * (81 / (81 + 7)) / (81 / (81 + 11) + 81 / (81 + 7))
f1_3 = 2 * (18 / (18 + 2)) * (18 / (18 + 5)) / (18 / (18 + 2) + 18 / (18 + 5))
print(f"precision = {precision}")
print(f"recall = {recall}")
print(f"f1 = {(f1_1 + f1_2 + f1_3)/3}")
models_info['LogisticRegression'] += [precision, recall]
```

Листинг 3.4.4 – Результат расчёта метрик на основе таблицы ошибок для логистической регрессии

```
precision = 0.8829519450800914
recall = 0.8497390290868552
f1 = 0.8647840531561463
```

Исходя из полученных расчётов, можно сделать вывод, что данные в матрице ошибок соответствуют результатам, полученным при использовании библиотечной функции.

3.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

Для отрисовки ROC-кривых и расчёта значений AUC была использована реализованная ранее функция print_roc_curve. Реализация вызова данной функции для тестовой выборки преставлена в Листинге 3.5.1, ROC-кривые представлены на Рисунке 3.5.1.

Листинг 3.5.1 – Реализация вызова функции для отрисовки ROC-кривых для логистической регрессии

```
print_roc_curve(log_reg, X_test_normal, y_test, 'LogisticRegression')
```

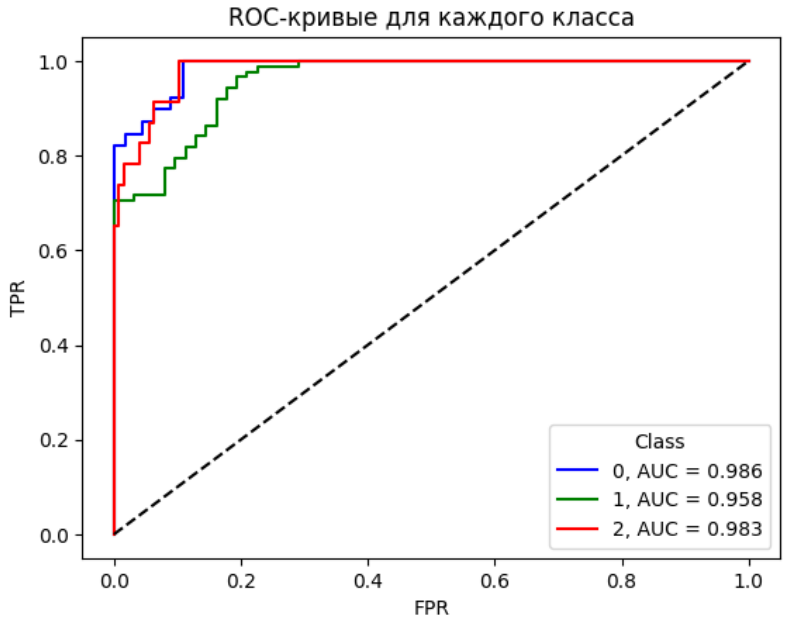

Рисунок 3.5.1 – ROC-кривые в задаче многоклассовой классификации для логистической регрессии

Исходя из AUC значений, можно сделать вывод об отличном качестве классификации, так как все значения AUC больше 0.9.

4. Метод опорных векторов.

4.1. Проведите классификацию методом опорных векторах при различных параметрах ядра: “linear”, “poly” (нужно выбрать степень), “rbf”. Постройте столбчатую диаграмму зависимости точности (Accuracy) от вида ядра. Дальнейшие пункты 4.* выполняйте для лучшего параметра.

По аналогии с предыдущими классификаторами, для данного был реализован пайплайн, который затем отправлялся на кросс-валидацию. Среди параметров данного алгоритма перебиралсся гиперпараметр ядра и его степень, что учитывается только для полиномиального ядра. Реализация подбора наилучшей модели и её предобработки представлены в Листинге 4.1.1, результат – в Листинге 4.1.2.

Листинг 4.1.1 – Реализация подбора наилучшей модели метода опорных векторов и её предобработки

```
from sklearn.svm import SVC

svc = SVC(random_state=RS)
pipe = Pipeline(steps=[("scaler", None), ("svc", svc)])
d = []
for i in range(2, 9):
    d.append(i)
param_grid = {
    "scaler": scalers,
    "svc__kernel": ['linear', 'poly', 'rbf'],
    "svc__degree": d
}
search = GridSearchCV(pipe, param_grid, scoring='balanced_accuracy')
search.fit(X_train, y_train)
print(search.best_params_)
```

Листинг 4.1.2 – Наилучшие гиперпараметры метода опорных векторов и его предобработка

```
{'scaler': 'passthrough', 'svc__degree': 5, 'svc__kernel': 'poly'}
```

Можно сделать вывод, что максимальное значение сбалансированной точности достигается при отсутствии какой-либо предобработки и при использовании полиномиального ядра пятой степени.

Для сравнения наилучших моделей для каждого ядра необходимо воспользоваться словарём search.cv_results_, преобразов его в pandas DataFrame, и сформировать записи, которые соответствуют наилучшим значениям mean_test_score для каждого ядра. Реализация представлена в Листинге 4.1.3, сформированный датафрейм – в Листинге 4.1.4.

Листинг 4.1.3 – Реализация формирования датафрейма с наилучшими моделями для каждого ядра

```
df = pd.DataFrame(search.cv_results_)
poly_df = df[df['param_svc__kernel'] == 'poly'].sort_values(
    by='mean_test_score', ascending=False).head(1)
rbf_df = df[df['param_svc__kernel'] == 'rbf'].sort_values(
    by='mean_test_score', ascending=False).head(1)
linear_df = df[df['param_svc__kernel'] == 'linear'].sort_values(
    by='mean_test_score', ascending=False).head(1)
pd.concat([poly_df[['param_svc__kernel', 'mean_test_score']],
           rbf_df[['param_svc__kernel', 'mean_test_score']],
           linear_df[['param_svc__kernel', 'mean_test_score']]])
```

Листинг 4.1.4 – Сравнение моделей с различными типами ядер

	param_svc__kernel	**mean_test_score**
52	poly	0.912050
2	rbf	0.911964
60	linear	0.849820

Реализация отрисовки столбчатых диаграмм на основе данных из Листинга 4.1.4 и Листинга 4.1.2 представлена в Листинге 4.1.5, результат отрисовки – на Рисунке 4.1.1.

Листинг 4.1.5 – Реализация отрисовки зависимости сбалансированной точности от типа ядра

```
svc_poly = SVC(kernel='poly', degree=5, probability=True)
svc_poly.fit(X_train, y_train)
svc_rbf = SVC(kernel='rbf')
svc_rbf.fit(X_train, y_train)
svc_linear = SVC(kernel='linear', degree=5)
svc_linear.fit(X_train, y_train)
b_accur = [balanced_accuracy_score(y_test, svc_poly.predict(X_test)),
           balanced_accuracy_score(y_test, svc_rbf.predict(X_test)),
           balanced_accuracy_score(y_test, svc_linear.predict(X_test))]
names = ['poly', 'rbf', 'linear']
plt.bar(names, b_accur)
for i in range(len(names)):
    plt.text(i, b_accur[i], str(b_accur[i]), ha='center', va='bottom')
plt.title('Зависимость сбалансированной точности от типа ядра')
plt.ylabel('Точность')
```

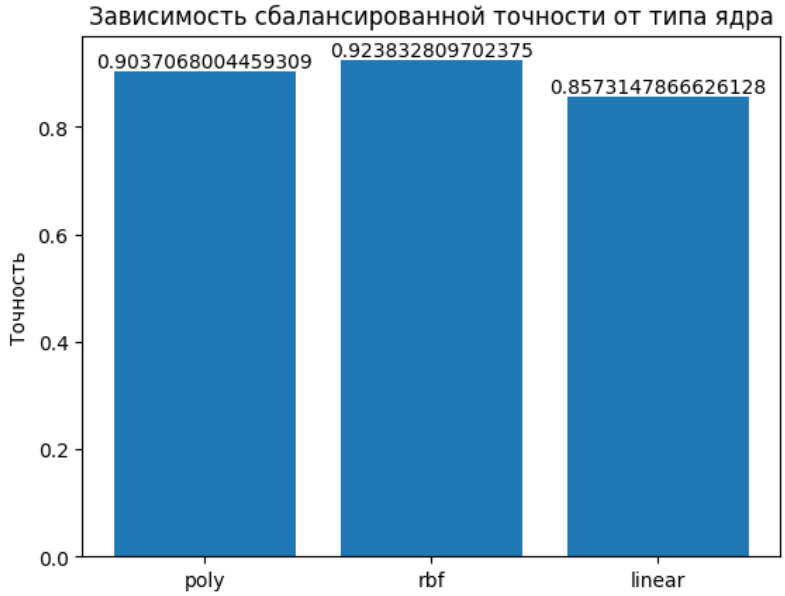

Рисунок 4.1.1 – Зависимость сбалансированной точности от типа ядра

Исходя из полученных данных, можно сделать вывод, наибольшее сбалансированное значение точности на тестовой выборке достигается при использовании радиально-базисного ядра, далее следует полиномиальное ядро пятой степени, а затем – линейное ядро.

Так как при кросс-валидации используется 5 тестовых выборок вместо одной, значит, далее будет использовано полиномиальное ядро пятой степени.

4.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

Для построения изображения с границами принятия решения на основе тренеровочных данных необходимо вызвать ранее реализованную функцию draw_decision_making_boundaries и передать ей аргументы, по смыслу аналогичные пунктам 2 и 3 (см. Листинг 4.2.1). Диаграмма рассеяния с границами принятия решения представлена на Рисунке 4.2.1.

Листинг 4.2.1 – Реализация вызова функции для отрисовки диаграммы рассеяния и границ принятия решения для метода опорных векторов

```
models_info['SVC'] = [balanced_accuracy_score(y_test, svc_poly.predict(X_test))]
draw_decision_making_boundaries(svc_poly, X.iloc[:, :-1].to_numpy(), X_train,
                                X['Class'], (0, 6), (0, 6))
```

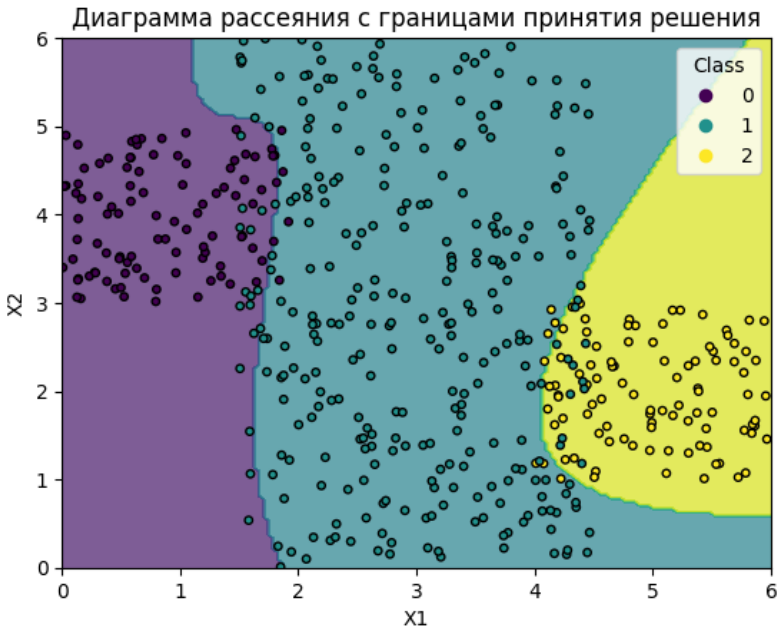

Рисунок 4.2.1 – Диаграмма рассеяния и границы принятия решений для метода опорных векторов

Исходя из данных на Рисунке 4.2.1, можно сделать вывод, что качество классификации очень высокое, так как модель умеет находить объекты небольших классов внутри большого класса, плавно в него углубляясь.

4.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

Реализация вывода матрицы ошибок для тестовой выборки представлена в Листинге 4.3.1, результат вывода – на Рисунке 4.3.1.

Листинг 4.3.1 – Реализация вызова функции для отрисовки таблицы ошибок для меда опорных векторов

```
print_error_matrix(svc_poly, X_test, y_test)
```

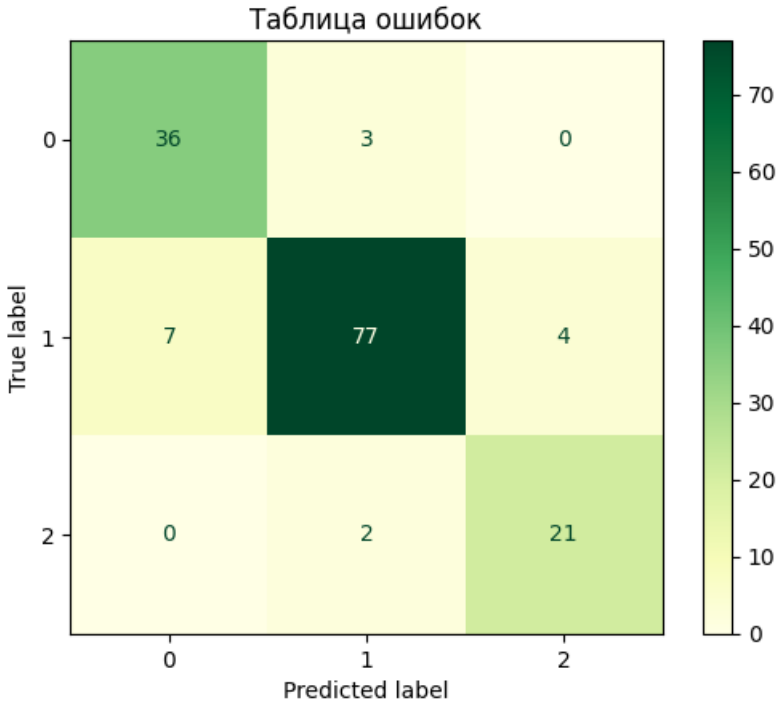

Рисунок 4.3.1 – Матрица ошибок для метода опорных векторов

Исходя из матрицы ошибок, можно сделать вывод, что классификация по-прежнему довольно высокого качества, несмотря на уменьшение чисел на главной диагонали. Данная модель, как и предыдущие, никогда не классифицирует объекты третьего класса как первого и объекты первого класса как третьего, что совпадает с действительностью: небольшие два класса разделяет один большой.

4.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

Реализация расчёта данных метрик для тестовой выборки с помощью библиотечной функции представлена в Листинге 4.4.1, результат расчёта – в Листинге 4.4.2.

Листинг 4.4.1 – Реализация вызова библиотечной функции для расчёта Precision, Recall, F1 для метода опорных векторов

```
y_pred = svc_poly.predict(X_test)
print(f"(precision, recall, f1score) = {precision_recall_fscore_support(y_test, y_pred,
↪   average='macro')[:-1]}")
```

Листинг 4.4.2 – Рассчитанные значения Precision, Recall и F1 при использовании библиотечной функции для метода опорных векторов

```
precision, recall, f1score) = (0.8720778975231612, 0.9037068004459309, 0.8863103778096604)
```

Для расчёта метрик была использована та же логика, что и в п.3. Реализация расчёта метрик представлена в Листинге 4.4.3, результат расчётов – в Листинге 4.4.4.

Листинг 4.4.3 – Реализация расчёта метрик на основе таблицы ошибок для метода опорных векторов

```
precision = (36 / (36 + 7) + 77 / (77 + 5) + 21 / (21 + 4)) / 3
recall = (36 / (36 + 3) + 77 / (77 + 11) + 21 / (21 + 2)) / 3
f1_1 = 2 * (36 / (36 + 7)) * (36 / (36 + 3)) / (36 / (36 + 7) + 36 / (36 + 3))
f1_2 = 2 * (77 / (77 + 5)) * (77 / (77 + 11)) / (77 / (77 + 5) + 77 / (77 + 11))
f1_3 = 2 * (21 / (21 + 4)) * (21 / (21 + 2)) / (21 / (21 + 2) + 21 / (21 + 4))
print(f"precision = {precision}")
print(f"recall = {recall}")
print(f"f1 = {(f1_1 + f1_2 + f1_3) / 3}")
models_info['SVC'] += [precision, recall]
```

Листинг 4.4.4 – Результат расчёта метрик на основе таблицы ошибок для метода опорных векторов

```
precision = 0.8720778975231612
recall = 0.9037068004459309
f1 = 0.8863103778096604
```

Исходя из полученных расчётов, можно сделать вывод, что данные в матрице ошибок соответствуют результатам, полученным при использовании библиотечной функции.

4.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

Для отрисовки ROC-кривых и расчёта AUC значений на основе тестовой выборки достаточно вызвать ранее реализованную функцию print_roc_curve (см. Листинг 4.5.1), ROC-кривые представлены на Рисунке 4.5.1.

Листинг 4.5.1 – Реализация вызова функции для отрисовки ROC-кривых для метода опорных векторов

```
print_roc_curve(svc_poly, X_test, y_test, 'SVC')
```

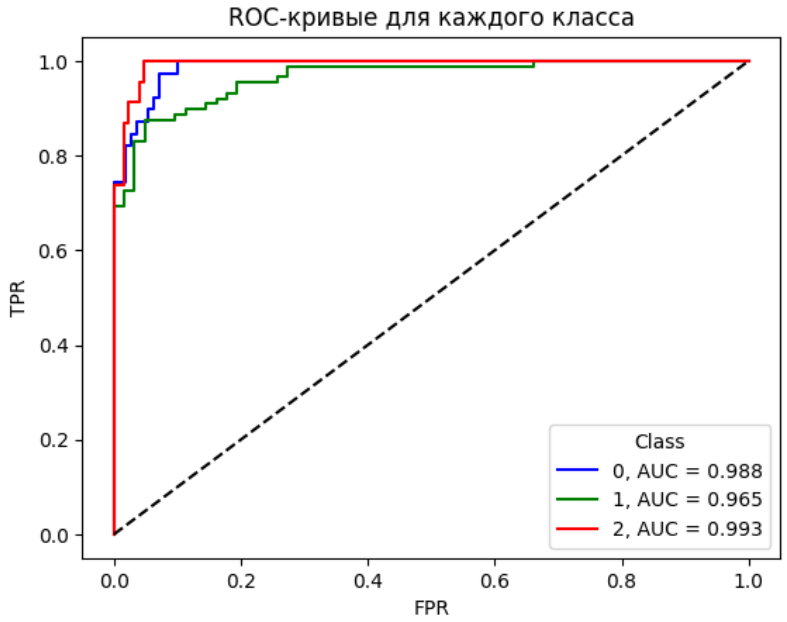

Рисунок 4.5.1 – ROC-кривые в задаче многоклассовой классификации для метода опорных векторов

Исходя из AUC значений, можно сделать вывод об отличном качестве классификации, так как все значения AUC больше 0.9.

5. Решающие деревья.

5.1. Проведите классификацию используя решающие деревья, подобрав параметры при которых получается лучшее обобщение (максимальная глубина/максимальное количество листьев/метрика загрязнения и т.д.).

В целях ускорения работы алгоритма гиперпараметру max_leaf_nodes было присвоено значение 3, так как исходные данные делятся на 3 класса и min_samples_leaf было присвоено значение 50, так как в каждом классе должно быть минимум 50 объектов. Среди гиперпараметров перебирается min_samples_split, от которого зависит глубина дерева: пока во все листах не будет менее min_samples_split

объектов, алгоритм не закончит свою работу. Также происходит перебор гиперпараметра, который отвечает за критерий разбиения в узле. Реализация подбора гиперпараметров представлена в Листинге 5.1.1, результат подбора – в Листинге 5.1.2.

Листинг 5.1.1 – Реализация подбора наилучших гиперпараметров для алгоритма решающих деревьев

```
from sklearn.tree import DecisionTreeClassifier, plot_tree

dtc = DecisionTreeClassifier(random_state=RS, max_leaf_nodes=3,
                             min_samples_leaf=50)
pipe = Pipeline(steps=[("scaler", None), ("dtc", dtc)])
param_grid = {
    "scaler": scalers,
    "dtc__min_samples_split": np.arange(50, 300, dtype=int),
    "dtc__criterion": ['gini', 'entropy', 'log_loss']
}
search = GridSearchCV(pipe, param_grid, scoring='balanced_accuracy')
search.fit(X_train, y_train)
print(search.best_params_)
```

Листинг 5.1.2 – Результат подбора наилучших гиперпараметров для алгоритма решающих деревьев

```
{'dtc__criterion': 'entropy', 'dtc__max_leaf_nodes': 3, 'dtc__min_samples_leaf': 50,
↪  'dtc__min_samples_split': 50, 'scaler': MinMaxScaler()}
```

Исходя из полученных данных, можно сделать вывод, при предварительной нормализации данных, использовании энтропии в узлах, при максимальном числе узлов, равном 3, при минимальном количестве объектов в листьях, равном 50 и при значении min_samples_split, равном 50, достигается максимальное значение сбалансированной точности. Реализация обучения при данных гиперпараметрах представлена в Листинге 5.1.3.

Листинг 5.1.3 – Реализация обучения алгоритма решающих деревьев при наилучших гиперпараметрах

```
dtc = DecisionTreeClassifier(criterion='entropy',
                             min_samples_split=50,
                             min_samples_leaf=50,
                             max_leaf_nodes=3,
                             random_state=RS)
dtc.fit(X_train_normal, y_train)
```

5.2. Постройте изображение с границами принятия решения отметив на них точки классов разным цветом. Сделайте выводы о качестве классификации на основе полученного изображения.

Для построения изображения с границами принятия решения на основе тренеровочных данных необходимо вызвать ранее реализованную функцию draw_decision_making_boundaries и передать ей аргументы, по смыслу аналогичные пунктам 2, 3 и 4 (см. Листинг 5.2.1). Диаграмма рассеяния с границами принятия решения представлена на Рисунке 5.2.1.

Листинг 5.2.1 – Реализация вызова функции для отрисовки диаграммы рассеяния и границ принятия решения для алгоритма решающих деревьев

```
models_info['DecisionTreeClassifier'] = [balanced_accuracy_score(y_test,
↪   dtc.predict(X_test_normal))]
draw_decision_making_boundaries(dtc, X_normal, X['Class'],
                                (-0.25, 1.25), (-0.25, 1.25))
```

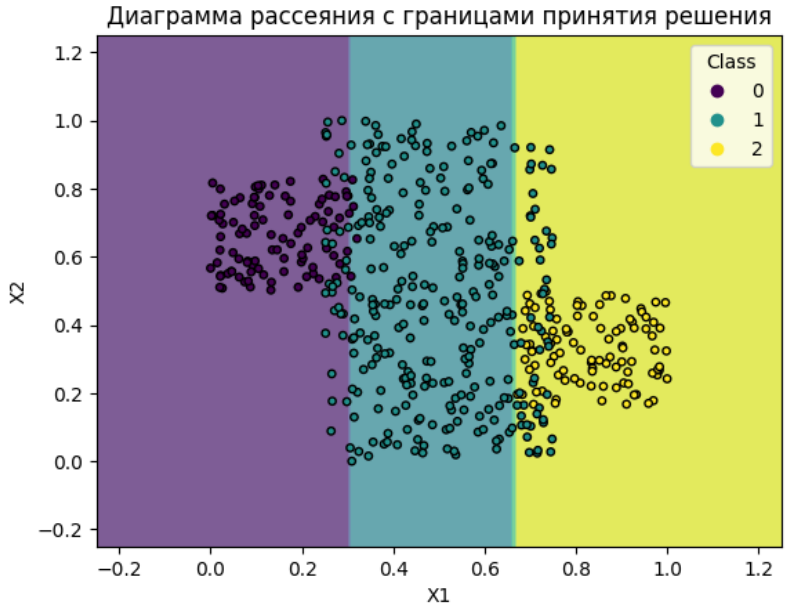

Рисунок 5.2.1 – Диаграмма рассеяния и границы принятия решений для алгоритма решающих деревьев

Исходя из данных на Рисунке 5.2.1, можно сделать вывод, что качество классификации довольно неплохое, но данная модель не умеет находить объекты меньших классов внутри большого класса.

5.3. Постройте таблицу ошибок для полученных результатов. Сделайте выводы о классификации на основе таблицы ошибок.

Для выполнения данного задания достаточно вызвать функцию print_error_matrix, передав ей параметры, аналогичные тем, что передавались в п.2, 3, 4. Реализация вызова представлена в Листинге 5.3.1, результат отрисовки – на Рисунке 3.3.1.

Листинг 5.3.1 – Реализация вызова функции для отрисовки таблицы ошибок

```
print_error_matrix(dtc, X_test_normal, y_test)
```

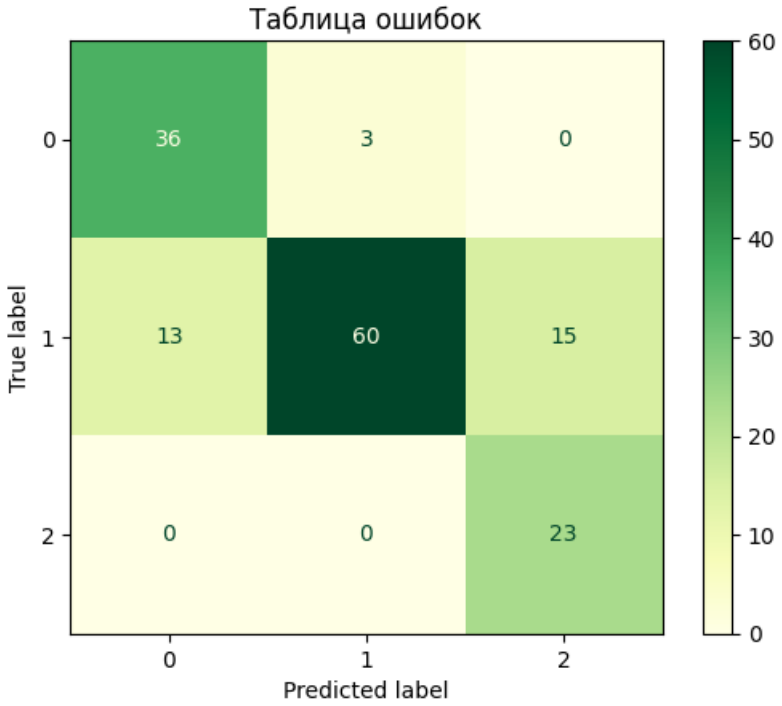

Рисунок 5.3.1 – Таблица ошибок

Исходя из матрицы ошибок, можно сделать вывод, что классификация довольно среднего качества, так как не диагональные элементы во второй строке по сравнению с предыдущими классификаторами стали зеленее, что говорит об ухудшении текущей модели, относительно предыдущих. Данная модель никогда не классифицирует объекты третьего класса как первого и объекты первого класса как третьего, что совпадает с действительностью: небольшие два класса разделяет один большой.

5.4. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

Реализация расчёта данных метрик для тестовой выборки с помощью библиотечной функции представлена в Листинге 5.4.1, результат расчёта – в Листинге 5.4.2.

Листинг 5.4.1 – Реализация вызова библиотечной функции для расчёта Precision, Recall, F1 для метода решающих деревьев

```
y_pred = dtc.predict(X_test_normal)
print(f"(precision, recall, f1score) = {precision_recall_fscore_support(y_test, y_pred,
↪    average='macro')[:-1]}")
```

Листинг 5.4.2 – Рассчитанные значения Precision, Recall и F1 при использовании библиотечной функции для метода решающих деревьев

```
(precision, recall, f1score) = (0.7641126626089032, 0.8682983682983684, 0.7889940551975075)
```

Для расчёта метрик была использована та же логика, что и в п.4. Реализация расчёта метрик представлена в Листинге 5.4.3, результат расчётов – в Листинге 5.4.4.

Листинг 5.4.3 – Реализация расчёта метрик на основе таблицы ошибок для метода решающих деревьев

```
precision = (36 / (36 + 13) + 60 / (60 + 3) + 23 / (23 + 15)) / 3
recall = (36 / (36 + 3) + 60 / (60 + 28) + 23 / (23)) / 3
f1_1 = 2 * (36 / (36 + 13)) * (36 / (36 + 3)) / (36 / (36 + 13) + 36 / (36 + 3))
f1_2 = 2 * (60 / (60 + 3)) * (60 / (60 + 28)) / (60 / (60 + 3) + 60 / (60 + 28))
f1_3 = 2 * (23 / (23 + 15)) * (23 / (23)) / (23 / (23 + 15) + 23 / (23))
print(f"precision = {precision}")
print(f"recall = {recall}")
print(f"f1 = {(f1_1 + f1_2 + f1_3) / 3}")
models_info['DecisionTreeClassifier'] += [precision, recall]
```

Листинг 5.4.4 – Результат расчёта метрик на основе таблицы ошибок для метода решающих деревьев

```
precision = 0.7641126626089032
recall = 0.8682983682983684
f1 = 0.7889940551975075
```

Исходя из полученных расчётов, можно сделать вывод, что данные в матрице ошибок соответствуют результатам, полученным при использовании библиотечной функции.

5.5. Постройте изображение ROC-кривой и рассчитайте значение AUC для полученных результатов. Сделайте выводы о классификации по полученному изображение и значению AUC.

Для отрисовки ROC-кривых и расчёта AUC значений на основе тестовой выборки достаточно вызвать ранее реализованную функцию print_roc_curve (см. Листинг 5.5.1), ROC-кривые представлены на Рисунке 5.5.1.

Листинг 5.5.1 – Реализация вызова функции для отрисовки ROC-кривых для метода решающих деревьев

```
print_roc_curve(svc_poly, X_test, y_test, 'SVC')
```

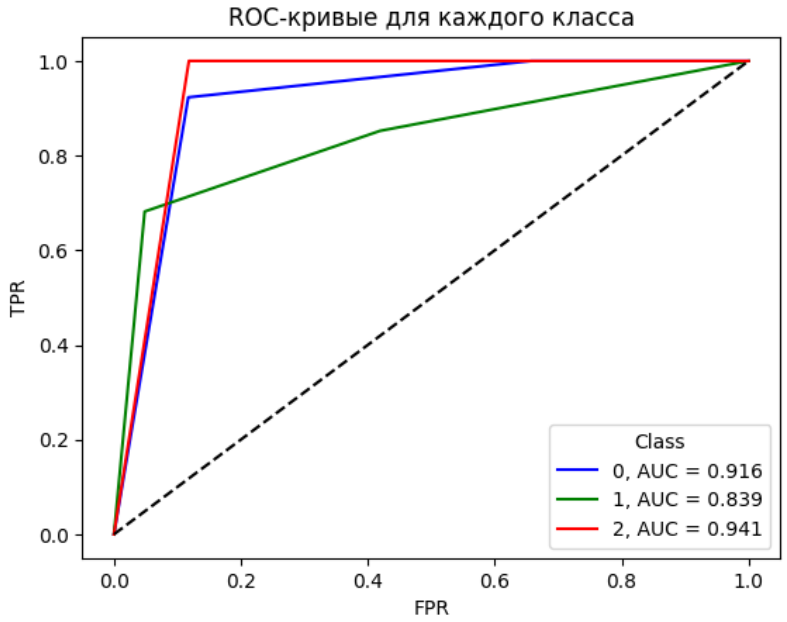

Рисунок 5.5.1 – ROC-кривые в задаче многоклассовой классификации для метода решающих деревьев

Исходя из AUC значений, можно сделать вывод об очень хорошем качестве классификации, так как два значения AUC больше 0.9, но одно из них находится в промежутке от 0.8 до 0.9.

5.6. Визуализируйте полученное дерево решений. Сделайте выводы о правилах в узлах дерева. Сопоставьте их с полученными границами принятия решений.

Для визуализации необходимо воспользоваться библиотечной функцией plot_tree, которая принимает на вход экземпляр классификатора и параметр заполнения узлов. Реализация отрисовки дерева решений представлена в Листинге 5.6.1, дерево решений – на Рисунке 5.6.1.

Листинг 5.6.1 – Реализация отрисовки дерева решений

```
plot_tree(dtc, filled=True)
plt.title('Дерево решений')
```

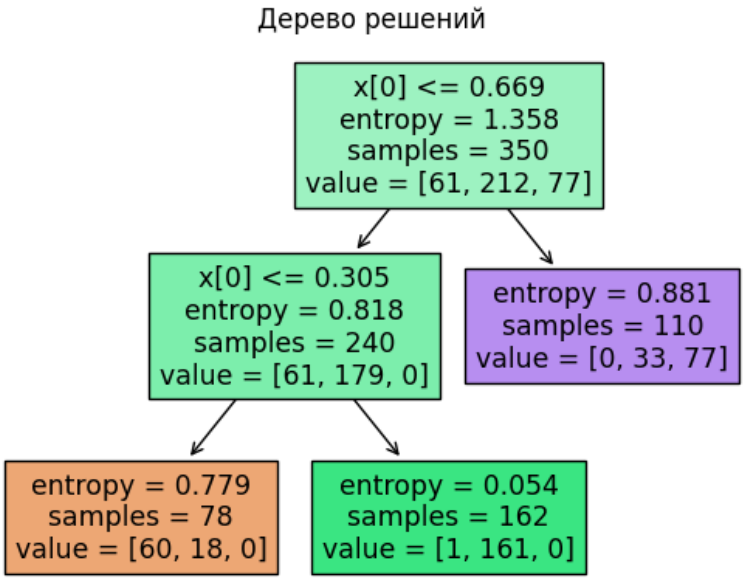

Рисунок 5.6.1 – Дерево решений

Исходя из дерева решений, можно сделать вывод, что оно соответствует границам принятия решений.

6. Выбор классификатора.

6.1. Постройте таблицу с метриками Precision, Recall, AUC для полученных результатов каждым классификатором.

Для построения таблицы для каждого алгоритма классификации рассчитанные метрики записывались в словарь models_info, в котором каждый ключ – название классификатора, а значение – массив, состоящий из сбалансированного значения accurancy, Precision, Recall и AUC для каждого класса. Затем данный словарь передавался конструктору датафреймов для более удобного анализа таблицы. Реализация преобразования словаря в pandas DataFrame представлена в Листинге 6.1.1, таблица – в Листинге 6.1.2.

Листинг 6.1.1 – Реализация отрисовки дерева решений

```
models_info_pd = pd.DataFrame(models_info, index=['b_accurancy', 'precision', 'recall',
↪   'auc #1', 'auc #2', 'auc #3']).T
models_info_pd
```

Листинг 6.1.2 – Таблица для сравнения методов классификации

	b_accurancy	precision	recall	auc #1	auc #2	auc #3
KNN	0.933351	0.907391	0.933351	0.996073	0.986804	0.993838
LogisticRegression	0.849739	0.882952	0.849739	0.986140	0.958211	0.982883
SVC	0.903707	0.872078	0.903707	0.987757	0.965543	0.993153
DecisionTreeClassifier	0.868298	0.764113	0.868298	0.916147	0.838801	0.940945

6.2. Сделайте выводы о том, какие классификаторы лучше всего подходят для вашего набора данных и в каких случаях.

На основе данных в Листинге 6.1.2 и диаграмм рассеяния с границами принятия решений (см. Листинге 2.2.1, Листинге 3.2.1, Листинге 4.2.1, Листинге 5.2.1), можно сделать вывод, что метод опорных векторов с полиномиальным ядром пятой степени является наилучшим классификатором. Несмотря на то, что метод опорных векторов имеет не наилучшие метрики для данного датасета, он лучше любой из рассмотренных моделей классифицирует объекты мелких классов внутри наибольшего: границы принятия решений небольших классов должны плавно входить в наибольший класс и огибать его данные. Алгоритм kNN, например, не удовлетворяет данному условию, так он «слишком хорошо» описывает границы, что ставит данную модель под угрозу переобучения. Остальные алгоритмы классификации довольно грубо описывали небольшие классы. Выбор данной модели является компромиссом между значениями метрик (метод опорных векторов занимает второе место по значениям метрик) и предположением о форме диаграмм рассеяния каждого класса (небольшие классы пересекаются с крупным и имеют формы, близкие к прямоугольным).

Вывод.

В ходе выполнения лабораторной работы изучены такие алгоритмы классификации, как kNN, логистическая регрессия, метод опорных векторов и решающие деревья. Для каждого алгоритма классификации были изучены такие метрики качества классификации, как TP, FP, FN, TN, Precision, Recall, F1 и AUC. Также изучены инструменты для визуализации матриц ошибок, границ принятия решений, деревья решений и ROC-кривых. Были изучены пайплайны и усовершенствованы навыки использования кросс-валидации, а также визуализации данных.

Было установлено, что для данного датасета наилучшим алгоритмом классификации является метод опорных векторов