Изображение выглядит как эмблема, символ, герб, нашивка

Автоматически созданное описание

|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА - Российский технологический университет»**  **РТУ МИРЭА** |

**Институт** Информационных Технологий

**Кафедра** Вычислительной Техники

**ПРАКТИЧЕСКАЯ РАБОТА №4**

**по дисциплине**

**«Проектирование интеллектуальных систем (часть 1/2)»**

Студент группы:ИКБО-04-22 \_\_Кликушин В.И.\_ *(Ф. И.О. студента)*

Преподаватель \_\_ Холмогоров В.В.\_\_

*(Ф.И.О. преподавателя)*

|  |  |  |
| --- | --- | --- |
|  |  |  |

Москва 2025

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc198916539)

[1 ПОСТАНОВКА ЗАДАЧИ 4](#_Toc198916540)

[2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 5](#_Toc198916541)

[2.1 Алгоритм KNN 5](#_Toc198916542)

[2.2 Метрики классификации 6](#_Toc198916543)

[2.3 Дерево решений 8](#_Toc198916544)

[2.4 Случайный лес 9](#_Toc198916545)

[3 ДОКУМЕНТАЦИЯ К ДАННЫМ 11](#_Toc198916546)

[3.1 Описание предметной области 11](#_Toc198916547)

[3.2 Анализ данных 11](#_Toc198916548)

[3.3 Предобработка данных 16](#_Toc198916549)

[4 ПРАКТИЧЕСКАЯ ЧАСТЬ 19](#_Toc198916550)

[4.1 Алгоритм KNN 19](#_Toc198916551)

[4.2 Дерево решений 22](#_Toc198916552)

[4.3 Случайный лес 23](#_Toc198916553)

[ЗАКЛЮЧЕНИЕ 26](#_Toc198916554)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 27](#_Toc198916555)

[ПРИЛОЖЕНИЯ 28](#_Toc198916556)

# ВВЕДЕНИЕ

# 1 ПОСТАНОВКА ЗАДАЧИ

Цель работы: приобрести навыки применения моделей регрессии для решения прикладных задач анализа и сбора данных.

Задачи: определить предметную область решаемой задачи, найти или сгенерировать набор данных для выбранной задачи, проведя предварительную предобработку и подготовку данных, провести визуализацию предобработанного набора данных и выбрать для него модель регрессии, выбрать метрику ошибку и обучить модель регрессии, выполнить задачу предметной области, интерпретировать и визуализировать результаты обучения модели.

# 2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## 2.1 Постановка задачи регрессии

Пусть имеется обучающая выборка из наблюдений (Формула 2.1.1).

, (2.1.1)

где – вектор из признаков для -ого объекта,

– значение целевой переменной для -ого объекта.

Введем «искусственный» признак , чтобы учесть свободный член (Формула 2.1.2).

(2.1.2)

Вектор весов модели обозначим в соответствие с Формулой 2.1.3.

(2.1.3)

Линейная модель предсказания тогда задается Формулой 2.1.4.

(2.1.4)

Задача состоит в том, чтобы подобрать , минимизирующий расхождение между реальными и предсказанными .

## 2.2 Метод наименьших квадратов

Определим ошибку для -ого объекта (Формула 2.2.1).

(2.2.1)

Метод наименьших квадратов минимизирует сумму квадратов этих ошибок (Формула 2.2.2).

(2.2.2)

Формулируется следующая оптимизационная задача (Формула 2.2.3).

(2.2.3)

Построим регрессионную матрицу , в которой в -ой строке лежит вектор (Формула 2.2.4).

(2.2.4)

Вектор истинных ответов задается по Формуле 2.2.5.

(2.2.5)

Все предсказания можно записать в матричном виде (Формула 2.2.6).

(2.2.6)

Вектор ошибок в матричном виде задается по Формуле 2.2.7.

(2.2.7)

Функцию потерь можно переписать (Формула 2.2.8).

(2.2.8)

Вычислим градиент и приравняем его к нулю (Формула 2.2.9).

(2.2.9)

Это и есть нормальные уравнения. Если матрица невырождена (обратима), решение записывается явным образом по Формуле 2.2.10.

(2.2.10)

Вектор содержит оценки свободного члена и коэффициентов , минимизирующих сумму квадратичных ошибок.

## 2.3 Метрики регрессии

В задачах регрессии принято оценивать, насколько точно модель предсказывает числовые значения. Наиболее популярны следующие метрики:

1. **Mean Squared Error (MSE)**

MSE измеряет среднее арифметическое квадратов ошибок (остатков) модели. Он чувствителен к большим отклонениям, потому что разность возводится в квадрат. Если есть крупные выбросы (несоответствия), они сильно увеличивают MSE. Низкое значение MSE означает, что в среднем модель предсказывает близко к истинным (Формула 2.3.1).

, (2.3.1)

где – истинное значение целевой переменной для -ого объекта;

– предсказанное моделью значение для того же -ого объекта;

– число наблюдений в выборке.

1. **Root Mean Squared Error (RMSE)**

# 3 ДОКУМЕНТАЦИЯ К ДАННЫМ

## 3.1 Описание предметной области

В качестве набора данных выбран Student Performance Dataset, синтетический датасет, предназначенный для исследования факторов, влияющих на академическую успеваемость школьников. Набор состоит из 10 000 записей, каждая из которых содержит информацию о совокупности предикторов: количестве часов, потраченных на обучение (Hours Studied), предыдущих оценках (Previous Scores), участии во внеклассных мероприятиях (Extracurricular Activities), среднем количестве часов сна в сутки (Sleep Hours) и числе прорешанных контрольных или тренировочных вариантов (Sample Question Papers Practiced). В качестве целевой переменной выступает Performance Index – показатель общей успеваемости, округлённый до целого значения в диапазоне от 10 до 100 (чем выше индекс, тем лучше академический результат). Цель сбора и использования этих данных — проанализировать взаимосвязь между объёмом самостоятельной работы, уровнем прошлых достижений, участием в активностях вне уроков, режимом сна и практикой решения заданий с итоговым Performance Index. Результаты исследования могут помочь выявить ключевые детерминанты успешности учащихся, спрогнозировать академическую успеваемость и разработать рекомендации для оптимизации учебного процесса.

## 3.2 Анализ данных

В исходных данных представлено 10 000 записей об обучении студентов. Каждый объект соответствует одному учащемуся и содержит пять предикторных переменных и одну целевую переменную:

* Hours Studied (Часы обучения): общее количество часов, которые студент посвятил самостоятельной подготовке;
* Previous Scores (Предыдущие оценки): средний балл, полученный студентом на предыдущих контрольных работах или экзаменах;
* Extracurricular Activities (Внеклассные мероприятия): бинарный индикатор участия в дополнительной активности («Yes»/«No»);
* Sleep Hours (Часы сна): среднее число часов сна в сутки;
* Sample Question Papers Practiced (Прорешанные примерные варианты): количество тренировочных заданий, прорешанных студентом перед экзаменом;
* Performance Index (Индекс успеваемости): итоговый показатель, отражающий общий академический результат студента. Значение индекса округлено до целого и лежит в диапазоне от 10 до 100 (чем выше, тем лучше успеваемость).

Для предобработки и анализа датасета в файле dataset\_manager.py написан класс DatasetManager. Содержание файла dataset\_manager.py представлено в Приложении А.

Описательная статистика признаков отображена на Рисунке 3.2.1.

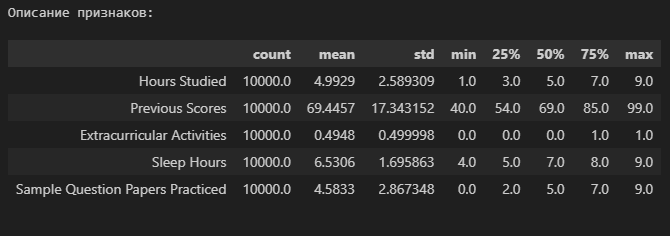


Рисунок 3.2.1 – Описательная статистика признаков

Распределение часов обучения (Hours Studied) демонстрирует относительную симметричность со средним значением 4.99 часа и медианой 5.0 часа, при этом наблюдается умеренная вариативность (стандартное отклонение 2.59) в диапазоне от 1 до 9 часов. Четверть студентов уделяет обучению не более 3 часов, тогда как 75% учатся до 7 часов ежедневно.

Показатель Previous Scores выделяется наиболее широким разбросом данных (стандартное отклонение 17.34) при среднем значении 69.45 баллов. Особого внимания требует нижний квартиль: 25% студентов имеют баллы не выше 54 при абсолютном минимуме 40 баллов, что указывает на возможное наличие выбросов в этой области. Верхние 25% учащихся демонстрируют результаты от 85 баллов.

Бинарный признак Extracurricular Activities показывает сбалансированное распределение: медиана 0.0 при среднем 0.495 свидетельствует, что примерно половина студентов вовлечена во внеклассную активность. Распределение часов сна (Sleep Hours) вызывает определенные опасения, так как четверть студентов спит не более 5 часов при медиане 7 часов и среднем значении 6.53 часа. Разброс значений от 4 до 9 часов с умеренным стандартным отклонением 1.7 указывает на потенциальную проблему дефицита сна у значительной части учащихся.

Признак Sample Question Papers Practiced характеризуется средним значением 4.58 при медиане 5.0 и значительном разбросе от 0 до 9 единиц (стандартное отклонение 2.87). Критичным представляется тот факт, что 25% студентов решают не более 2 пробных тестов, что может негативно влиять на их академические результаты. Полученные данные подчеркивают необходимость масштабирования числовых признаков, особенно Previous Scores, и проверки на выбросы в нижних квартилях.

На Рисунке 3.2.2 показано гистограммное распределение целевой переменной Performance Index.

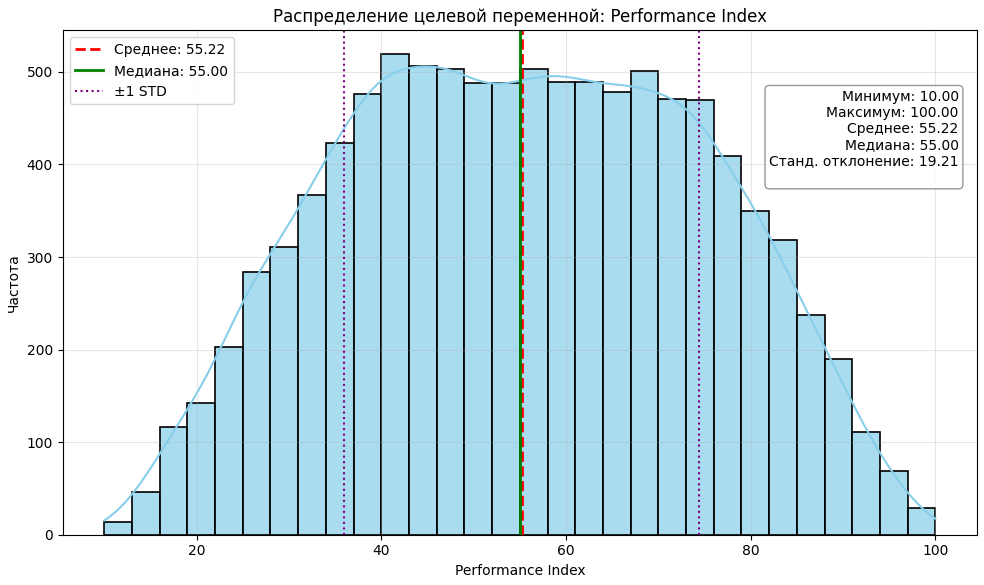


Рисунок 3.2.2 – Гистограмма распределения Performance Index

На Рисунке 3.2.2 видно, что есть небольшая доля экстремально низких и экстремально высоких значений индекса, однако большинство лежит между 60 и 90.

На Рисунке 3.2.3 представлены гистограммы распределений всех пяти признаков.

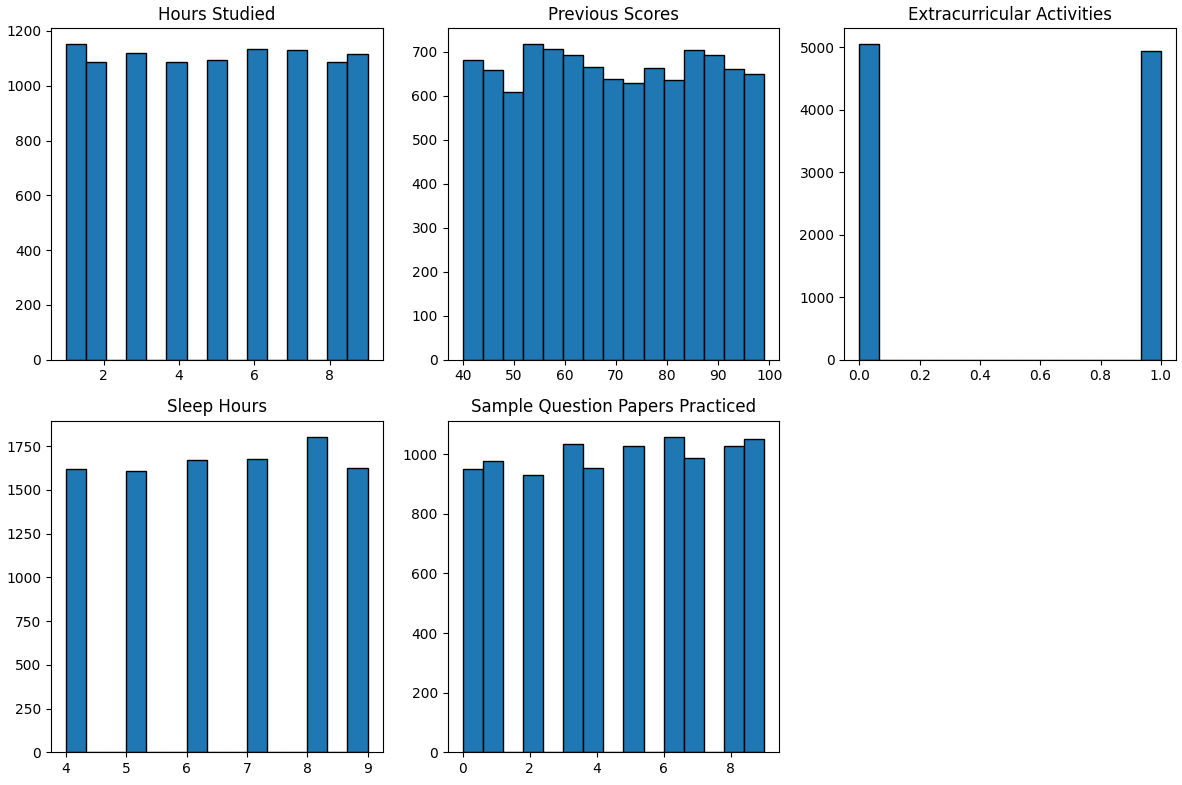


Рисунок 3.2.3 - Гистограммы распределений признаков

Признаки имеют нормальное распределение.

Матрица корреляции признаков представлена на Рисунке 3.2.4.

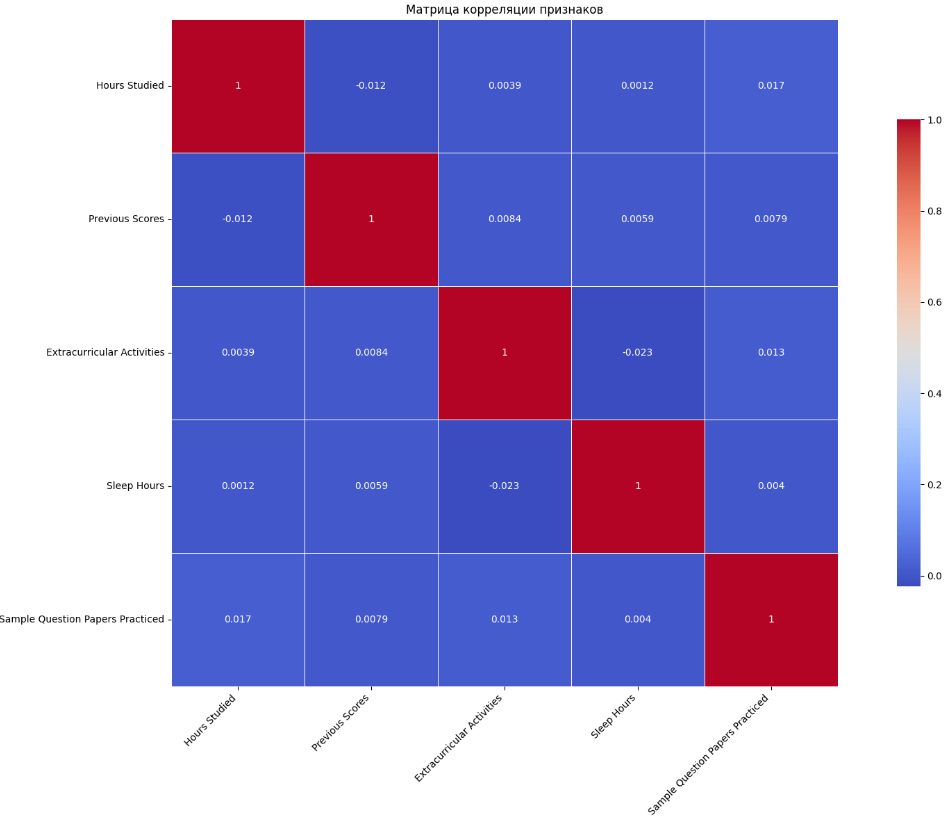


Рисунок 3.2.4 – Матрица корреляции признаков

Матрица указывает на сложную, нелинейную природу образовательных данных, где традиционные факторы успеваемости проявляют неожиданно слабые взаимосвязи. Высоких корреляций между признаками не имеется.

## 3.3 Предобработка данных

Реализованы следующие этапы предобработки данных:

* удаление дубликатов строк;
* удаление выбросов по Z-оценке;
* масштабирование признаков (StandardScaler);
* выбор и удаление избыточных признаков.

Удаление дубликатов строк подразумевает обнаружение и удаление полностью идентичных записей (строк) в исходном датасете. Под «идентичностью» понимается совпадение всех значений по всем признакам. В рассматриваемом датасете удалено 127 дубликатов.

Выбросы (аномальные значения) — это отдельные объекты, сильно отклоняющиеся от общей «массы» точек. Чаще всего они встречаются в признаках с широким диапазоном. Один из способов формального выявления выбросов — использовать Z-оценку.

Для каждого значения признака в образце рассчитывается величина по Формуле 3.3.1.

, (3.3.1)

где – среднее отклонение признака по всем образцам;

– стандартное отклонение признака по всем образцам.

Образец считается выбросом, если хотя бы один признак имеет .

В качестве порогового значения выбран = 3. Таким образом, удалены все образцы, в которых хотя бы один признак отклонён от среднего более чем на три стандартных отклонения. В результате удаления выбросов по Z-оценке было исключено десять строк из исходного набора данных.

Скалирование признаков — это приведение всех измеряемых величин в единый единичный масштаб. Без масштабирования признаки с большим диапазоном «будут весить» значительно больше при классификации, чем признаки с узким диапазоном.

StandardScaler — один из наиболее распространённых способов стандартизации. Для каждого признака вычисляется среднее и стандартное отклонение . Каждое значение преобразуется по Формуле 3.3.2.

, (3.3.2)

В результате стандартизированный признак ​ имеет среднее 0 и стандартное отклонение 1.

Данные разделены на обучающую и тестовую выборки в соотношении 80:20. Обучающая выборка содержит 7898 образца, а тестовая выборка – 1975 образцов.

# 4 ПРАКТИЧЕСКАЯ ЧАСТЬ

# ЗАКЛЮЧЕНИЕ

# СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Сорокин, А. Б. Безусловная оптимизация. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин, О. В. Платонова, Л. М. Железняк — М. РТУ МИРЭА , 2020.
2. Сорокин, А. Б. Введение в генетические алгоритмы: теория, расчеты и приложения. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин — М. МИРЭА , 2018.
3. Student Performance (Multiple Linear Regression) [Электронный ресурс]: Kaggle. URL: https://www.kaggle.com/datasets/nikhil7280/student-performance-multiple-linear-regression/data (Дата обращения: 19.05.2025).
4. Основы линейной регрессии [Электронный ресурс]: Habr. URL: https://habr.com/ru/articles/514818/ (Дата обращения: 20.05.2025).

# ПРИЛОЖЕНИЯ

Приложение А — Файл dataset\_manager.py для предобработки и анализа датасета.

Приложение Б — Файл KNN.py с использованием готовой реализации алгоритма KNN.

Приложение В — Файл KNN\_custom.py с собственной реализацией алгоритма KNN.

Приложение Г — Файл DecisionTree.py с использованием готовой реализации алгоритма Decision Tree.

Приложение Д — Файл RandomForest.py с использованием готовой реализации модели Random Forest.

### Приложение А

Файл dataset\_manager.py для предобработки и анализа датасета

Листинг А – Содержание файла dataset\_manager.py

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import StandardScaler

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

from typing import Optional, Dict, Tuple

from pandas import DataFrame, Series

class DatasetManager:

    """

    Менеджер датасета для задач регрессии: загрузка из CSV, анализ, предобработка и визуализация.

    Параметры:

        csv\_path (str): Путь к CSV-файлу с данными.

        target\_column (str): Название столбца с целевой переменной.

    Атрибуты:

        df (DataFrame): Полный исходный DataFrame.

        features (DataFrame): DataFrame с только признаками (без целевой переменной).

        target (Series): Серия с целевой переменной.

        scaled\_features (DataFrame): DataFrame с масштабированными признаками.

        stats (Dict[str, DataFrame]): Словарь рассчитанных статистик.

    """

    def \_\_init\_\_(self, csv\_path: str, target\_column: str) -> None:

        self.csv\_path: str = csv\_path

        self.target\_column: str = target\_column

        self.df: Optional[DataFrame] = None

        self.features: Optional[DataFrame] = None

        self.target: Optional[Series] = None

        self.scaled\_features: Optional[DataFrame] = None

        self.stats: Dict[str, DataFrame] = {}

        self.\_load\_data()

        self.\_dataset\_custom\_preprocess()

        self.\_extract\_features\_target()

        if self.csv\_path == 'housing\_dataset.csv':

            self.remove\_feature('Location')

            self.remove\_feature('City')

    def \_dataset\_custom\_preprocess(self):

        if self.csv\_path == 'Student\_Performance.csv':

            self.df["Extracurricular Activities"] = self.df["Extracurricular Activities"].map({"Yes":1 , "No":0})

        else:

            cols\_to\_check = [col for col in self.df.columns if col != "No. of Bedrooms"]

            mask\_no\_nines = (self.df[cols\_to\_check] != 9).all(axis=1)

            self.df = self.df[mask\_no\_nines].reset\_index(drop=True)

    def \_load\_data(self) -> None:

        """

        Загружает данные из CSV-файла в self.df.

Продолжение Листинга А

        Выбрасывает:

            FileNotFoundError: если файл по пути csv\_path не найден.

            pd.errors.EmptyDataError: если CSV пустой или неверный формат.

        """

        try:

            self.df = pd.read\_csv(self.csv\_path)

        except FileNotFoundError as e:

            raise FileNotFoundError(f"Файл не найден по пути: {self.csv\_path}") from e

        except pd.errors.EmptyDataError as e:

            raise pd.errors.EmptyDataError(

                f"Пустой или некорректный CSV: {self.csv\_path}"

            ) from e

        print(

            f"Данные загружены: {self.df.shape[0]} строк, {self.df.shape[1]} столбцов"

        )

    def \_extract\_features\_target(self) -> None:

        """

        Разделяет DataFrame на признаки и целевую переменную.

        После выполнения:

            - self.features содержит все столбцы, кроме target\_column.

            - self.target содержит Series с данными целевой переменной.

        Выбрасывает:

            ValueError: если target\_column отсутствует в self.df.

        """

        if self.df is None:

            raise RuntimeError("Данные не загружены. Сначала вызовите \_load\_data().")

        if self.target\_column not in self.df.columns:

            raise ValueError(

                f"Целевая переменная '{self.target\_column}' не найдена в данных."

            )

        self.target = self.df[self.target\_column].copy()

        self.features = self.df.drop(columns=[self.target\_column]).copy()

    def compute\_basic\_statistics(self) -> Dict[str, DataFrame]:

        """

        Вычисляет базовые статистики по признакам и сохраняет их в self.stats.

        Сохраняются:

            - "describe": описательные статистики (mean, std, min, max, квартили) для каждого признака.

            - "correlation\_matrix": матрица корреляций между признаками.

            - "target\_distribution": описательные статистики целевой переменной.

        Возвращает:

            Dict[str, DataFrame]: Словарь со статистиками.

        Выбрасывает:

            RuntimeError: если признаки или целевая переменная не выделены.

        """

        if self.features is None or self.target is None:

            raise RuntimeError(

                "Признаки или целевая переменная не выделены. Вызовите \_extract\_features\_target()."

            )

Продолжение Листинга А

        desc\_features = self.features.describe().T

        self.stats["describe"] = desc\_features

        corr\_features = self.features.corr()

        self.stats["correlation\_matrix"] = corr\_features

        desc\_target = self.target.describe().to\_frame(name="target\_stats")

        self.stats["target\_distribution"] = desc\_target

        return self.stats

    def preprocess(

        self,

        drop\_duplicates: bool = True,

        drop\_outliers: bool = True,

        z\_thresh: float = 3.0,

    ) -> None:

        """

        Предобработка данных:

            1. Удаление дубликатов (если drop\_duplicates=True).

            2. Удаление выбросов по Z-оценке (если drop\_outliers=True).

            3. Масштабирование признаков StandardScaler.

        Параметры:

            drop\_duplicates (bool): Удалять ли полные дубликаты строк.

            drop\_outliers (bool): Удалять ли выбросы по Z-оценке для каждого признака.

            z\_thresh (float): Порог Z-оценки; строки, у которых |z\_score| > z\_thresh хотя бы по одному признаку, удаляются.

        После выполнения:

            - self.df обновляется без дубликатов и выбросов.

            - self.features и self.target обновляются согласно очищенному DataFrame.

            - self.scaled\_features заполняется DataFrame с масштабированными признаками.

        Выбрасывает:

            RuntimeError: если self.df не инициализирован.

        """

        if self.df is None:

            raise RuntimeError("Данные не загружены. Сначала вызовите \_load\_data().")

        df\_proc = self.df.copy()

        if drop\_duplicates:

            before\_dupes = df\_proc.shape[0]

            df\_proc = df\_proc.drop\_duplicates().reset\_index(drop=True)

            after\_dupes = df\_proc.shape[0]

            print(f"Удалено дубликатов: {before\_dupes - after\_dupes}")

        if drop\_outliers:

            df\_no\_target = df\_proc.drop(columns=[self.target\_column], errors="ignore")

            means = df\_no\_target.mean()

            stds = df\_no\_target.std(ddof=0)

            z\_scores = (df\_no\_target - means) / stds

            mask = (z\_scores.abs() <= z\_thresh).all(axis=1)

            before\_out = df\_proc.shape[0]

            df\_proc = df\_proc[mask].reset\_index(drop=True)

Продолжение Листинга А

            after\_out = df\_proc.shape[0]

            print(f"Удалено выбросов: {before\_out - after\_out}")

        self.df = df\_proc

        self.target = df\_proc[self.target\_column].copy()

        self.features = df\_proc.drop(columns=[self.target\_column]).copy()

        scaler = StandardScaler()

        scaled\_array = scaler.fit\_transform(self.features)

        self.scaled\_features = pd.DataFrame(

            scaled\_array, columns=self.features.columns, index=self.features.index

        )

        print(

            "Предобработка завершена: дубликаты и выбросы удалены (если указано), признаки масштабированы."

        )

    def visualize\_distributions(self, figsize: Tuple[int, int] = (12, 8)) -> None:

        """

        Строит гистограммы распределений каждого признака (до масштабирования).

        Параметры:

            figsize (Tuple[int, int]): Размер фигуры (ширина, высота) в дюймах.

        """

        if self.features is None:

            raise RuntimeError(

                "Признаки не выделены. Сначала вызовите \_extract\_features\_target()."

            )

        n = len(self.features.columns)

        cols = 3

        rows = (n + cols - 1) // cols

        fig, axes = plt.subplots(rows, cols, figsize=figsize)

        axes = axes.flatten()

        for i, col in enumerate(self.features.columns):

            axes[i].hist(self.features[col], bins=15, edgecolor="black")

            axes[i].set\_title(col)

        for j in range(n, len(axes)):

            axes[j].axis("off")

        plt.tight\_layout()

        plt.show()

    def visualize\_correlation\_heatmap(

        self, figsize: Tuple[int, int] = (18, 12)

    ) -> None:

        """

        Строит тепловую карту корреляций между признаками с помощью seaborn.

        Параметры:

            figsize (Tuple[int, int]): Размер фигуры (ширина, высота) в дюймах.

        Выбрасывает:

            RuntimeError: если self.features не инициализирован.

        """

        if self.features is None:

            raise RuntimeError(

Продолжение Листинга А

                "Признаки не выделены. Вызовите \_extract\_features\_target()."

            )

        plt.figure(figsize=figsize)

        sns.heatmap(

            self.features.corr(),

            annot=True,

            cmap="coolwarm",

            linewidths=0.5,

            square=True,

            cbar\_kws={"shrink": 0.7},

        )

        plt.title("Матрица корреляции признаков")

        plt.xticks(rotation=45, ha="right")

        plt.yticks(rotation=0)

        plt.tight\_layout()

        plt.show()

    def split\_data(self, test\_size: float = 0.2, random\_state: int = 42) -> None:

        """

        Разделяет данные на обучающую и тестовую выборки.

        Параметры:

            test\_size (float): Доля тестовой выборки (по умолчанию 0.2).

            random\_state (int): Seed для воспроизводимости (по умолчанию 42).

        Выбрасывает:

            RuntimeError: если self.scaled\_features или self.target не инициализированы.

        """

        if self.scaled\_features is None or self.target is None:

            raise RuntimeError("Данные не предобработаны. Вызовите preprocess().")

        self.X\_train, self.X\_test, self.y\_train, self.y\_test = train\_test\_split(

            self.scaled\_features,

            self.target,

            test\_size=test\_size,

            random\_state=random\_state,

        )

        print(

            f"Данные разделены:\n"

            f"- Обучающая выборка: {self.X\_train.shape[0]} образцов\n"

            f"- Тестовая выборка: {self.X\_test.shape[0]} образцов"

        )

    def get\_training\_data(self) -> Tuple[DataFrame, Series]:

        """

        Возвращает обучающие данные (признаки и целевую переменную).

        Возвращает:

            Tuple[DataFrame, Series]: (X\_train, y\_train).

        Выбрасывает:

            RuntimeError: если split\_data не был вызван.

        """

        if not hasattr(self, "X\_train") or not hasattr(self, "y\_train"):

            raise RuntimeError("Данные не разделены. Вызовите split\_data().")

        return self.X\_train, self.y\_train

    def get\_testing\_data(self) -> Tuple[DataFrame, Series]:

Продолжение Листинга А

        """

        Возвращает тестовые данные (признаки и целевую переменную).

        Возвращает:

            Tuple[DataFrame, Series]: (X\_test, y\_test).

        Выбрасывает:

            RuntimeError: если split\_data не был вызван.

        """

        if not hasattr(self, "X\_test") or not hasattr(self, "y\_test"):

            raise RuntimeError("Данные не разделены. Вызовите split\_data().")

        return self.X\_test, self.y\_test

    def remove\_feature(self, feature\_name: str) -> None:

        """

        Удаляет указанный признак из текущего набора данных.

        Параметры:

            feature\_name (str): Название удаляемого признака.

        Исключения:

            ValueError: если feature\_name не строка.

            KeyError: если признак отсутствует в self.features.

            RuntimeError: если self.features не инициализирован.

        """

        if self.features is None:

            raise RuntimeError(

                "Набор признаков пуст. Вызовите \_extract\_features\_target()."

            )

        if not isinstance(feature\_name, str):

            raise ValueError(

                f"Имя признака должно быть строкой, получено {type(feature\_name).\_\_name\_\_}"

            )

        if feature\_name not in self.features.columns:

            raise KeyError(

                f"Признак '{feature\_name}' отсутствует в текущем наборе признаков."

            )

        self.features.drop(columns=[feature\_name], inplace=True)

        if self.df is not None and feature\_name in self.df.columns:

            self.df.drop(columns=[feature\_name], inplace=True)

        if (

            self.scaled\_features is not None

            and feature\_name in self.scaled\_features.columns

        ):

            self.scaled\_features.drop(columns=[feature\_name], inplace=True)

        print(f"Признак '{feature\_name}' успешно удалён из набора данных.")

    def visualize\_target\_distribution(

        self,

        figsize: Tuple[int, int] = (10, 6),

        bins: int = 30,

        kde: bool = True,

        log\_scale: bool = False

    ) -> None:

        """

        Строит гистограмму распределения целевой переменной (цены дома).

        Параметры:

            figsize (Tuple[int, int]): Размер графика (ширина, высота).

Продолжение Листинга А

            bins (int): Количество бинов для гистограммы.

            kde (bool): Отображать ли кривую оценки плотности.

            log\_scale (bool): Использовать логарифмическую шкалу для оси Y.

        Выбрасывает:

            RuntimeError: Если целевая переменная не загружена.

        """

        if self.target is None:

            raise RuntimeError("Целевая переменная не загружена. Сначала вызовите \_extract\_features\_target().")

        plt.figure(figsize=figsize)

        sns.histplot(

            self.target,

            bins=bins,

            kde=kde,

            color='skyblue',

            edgecolor='black',

            linewidth=1.2,

            alpha=0.7

        )

        mean\_val = self.target.mean()

        median\_val = self.target.median()

        std\_val = self.target.std()

        plt.axvline(mean\_val, color='red', linestyle='--', linewidth=2, label=f'Среднее: {mean\_val:.2f}')

        plt.axvline(median\_val, color='green', linestyle='-', linewidth=2, label=f'Медиана: {median\_val:.2f}')

        plt.axvline(mean\_val - std\_val, color='purple', linestyle=':', linewidth=1.5, label=f'±1 STD')

        plt.axvline(mean\_val + std\_val, color='purple', linestyle=':', linewidth=1.5)

        if log\_scale:

            plt.yscale('log')

            plt.ylabel('Частота (log scale)')

        else:

            plt.ylabel('Частота')

        plt.title(f'Распределение целевой переменной: {self.target\_column}')

        plt.xlabel(self.target\_column)

        plt.grid(True, alpha=0.3)

        plt.legend()

        stats\_text = (

            f"Минимум: {self.target.min():.2f}\n"

            f"Максимум: {self.target.max():.2f}\n"

            f"Среднее: {mean\_val:.2f}\n"

            f"Медиана: {median\_val:.2f}\n"

            f"Станд. отклонение: {std\_val:.2f}\n"

        )

        plt.annotate(

            stats\_text,

            xy=(0.98, 0.7),

            xycoords='axes fraction',

            bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alpha=0.8),

            horizontalalignment='right',

Окончание Листинга А

            fontsize=10

        )

        plt.tight\_layout()

        plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    # manager = DatasetManager(csv\_path="Student\_Performance.csv", target\_column="Performance Index")

    # stats = manager.compute\_basic\_statistics()

    # print("Описание признаков:")

    # print(stats["describe"])

    # print("\nКорреляционная матрица признаков:")

    # print(stats["correlation\_matrix"])

    # print("\nСтатистика по целевой переменной:")

    # print(stats["target\_distribution"])

    # manager.visualize\_distributions()

    # manager.visualize\_correlation\_heatmap()

    # manager.visualize\_target\_distribution()

    # manager.preprocess(drop\_duplicates=True, drop\_outliers=True, z\_thresh=2.0)

    # manager.split\_data(test\_size=0.2, random\_state=42)

    # X\_train, y\_train = manager.get\_training\_data()

    # X\_test, y\_test = manager.get\_testing\_data()

    test\_manager = DatasetManager('housing\_dataset.csv', 'Price')

    test\_manager.preprocess(z\_thresh=5)

### Приложение Б

Файл KNN.py с использованием готовой реализации алгоритма KNN

Листинг Б – Файл KNN.py

import numpy as np

import pandas as pd

from typing import Union, Optional, Dict

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import (

    accuracy\_score,

    precision\_score,

    recall\_score,

    f1\_score,

    confusion\_matrix

)

from dataset\_manager import DatasetManager

class KNNClassifier:

    def \_\_init\_\_(

        self,

        n\_neighbors: int = 5,

        weights: str = 'uniform',

        metric: str = 'minkowski'

    ) -> None:

        """

        Классификатор k-ближайших соседей (KNN).

        Параметры:

            n\_neighbors (int): Количество соседей (по умолчанию 5).

            weights (str): Стратегия взвешивания:

                - 'uniform': все соседи имеют равный вес

                - 'distance': вес обратно пропорционален расстоянию (по умолчанию 'uniform').

            metric (str): Метрика для расчета расстояний (по умолчанию 'minkowski').

        """

        self.n\_neighbors = n\_neighbors

        self.weights = weights

        self.metric = metric

        self.model: Optional[KNeighborsClassifier] = None

        self.classes\_: Optional[np.ndarray] = None

    def fit(

        self,

        X\_train: Union[pd.DataFrame, np.ndarray],

        y\_train: Union[pd.Series, np.ndarray]

    ) -> None:

        """

        Обучение модели на обучающих данных.

        Параметры:

            X\_train (DataFrame/ndarray): Матрица признаков обучающей выборки.

            y\_train (Series/ndarray): Вектор целевых меток.

        """

        self.model = KNeighborsClassifier(

            n\_neighbors=self.n\_neighbors,

            weights=self.weights,

            metric=self.metric

        )

        self.model.fit(X\_train, y\_train)

Продолжение Листинга Б

        self.classes\_ = self.model.classes\_

    def predict(

        self,

        X\_test: Union[pd.DataFrame, np.ndarray]

    ) -> np.ndarray:

        """

        Предсказание классов для новых данных.

        Параметры:

            X\_test (DataFrame/ndarray): Матрица признаков тестовой выборки.

        Возвращает:

            ndarray: Массив предсказанных меток.

        Исключения:

            RuntimeError: Если модель не обучена.

        """

        if self.model is None:

            raise RuntimeError("Сначала выполните обучение модели (fit()).")

        return self.model.predict(X\_test)

    def calculate\_accuracy(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray

    ) -> float:

        """

        Вычисление точности (Accuracy).

        Параметры:

            y\_true (Series/ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            float: Значение метрики Accuracy ∈ [0, 1].

        """

        return accuracy\_score(y\_true, y\_pred)

    def calculate\_precision(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray,

        average: str = 'macro'

    ) -> float:

        """

        Вычисление точности (Precision).

        Параметры:

            y\_true (Series/ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Стратегия усреднения:

                - 'macro': среднее по классам

                - 'micro': глобальное усреднение

                - 'weighted': взвешенное среднее

        Возвращает:

            float: Значение метрики Precision.

        """

        return precision\_score(y\_true, y\_pred, average=average, zero\_division=0)

Продолжение Листинга Б

    def calculate\_recall(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray,

        average: str = 'macro'

    ) -> float:

        """

        Вычисление полноты (Recall).

        Параметры:

            y\_true (Series или ndarray): Истинные метки классов.

            y\_pred (ndarray): Предсказанные моделью метки классов.

            average (str): Способ усреднения:

                - 'macro': среднее значение recall по всем классам;

                - 'micro': глобальная метрика по всем объектам;

                - 'weighted': среднее, взвешенное по количеству объектов в каждом классе.

        Возвращает:

            float: Значение метрики Recall в диапазоне [0, 1].

        """

        return recall\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_f1(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray,

        average: str = 'macro'

    ) -> float:

        """

        Вычисление F1-меры (F1-score).

        Параметры:

            y\_true (Series или ndarray): Истинные метки классов.

            y\_pred (ndarray): Предсказанные моделью метки классов.

            average (str): Способ усреднения:

                - 'macro': F1-score по каждому классу, затем среднее;

                - 'micro': общее число TP, FP и FN;

                - 'weighted': среднее, взвешенное по количеству объектов каждого класса.

        Возвращает:

            float: Значение метрики F1-score в диапазоне [0, 1].

        """

        return f1\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_confusion\_matrix(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray

    ) -> np.ndarray:

        """

        Построение матрицы ошибок (confusion matrix), показывающей распределение предсказаний модели по классам.

        Параметры:

            y\_true (Series или ndarray): Истинные метки классов.

            y\_pred (ndarray): Предсказанные моделью метки классов.

        Возвращает:

            ndarray: Квадратная матрица размера [n\_classes x n\_classes], где

Окончание Листинга Б

                    строки соответствуют истинным меткам, а столбцы — предсказанным.

        """

        return confusion\_matrix(y\_true, y\_pred)

    def get\_metrics\_report(

        self,

        y\_true: Union[pd.Series, np.ndarray],

        y\_pred: np.ndarray,

        average: str = 'macro'

    ) -> Dict[str, float]:

        """

        Генерация сводного отчёта по основным метрикам классификации.

        Параметры:

            y\_true (Series или ndarray): Истинные метки классов.

            y\_pred (ndarray): Предсказанные моделью метки классов.

            average (str): Способ усреднения для precision, recall и F1:

                - 'macro': по всем классам одинаково,

                - 'micro': глобально по всем примерам,

                - 'weighted': с учётом долей классов в выборке.

        Возвращает:

            Dict[str, float]: Словарь, содержащий значения следующих метрик:

                - 'accuracy'

                - 'precision'

                - 'recall'

                - 'f1'

        """

        return {

            "accuracy": self.calculate\_accuracy(y\_true, y\_pred),

            "precision": self.calculate\_precision(y\_true, y\_pred, average),

            "recall": self.calculate\_recall(y\_true, y\_pred, average),

            "f1": self.calculate\_f1(y\_true, y\_pred, average)

        }

if \_\_name\_\_ == "\_\_main\_\_":

    manager = DatasetManager(source="sklearn")

    manager.preprocess()

    manager.remove\_feature("total\_phenols")

    manager.split\_data(test\_size=0.2, stratify=True)

    X\_train, y\_train = manager.get\_training\_data()

    X\_test, y\_test = manager.get\_testing\_data()

    knn = KNNClassifier(n\_neighbors=3, weights='distance')

    knn.fit(X\_train, y\_train)

    y\_pred = knn.predict(X\_test)

    report = knn.get\_metrics\_report(y\_test, y\_pred)

    print("\nОтчет о метриках классификации:")

    for metric, value in report.items():

        print(f"- {metric}: {value:.4f}")

    print("\nМатрица ошибок:")

    print(knn.calculate\_confusion\_matrix(y\_test, y\_pred))

### Приложение В

Файл KNN\_custom.py с собственной реализацией алгоритма KNN

Листинг В – Файл KNN\_custom.py

import numpy as np

import pandas as pd

from typing import Union, Optional, Dict

from collections import Counter

from dataset\_manager import DatasetManager

class KNNCustom:

    def \_\_init\_\_(

        self, n\_neighbors: int = 5, weights: str = "uniform", metric: str = "euclidean"

    ) -> None:

        """

        Кастомная реализация классификатора k-ближайших соседей.

        Параметры:

            n\_neighbors (int): Количество соседей (по умолчанию 5).

            weights (str): Стратегия взвешивания:

                - 'uniform': равные веса

                - 'distance': обратно пропорционально расстоянию

            metric (str): Метрика расстояния ('euclidean', 'manhattan', 'cosine')

        """

        self.n\_neighbors = n\_neighbors

        self.weights = weights

        self.metric = metric

        self.X\_train: Optional[np.ndarray] = None

        self.y\_train: Optional[np.ndarray] = None

        self.classes\_: Optional[np.ndarray] = None

    def fit(

        self,

        X\_train: Union[pd.DataFrame, np.ndarray],

        y\_train: Union[pd.Series, np.ndarray],

    ) -> None:

        """

        Сохраняет обучающую выборку в памяти модели.

        Параметры:

            X\_train (DataFrame или ndarray): Матрица признаков обучающих объектов.

            y\_train (Series или ndarray): Вектор меток классов.

        """

        self.X\_train = np.array(X\_train)

        self.y\_train = np.array(y\_train)

        self.classes\_ = np.unique(y\_train)

    def \_calculate\_distance(self, a: np.ndarray, b: np.ndarray) -> float:

        """

        Вычисляет расстояние между двумя точками в заданной метрике.

        Параметры:

            a (ndarray): Первая точка.

            b (ndarray): Вторая точка.

        Возвращает:

            float: Расстояние между a и b в соответствии с выбранной метрикой ('euclidean', 'manhattan', 'cosine').

Продолжение Листинга В

        Исключения:

            ValueError: Если указана неподдерживаемая метрика.

        """

        if self.metric == "euclidean":

            return np.sqrt(np.sum((a - b) \*\* 2))

        elif self.metric == "manhattan":

            return np.sum(np.abs(a - b))

        elif self.metric == "cosine":

            return 1 - np.dot(a, b) / (np.linalg.norm(a) \* np.linalg.norm(b))

        else:

            raise ValueError(f"Unknown metric: {self.metric}")

    def predict(self, X\_test: Union[pd.DataFrame, np.ndarray]) -> np.ndarray:

        """

        Предсказывает метки классов для объектов тестовой выборки.

        Параметры:

            X\_test (DataFrame или ndarray): Матрица признаков тестовых объектов.

        Возвращает:

            ndarray: Вектор предсказанных меток.

        Исключения:

            RuntimeError: Если модель не была обучена методом fit().

        """

        if self.X\_train is None or self.y\_train is None:

            raise RuntimeError("Модель не обучена. Вызовите fit() перед predict().")

        X\_test = np.array(X\_test)

        predictions = []

        for x in X\_test:

            distances = [

                self.\_calculate\_distance(x, x\_train) for x\_train in self.X\_train

            ]

            k\_indices = np.argsort(distances)[: self.n\_neighbors]

            k\_labels = self.y\_train[k\_indices]

            k\_distances = np.array(distances)[k\_indices]

            if self.weights == "distance":

                weights = 1 / (k\_distances + 1e-8)

            else:

                weights = np.ones\_like(k\_distances)

            counter = Counter()

            for label, weight in zip(k\_labels, weights):

                counter[label] += weight

            predictions.append(max(counter, key=counter.get))

        return np.array(predictions)

    def calculate\_accuracy(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

        """

        Вычисляет точность (accuracy) классификации.

        Параметры:

Продолжение Листинга В

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            float: Доля правильных ответов от общего числа наблюдений.

        """

        return np.sum(y\_true == y\_pred) / len(y\_true)

    def calculate\_confusion\_matrix(

        self, y\_true: np.ndarray, y\_pred: np.ndarray

    ) -> np.ndarray:

        """

        Построение матрицы ошибок (confusion matrix).

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            ndarray: Матрица размера [n\_classes x n\_classes], где строки — истинные классы,

                    столбцы — предсказанные классы.

        """

        n\_classes = len(self.classes\_)

        matrix = np.zeros((n\_classes, n\_classes), dtype=int)

        for true, pred in zip(y\_true, y\_pred):

            matrix[true, pred] += 1

        return matrix

    def \_calculate\_class\_metrics(

        self, y\_true: np.ndarray, y\_pred: np.ndarray

    ) -> Dict[int, Dict[str, float]]:

        """

        Вычисляет метрики precision, recall и F1 для каждого класса отдельно.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            Dict[int, Dict[str, float]]: Словарь, в котором для каждого класса содержатся:

                - precision

                - recall

                - f1

                - support (количество наблюдений данного класса)

        """

        matrix = self.calculate\_confusion\_matrix(y\_true, y\_pred)

        metrics = {}

        for i, class\_label in enumerate(self.classes\_):

            tp = matrix[i, i]

            fp = np.sum(matrix[:, i]) - tp

            fn = np.sum(matrix[i, :]) - tp

            tn = np.sum(matrix) - tp - fp - fn

            precision = tp / (tp + fp) if (tp + fp) > 0 else 0

            recall = tp / (tp + fn) if (tp + fn) > 0 else 0

            f1 = (

Продолжение Листинга В

                2 \* (precision \* recall) / (precision + recall)

                if (precision + recall) > 0

                else 0

            )

            metrics[class\_label] = {

                "precision": precision,

                "recall": recall,

                "f1": f1,

                "support": tp + fn,

            }

        return metrics

    def calculate\_precision(

        self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro"

    ) -> float:

        """

        Вычисляет среднюю точность (precision) по классам.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Метод усреднения:

                - 'macro': равное среднее по всем классам,

                - 'weighted': среднее с учетом поддержки (support),

                - 'micro': глобальная точность по всем классам.

        Возвращает:

            float: Значение метрики precision.

        """

        metrics = self.\_calculate\_class\_metrics(y\_true, y\_pred)

        precisions = [m["precision"] for m in metrics.values()]

        supports = [m["support"] for m in metrics.values()]

        if average == "macro":

            return np.mean(precisions)

        elif average == "weighted":

            return np.average(precisions, weights=supports)

        elif average == "micro":

            matrix = self.calculate\_confusion\_matrix(y\_true, y\_pred)

            tp = np.sum(np.diag(matrix))

            fp = np.sum(matrix, axis=0) - np.diag(matrix)

            return tp / (tp + np.sum(fp))

        else:

            raise ValueError("Неподдерживаемый тип усреднения")

    def calculate\_recall(

        self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro"

    ) -> float:

        """

        Вычисляет среднюю полноту (recall) по классам.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Метод усреднения:

                - 'macro': равное среднее по всем классам,

                - 'weighted': среднее с учетом поддержки (support),

                - 'micro': глобальная полнота по всем классам.

Продолжение Листинга В

        Возвращает:

            float: Значение метрики recall.

        """

        metrics = self.\_calculate\_class\_metrics(y\_true, y\_pred)

        recalls = [m["recall"] for m in metrics.values()]

        supports = [m["support"] for m in metrics.values()]

        if average == "macro":

            return np.mean(recalls)

        elif average == "weighted":

            return np.average(recalls, weights=supports)

        elif average == "micro":

            matrix = self.calculate\_confusion\_matrix(y\_true, y\_pred)

            tp = np.sum(np.diag(matrix))

            fn = np.sum(matrix, axis=1) - np.diag(matrix)

            return tp / (tp + np.sum(fn))

        else:

            raise ValueError("Неподдерживаемый тип усреднения")

    def calculate\_f1(

        self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro"

    ) -> float:

        """

        Вычисляет среднее значение F1-меры по классам.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Метод усреднения ('macro', 'weighted', 'micro').

        Возвращает:

            float: Значение F1-меры.

        """

        precision = self.calculate\_precision(y\_true, y\_pred, average)

        recall = self.calculate\_recall(y\_true, y\_pred, average)

        return (

            2 \* (precision \* recall) / (precision + recall)

            if (precision + recall) > 0

            else 0

        )

    def get\_metrics\_report(

        self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro"

    ) -> Dict[str, float]:

        """

        Генерирует сводный отчёт по метрикам классификации.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Тип усреднения для precision, recall и F1 ('macro', 'weighted', 'micro').

        Возвращает:

            Dict[str, float]: Словарь с метриками:

                - 'accuracy'

                - 'precision'

                - 'recall'

                - 'f1'

        """

        return {

Окончание Листинга В

            "accuracy": self.calculate\_accuracy(y\_true, y\_pred),

            "precision": self.calculate\_precision(y\_true, y\_pred, average),

            "recall": self.calculate\_recall(y\_true, y\_pred, average),

            "f1": self.calculate\_f1(y\_true, y\_pred, average),

        }

if \_\_name\_\_ == "\_\_main\_\_":

    manager = DatasetManager(source="sklearn")

    manager.preprocess()

    manager.remove\_feature("total\_phenols")

    manager.split\_data(test\_size=0.2, stratify=True)

    X\_train, y\_train = manager.get\_training\_data()

    X\_test, y\_test = manager.get\_testing\_data()

    knn = KNNCustom(n\_neighbors=3, weights="distance", metric="euclidean")

    knn.fit(X\_train.values, y\_train.values)

    y\_pred = knn.predict(X\_test.values)

    report = knn.get\_metrics\_report(y\_test.values, y\_pred)

    print("\nОтчет о метриках классификации:")

    for metric, value in report.items():

        print(f"- {metric}: {value:.4f}")

    print("\nМатрица ошибок:")

    print(knn.calculate\_confusion\_matrix(y\_test.values, y\_pred))

### Приложение Г

Файл DecisionTree.py с использованием готовой реализации алгоритма Decision Tree

Листинг Г – Файл DecisionTree.py

import numpy as np

import pandas as pd

from typing import Union, Optional, Dict

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

import matplotlib.pyplot as plt

from dataset\_manager import DatasetManager

class DecisionTreeModel:

    def \_\_init\_\_(self, criterion: str = "gini", max\_depth: Optional[int] = None) -> None:

        """

        Инициализирует классификатор на основе дерева решений.

        Параметры:

            criterion (str): Критерий для оценки качества разбиения:

                - 'gini': индекс Джини;

                - 'entropy': информация по Шеннону.

            max\_depth (Optional[int]): Максимально допустимая глубина дерева.

                Если None, дерево строится до исчерпания выборки.

        """

        self.criterion = criterion

        self.max\_depth = max\_depth

        self.model: Optional[DecisionTreeClassifier] = None

        self.classes\_: Optional[np.ndarray] = None

    def fit(self, X\_train: Union[pd.DataFrame, np.ndarray], y\_train: Union[pd.Series, np.ndarray]) -> None:

        """

        Обучает модель дерева решений по обучающим данным.

        Параметры:

            X\_train (DataFrame | ndarray): Матрица признаков обучающей выборки.

            y\_train (Series | ndarray): Вектор истинных меток классов.

        """

        self.model = DecisionTreeClassifier(criterion=self.criterion, max\_depth=self.max\_depth, random\_state=42)

        self.model.fit(X\_train, y\_train)

        self.classes\_ = self.model.classes\_

    def predict(self, X\_test: Union[pd.DataFrame, np.ndarray]) -> np.ndarray:

        """

        Предсказывает метки классов для новых объектов.

        Параметры:

            X\_test (DataFrame | ndarray): Матрица признаков тестовой выборки.

        Возвращает:

            ndarray: Предсказанные метки классов.

        Исключения:

            RuntimeError: если модель не обучена.

        """

Продолжение Листинга Г

        if self.model is None:

            raise RuntimeError("Сначала обучите модель с помощью fit().")

        return self.model.predict(X\_test)

    def plot(self, feature\_names: Optional[list] = None, class\_names: Optional[list] = None) -> None:

        """

        Визуализирует структуру дерева решений.

        Параметры:

            feature\_names (list): Названия признаков (столбцов).

            class\_names (list): Названия классов (если есть).

        """

        if self.model is None:

            raise RuntimeError("Сначала обучите модель.")

        plt.figure(figsize=(16, 10))

        plot\_tree(self.model, filled=True, feature\_names=feature\_names, class\_names=class\_names)

        plt.title("Визуализация дерева решений")

        plt.show()

    def calculate\_accuracy(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

        """

        Вычисляет метрику Accuracy (долю правильных классификаций).

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            float: Значение Accuracy в диапазоне [0, 1].

        """

        return accuracy\_score(y\_true, y\_pred)

    def calculate\_precision(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет метрику Precision (точность предсказания классов).

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Тип усреднения:

                - 'macro', 'micro', 'weighted'.

        Возвращает:

            float: Значение Precision.

        """

        return precision\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_recall(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет метрику Recall (полноту) — насколько хорошо модель находит положительные примеры.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Способ усреднения ('macro', 'micro', 'weighted').

Продолжение Листинга Г

        Возвращает:

            float: Значение Recall.

        """

        return recall\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_f1(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет F1-меру — гармоническое среднее между точностью и полнотой.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Тип усреднения ('macro', 'micro', 'weighted').

        Возвращает:

            float: Значение F1.

        """

        return f1\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_confusion\_matrix(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> np.ndarray:

        """

        Строит матрицу ошибок (confusion matrix).

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            ndarray: Матрица размера [n\_classes, n\_classes].

        """

        return confusion\_matrix(y\_true, y\_pred)

    def get\_metrics\_report(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> Dict[str, float]:

        """

        Генерирует словарь с основными метриками классификации.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Способ усреднения (macro, micro, weighted).

        Возвращает:

            Dict[str, float]: Метрики: accuracy, precision, recall, f1.

        """

        return {

            "accuracy": self.calculate\_accuracy(y\_true, y\_pred),

            "precision": self.calculate\_precision(y\_true, y\_pred, average),

            "recall": self.calculate\_recall(y\_true, y\_pred, average),

            "f1": self.calculate\_f1(y\_true, y\_pred, average),

        }

if \_\_name\_\_ == "\_\_main\_\_":

    manager = DatasetManager(source="sklearn")

    manager.preprocess()

    manager.remove\_feature("total\_phenols")

    manager.split\_data(test\_size=0.2, stratify=True)

    manager.balance\_classes()

Продолжение Листинга Г

    X\_train, y\_train = manager.get\_training\_data()

    X\_test, y\_test = manager.get\_testing\_data()

    decision\_tree = DecisionTreeModel(criterion='gini', max\_depth=None)

    decision\_tree.fit(X\_train, y\_train)

    y\_pred = decision\_tree.predict(X\_test)

    report = decision\_tree.get\_metrics\_report(y\_test, y\_pred)

    print("\nОтчет о метриках классификации:")

    for metric, value in report.items():

        print(f"- {metric}: {value:.4f}")

    print("\nМатрица ошибок:")

    print(decision\_tree.calculate\_confusion\_matrix(y\_test, y\_pred))

    decision\_tree.plot(feature\_names=X\_train.columns.tolist(), class\_names=[str(cls) for cls in decision\_tree.classes\_])

### Приложение Д

Файл RandomForest.py с использованием готовой реализации модели Random Forest

Листинг Д – Файл RandomForest.py

import numpy as np

import pandas as pd

from typing import Union, Optional, Dict, List

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

from sklearn.tree import plot\_tree

import matplotlib.pyplot as plt

from dataset\_manager import DatasetManager

class RandomForestModel:

    def \_\_init\_\_(

        self,

        n\_estimators: int = 10,

        criterion: str = "gini",

        max\_depth: Optional[int] = None,

        max\_features: Optional[str] = "sqrt",

        random\_state: int = 42

    ) -> None:

        """

        Инициализирует классификатор на основе случайного леса.

        Параметры:

            n\_estimators (int): Количество деревьев в лесу.

            criterion (str): Критерий для оценки качества разбиения:

                - 'gini': индекс Джини;

                - 'entropy': информация по Шеннону.

            max\_depth (Optional[int]): Максимальная глубина деревьев.

            max\_features (str): Количество признаков для выбора при разделении:

                - 'sqrt': корень из числа признаков;

                - 'log2': логарифм по основанию 2;

                - int/float: конкретное количество или доля признаков.

            random\_state (int): Начальное значение генератора случайных чисел.

        """

        self.n\_estimators = n\_estimators

        self.criterion = criterion

        self.max\_depth = max\_depth

        self.max\_features = max\_features

        self.random\_state = random\_state

        self.model: Optional[RandomForestClassifier] = None

        self.classes\_: Optional[np.ndarray] = None

    def fit(

        self,

        X\_train: Union[pd.DataFrame, np.ndarray],

        y\_train: Union[pd.Series, np.ndarray]

    ) -> None:

        """

        Обучает модель случайного леса по предоставленным обучающим данным.

        Параметры:

            X\_train (DataFrame | ndarray): Матрица признаков обучающей выборки.

            y\_train (Series | ndarray): Вектор истинных меток классов.

        """

Продолжение Листинга Д

        self.model = RandomForestClassifier(

            n\_estimators=self.n\_estimators,

            criterion=self.criterion,

            max\_depth=self.max\_depth,

            max\_features=self.max\_features,

            random\_state=self.random\_state

        )

        self.model.fit(X\_train, y\_train)

        self.classes\_ = self.model.classes\_

    def predict(

        self,

        X\_test: Union[pd.DataFrame, np.ndarray]

    ) -> np.ndarray:

        """

        Предсказывает метки классов для новых объектов.

        Параметры:

            X\_test (DataFrame | ndarray): Матрица признаков тестовой выборки.

        Возвращает:

            ndarray: Предсказанные метки классов.

        Исключения:

            RuntimeError: если модель не обучена.

        """

        if self.model is None:

            raise RuntimeError("Сначала обучите модель с помощью fit().")

        return self.model.predict(X\_test)

    def plot\_tree(

        self,

        tree\_idx: int = 0,

        feature\_names: Optional[List[str]] = None,

        class\_names: Optional[List[str]] = None

    ) -> None:

        """

        Визуализирует структуру одного дерева из случайного леса.

        Параметры:

            tree\_idx (int): Индекс дерева для отображения (по умолчанию 0).

            feature\_names (list): Список имён признаков.

            class\_names (list): Список имён классов.

        """

        if self.model is None:

            raise RuntimeError("Сначала обучите модель.")

        plt.figure(figsize=(16, 10))

        plot\_tree(

            self.model.estimators\_[tree\_idx],

            filled=True,

            feature\_names=feature\_names,

            class\_names=class\_names

        )

        plt.title(f"Дерево №{tree\_idx} случайного леса")

        plt.show()

    def get\_feature\_importance(self) -> pd.DataFrame:

        """

        Возвращает важность признаков в обученной модели.

Продолжение Листинга Д

        Возвращает:

            DataFrame: Таблица с признаками и их важностью, отсортированная по убыванию.

        """

        if self.model is None:

            raise RuntimeError("Сначала обучите модель.")

        return pd.DataFrame({

            'feature': self.model.feature\_names\_in\_,

            'importance': self.model.feature\_importances\_

        }).sort\_values('importance', ascending=False)

    def calculate\_accuracy(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

        """

        Вычисляет метрику Accuracy — долю правильно классифицированных объектов.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            float: Значение accuracy ∈ [0, 1].

        """

        return accuracy\_score(y\_true, y\_pred)

    def calculate\_precision(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет метрику Precision — точность предсказания классов.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Стратегия усреднения ('macro', 'micro', 'weighted').

        Возвращает:

            float: Значение precision.

        """

        return precision\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_recall(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет метрику Recall — полноту предсказания.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Стратегия усреднения ('macro', 'micro', 'weighted').

        Возвращает:

            float: Значение recall.

        """

        return recall\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_f1(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> float:

        """

        Вычисляет F1-меру — гармоническое среднее точности и полноты.

Продолжение Листинга Д

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Способ усреднения ('macro', 'micro', 'weighted').

        Возвращает:

            float: Значение F1-метрики.

        """

        return f1\_score(y\_true, y\_pred, average=average, zero\_division=0)

    def calculate\_confusion\_matrix(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> np.ndarray:

        """

        Строит матрицу ошибок (confusion matrix) по результатам классификации.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

        Возвращает:

            ndarray: Матрица размера [n\_classes, n\_classes].

        """

        return confusion\_matrix(y\_true, y\_pred)

    def get\_metrics\_report(self, y\_true: np.ndarray, y\_pred: np.ndarray, average: str = "macro") -> Dict[str, float]:

        """

        Возвращает сводный отчёт по метрикам классификации.

        Параметры:

            y\_true (ndarray): Истинные метки.

            y\_pred (ndarray): Предсказанные метки.

            average (str): Стратегия усреднения (macro, micro, weighted).

        Возвращает:

            Dict[str, float]: Метрики: accuracy, precision, recall, f1.

        """

        return {

            "accuracy": self.calculate\_accuracy(y\_true, y\_pred),

            "precision": self.calculate\_precision(y\_true, y\_pred, average),

            "recall": self.calculate\_recall(y\_true, y\_pred, average),

            "f1": self.calculate\_f1(y\_true, y\_pred, average),

        }

if \_\_name\_\_ == "\_\_main\_\_":

    manager = DatasetManager(source="sklearn")

    manager.preprocess()

    manager.remove\_feature("total\_phenols")

    manager.split\_data(test\_size=0.2, stratify=True)

    X\_train, y\_train = manager.get\_training\_data()

    X\_test, y\_test = manager.get\_testing\_data()

    rf = RandomForestModel(

        n\_estimators=5,

        criterion='gini',

        max\_depth=5,

        max\_features='sqrt',

        random\_state=42

    )

    rf.fit(X\_train, y\_train)

Окончание Листинга Д

    y\_pred = rf.predict(X\_test)

    report = rf.get\_metrics\_report(y\_test, y\_pred)

    print("\nОтчет о метриках классификации:")

    for metric, value in report.items():

        print(f"- {metric}: {value:.4f}")

    print("\nМатрица ошибок:")

    print(rf.calculate\_confusion\_matrix(y\_test, y\_pred))

    print("\nВажность признаков:")

    print(rf.get\_feature\_importance().to\_string(index=False))

    rf.plot\_tree(

        tree\_idx=0,

        feature\_names=X\_train.columns.tolist(),

        class\_names=[str(cls) for cls in rf.classes\_]

    )