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

Институт Информационных Технологий

Кафедра Вычислительной Техники

ПРАКТИЧЕСКАЯ РАБОТА №3

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

Современные интеллектуальные информационные системы активно применяются в задачах анализа данных, автоматизации принятия решений и построения прогностических моделей. Одним из важнейших направлений обработки информации является задача классификации, заключающаяся в отнесении объектов к заранее известным категориям на основании признаков.

Классификация используется в самых различных предметных областях: от медицинской диагностики и финансового скоринга до биоинформатики и анализа потребительского поведения. Эффективность алгоритма классификации зависит от качества предварительной обработки данных, выбора информативных признаков, устойчивости модели к выбросам и дисбалансу классов, а также от адекватности выбранного алгоритма.

В рамках данной работы проводится анализ применимости различных алгоритмов классификации для решения задачи многоклассового распознавания. Для построения и оценки моделей используются как стандартные средства библиотек машинного обучения, так и собственные реализации алгоритмов. Также выполняется расчёт ключевых метрик качества, позволяющих оценить точность, полноту и общую эффективность построенной модели классификатора.

1 ПОСТАНОВКА ЗАДАЧИ

Цель работы: приобрести навыки классификации как инструмента категориального и предикативного анализа данных с контролируемым обучением.

Задачи: определить предметную область решаемой задачи, найти или сгенерировать набор данных для выбранной задачи, проведя предварительную предобработку и подготовку данных, выбрать модель классификации, провести её обучение и тестирование, определить качество модели с помощью метрик потерь, изучить алгоритмы классификации, написать программный код для реализации указанных алгоритмов, провести бинарную и/или многоклассовую классификацию данных подготовленного набора.

2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1 Алгоритм KNN

Алгоритм k-ближайших соседей (k-nearest neighbors, KNN) относится к числу простых и интуитивно понятных методов классификации. Его суть заключается в том, что для нового объекта определяется класс на основе классов ближайших к нему объектов из обучающей выборки. Алгоритм не требует этапа обучения в традиционном смысле и относится к ленивым (instance-based) методам обучения.

Классификация объекта происходит по большинству голосов среди k ближайших соседей, найденных по выбранной метрике расстояния. При этом может применяться взвешивание голосов: более близким объектам присваивается больший вес, чем дальним.

Алгоритм KNN включает следующие шаги:

1. Выбор параметра k – количества ближайших соседей.
2. Вычисление расстояний от классифицируемого объекта до всех объектов обучающей выборки. Обычно используется одна из метрик:

- Евклидова метрика (Формула 2.1.1):

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \quad (2.1.1)$$

- Манхэттенская метрика (Формула 2.1.2):

$$d(x, y) = \sum_{i=1}^{n}|x_i - y_i| \quad (2.1.2)$$

- Косинусная мера (Формула 2.1.3):

$$d(x, y) = 1 - \frac{x * y}{\|x\| * \|y\|} \quad (2.1.3)$$

3. Выбор k ближайших объектов (с минимальным расстоянием).
4. Определение метки класса. Если используется равное голосование (uniform), побеждает класс, наиболее часто встречающийся среди соседей. Если используется взвешенное голосование (distance), вес каждого соседа обратно пропорционален расстоянию (Формула 2.1.4).

$$w_i = \frac{1}{d(x, x_i)+\varepsilon}, \tag{2.1.4}$$

где ε – малое число, чтобы избежать деления на ноль.

5. Присвоение метки класса объекту на основе голосов.

Алгоритм KNN эффективен в задачах, где границы между классами несложны, а также когда требуется быстрое прототипирование без обучения модели. Однако при использовании его на практике необходимо тщательно подбирать значение параметра k, а также учитывать важность масштабирования и балансировки классов.

2.2 Метрики классификации

Для оценки качества работы классификационных алгоритмов применяются различные метрики качества, отражающие, насколько точно и надёжно модель предсказывает метки классов. В данной работе рассматриваются следующие основные метрики: accuracy, precision, recall, F1-мера, а также матрица ошибок (confusion matrix).

1. Accuracy (доля правильных предсказаний)

Метрика accuracy (точность в смысле «правильности» предсказаний) отражает долю правильно классифицированных объектов от общего числа наблюдений (Формула 2.2.1).

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \tag{2.2.1}$$

где TP – объекты, правильно отнесённые к положительному классу;

TN – объекты, правильно отнесённые к отрицательному классу;

FP – объекты, ошибочно отнесённые к положительному классу;

FN – объекты, ошибочно отнесённые к отрицательному классу.

Метрика эффективна при сбалансированных классах, но может вводить в заблуждение при дисбалансе.

2. Precision (Точность)

Метрика precision показывает, какую долю предсказанных положительных объектов действительно составляют положительные примеры (Формула 2.2.2).

$$Precision = \frac{TP}{TP+FP} \quad (2.2.2)$$

Высокое значение precision означает, что среди объектов, отнесённых моделью к положительному классу, большинство действительно принадлежат к этому классу.

3. Recall (полнота)

Метрика recall отражает долю правильно предсказанных положительных объектов среди всех реально положительных (Формула 2.2.3).

$$Recall = \frac{TP}{TP+FN} \quad (2.2.3)$$

Высокая полнота указывает на то, что модель пропускает мало объектов положительного класса.

4. F1-Score (F-мера)

F1-Score — гармоническое среднее между Precision и Recall, позволяющее оценить баланс между ними (Формула 2.2.4).

$$F1 = 2 * \frac{Precision*Recall}{Precision+Recall} \quad (2.2.4)$$

Значение F1 варьируется от 0 до 1, где 1 соответствует идеальной классификации.

5. Confusion matrix (матрица ошибок)

Матрица ошибок (confusion matrix) позволяет визуально отразить, как модель путает классы. Это квадратная таблица размером $N * N$, где N — число классов. Строки представляют истинные метки (диагональные элементы), а столбцы — предсказанные.

2.3 Дерево решений

Дерево решений (Decision Tree) — это один из простейших и наиболее интерпретируемых алгоритмов классификации и регрессии. Его основная идея заключается в построении древовидной структуры, в узлах которой выполняются проверки по значениям признаков, а в листьях — находятся метки классов или числовые прогнозы.

Алгоритм рекурсивно разбивает обучающее пространство на подпространства, основываясь на признаках, которые наилучшим образом разделяют данные. Такое разбиение продолжается до тех пор, пока не будут достигнуты листья дерева, содержащие итоговые решения (классы).

Внутренние узлы содержат логические правила вида $x_j \leq t$, где x_j - значения j-ого признака, а t - порог. Ветви представляют возможные исходы проверки условия. Листья содержат предсказанные метки классов.

Шаги алгоритма:

1. Выбирается наилучший признак и порог для разбиения текущей выборки.
2. Выбор осуществляется по метрике качества.
3. Данные разбиваются на две подгруппы.
4. Для каждой из подгрупп повторяется процесс рекурсивно.
5. Процесс завершается, если достигнуты максимальная глубина, минимальное количество объектов в узле, все объекты в узле

принадлежат одному классу.

Для выбора оптимального признака и порога разделения используются метрики, минимизирующие неоднородность данных:

1. Энтропия — мера хаоса в данных (Формула 2.3.1).

$$\text{Энтропия}(S) = -\sum_{i=1}^{C} p_i \log_2 p_i \ , \tag{2.3.1}$$

где p_i – доля объектов класса i в подмножестве S;

C – число классов.

2. Индекс Джини — мера неоднородности (Формула 2.3.2).

$$\text{Джини}(S) = 1 - \sum_{i=1}^{C} p_i^2 \tag{2.3.2}$$

2.4 Случайный лес

Случайный лес (Random Forest) — это ансамблевый метод машинного обучения, основанный на построении множества решающих деревьев и агрегации их предсказаний. Он относится к классу бэггинг-методов (bagging), которые направлены на уменьшение переобучения и повышение обобщающей способности модели.

Случайный лес создаёт множество деревьев решений, каждое из которых обучается на случайной подвыборке исходных данных. При этом в каждом узле дерева для разделения выбирается случайное подмножество признаков. Итоговое решение принимается на основе голосования деревьев (в задаче классификации) или усреднения (в задаче регрессии).

Шаги алгоритма:

1. Для каждого дерева случайным образом выбирается подмножество объектов из обучающей выборки с возвращением (bootstrapping). При построении дерева в каждом узле выбирается случайное подмножество признаков для нахождения лучшего разделения.

2. Шаг 1 повторяется для заданного числа деревьев n.
3. Предсказание на основе голосования большинства деревьев (Формула 2.4.1).

$$y = argmax_c \sum_{t=1}^{T} I(h_t(x) = c) , \quad (2.4.1)$$

где $h_t(x)$ – предсказание t-го дерева;

T – число деревьев.

3 ДОКУМЕНТАЦИЯ К ДАННЫМ

3.1 Описание предметной области

В качестве набора данных выбран широко известный Wine Dataset (данные о сортах итальянского вина, полученных в результате химического анализа образцов), хранящийся в открытом репозитории UCI Machine Learning Repository и доступный в библиотеке scikit-learn. Датасет содержит результаты аналитических измерений для красных и белых вин, произведённых тремя различными винодельческими культурами из региона северо-западной Италии. Целью сбора таких данных является изучение химических и физических свойств вина, которые позволяют не только классифицировать образцы по сорту, но и делать выводы о качестве продукта и возможных дефектах при его изготовлении.

3.2 Анализ данных

В исходных данных представлено 178 образцов вин. Каждый образец соответствует конкретной бутылке вина одного из трех сортов (классов), обозначенных как «Class 0», «Class 1» и «Class 2». Эти сорта соответствуют коммерческой классификации винодельческих культур:

- class 0 – сорт «Barolo»;
- class 1 – сорт «Grignolino»;
- class 2 – сорт «Barbera».

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

Исходные данные представлены в виде таблицы с 14 колонками (13 признаков и целевая метка класса). Каждый объект содержит 13 числовых признаков, характеризующих физико-химическое состояние образца, и целевое значение «target» – метку сорта (0, 1 или 2). Колонки с признаками включают:

1. Alcohol (спиртовая крепость): концентрация этанола в вине.
2. Malic acid (яблочная кислота, г/дм^3): остаточное содержание яблочной кислоты после ферментации.
3. Ash (зола, г/дм^3): количество минеральных веществ, оставшихся после сжигания пробы.
4. Alcalinity of ash (щелочность золы): показатель щелочности зольных составляющих.
5. Magnesium (магний, мг/дм^3): концентрация ионов магния.
6. Total phenols (общие фенолы, г/дм^3): суммарное содержание фенольных соединений, влияющих на цвет и вкус.
7. Flavanoids (флавоноиды, оптическая плотность): концентрация флавоноидных соединений, отвечающих за танинность и антиоксидантные свойства.
8. Nonflavanoid phenols (нефлавоноидные фенолы, оптическая плотность): другие фенольные соединения, не относящиеся к флавоноидам.
9. Proanthocyanins (проантоцианидины, оптическая плотность): полифенолы, влияющие на структуру и долговечность вина.
10. Color intensity (интенсивность цвета, оптическая плотность): показатель насыщенности цвета, получаемый спектрофотометрически.
11. Hue (оттенок): отношение определённых спектральных поглощений, характеризующее оттенок красного/фиолетового.
12. OD280/OD315 of diluted wines (отношение оптической плотности при 280 нм и 315 нм): индикатор содержания фенольных соединений при разведении.
13. Proline (пролин, мг/дм^3): аминокислота, одна из наиболее представленных в вине, влияющая на вкус и аромат.

Описательная статистика признаков отображена на Рисунке 3.2.1.

	count	mean	std	min	25%	50%	75%	max
alcohol	178.0	13.000618	0.811827	11.03	12.3625	13.050	13.6775	14.83
malic_acid	178.0	2.336348	1.117146	0.74	1.6025	1.865	3.0825	5.80
ash	178.0	2.366517	0.274344	1.36	2.2100	2.360	2.5575	3.23
alcalinity_of_ash	178.0	19.494944	3.339564	10.60	17.2000	19.500	21.5000	30.00
magnesium	178.0	99.741573	14.282484	70.00	88.0000	98.000	107.0000	162.00
total_phenols	178.0	2.295112	0.625851	0.98	1.7425	2.355	2.8000	3.88
flavanoids	178.0	2.029270	0.998859	0.34	1.2050	2.135	2.8750	5.08
nonflavanoid_phenols	178.0	0.361854	0.124453	0.13	0.2700	0.340	0.4375	0.66
proanthocyanins	178.0	1.590899	0.572359	0.41	1.2500	1.555	1.9500	3.58
color_intensity	178.0	5.058090	2.318286	1.28	3.2200	4.690	6.2000	13.00
hue	178.0	0.957449	0.228572	0.48	0.7825	0.965	1.1200	1.71
od280/od315_of_diluted_wines	178.0	2.611685	0.709990	1.27	1.9375	2.780	3.1700	4.00
proline	178.0	746.893258	314.907474	278.00	500.5000	673.500	985.0000	1680.00

Рисунок 3.2.1 – Описательная статистика признаков

В таблице приведены стандартные метрики для каждого из 13 признаков: среднее значение (mean), стандартное отклонение (std), минимум (min), первые и третьи квартили (25% и 75%), медиана (50%) и максимум (max).

Описательные статистики показывают, что часть признаков (например, Alcohol, Magnesium) распределены относительно компактно, в то время как другие признаки (Malic acid, Proline, Proanthocyanins) обладают более широким разбросом и выраженной скошенностью. Для корректной классификации потребуется стандартизация и, возможно, дополнительная обработка выбросов.

Распределение вин по классам представлено на Рисунке 3.2.2.

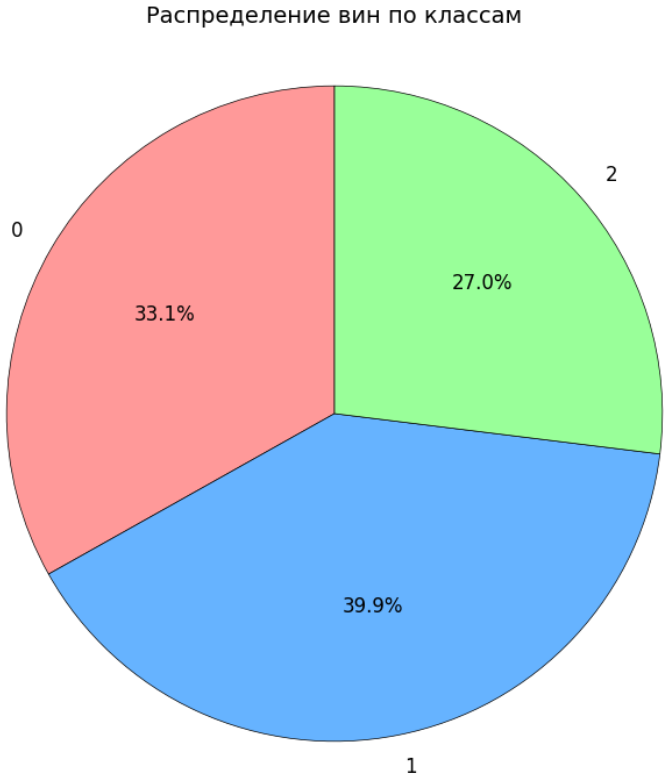

Рисунок 3.2.2 – Распределение вин по классам

Таким образом, класс 1 представлен наиболее обильно, а класс 2 – наименее. Несмотря на небольшую дисбалансировку, соотношение классов достаточно близко к равномерному.

Гистограммы распределений признаков представлены на Рисунке 3.2.3.

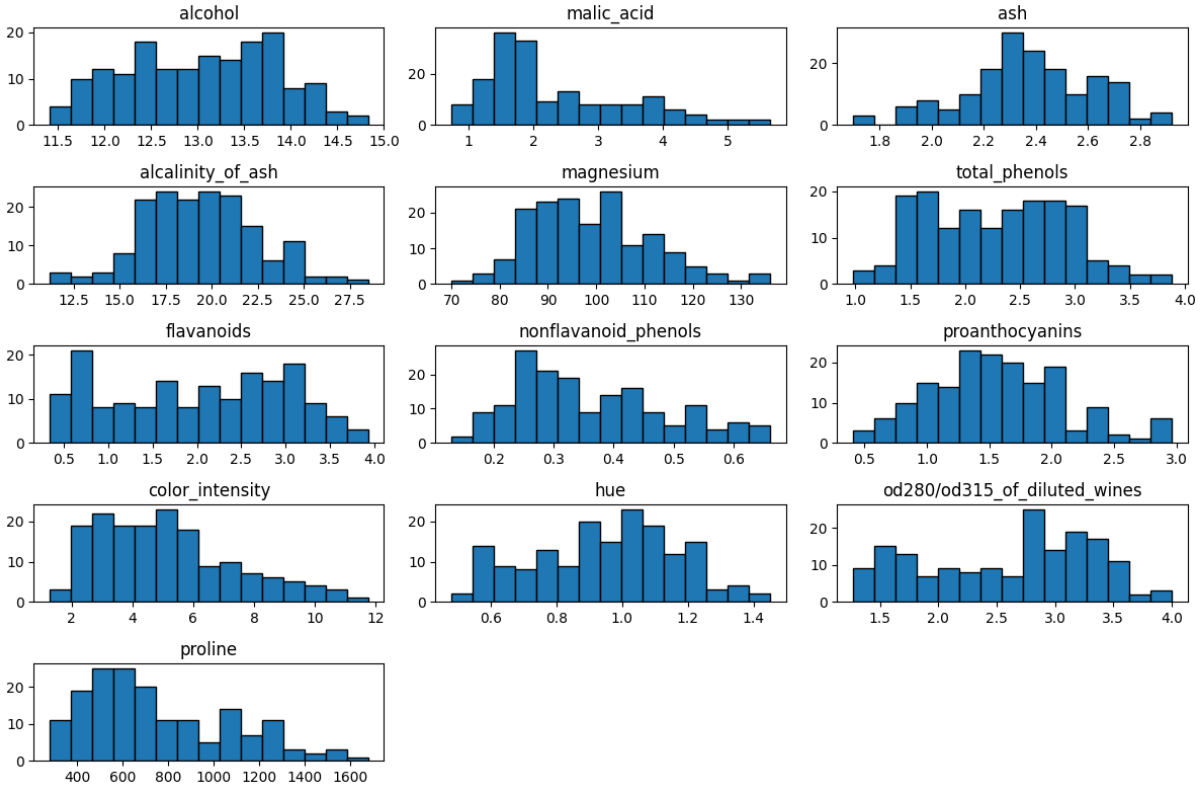

Рисунок 3.2.3 - Гистограммы распределений признаков

Практически все признаки имеют выраженную скошенность и отдельные выбросы. Это подчёркивает необходимость обработки экстремумов и обязательное масштабирование перед кластеризацией.

Матрица рассеяния для первых семи признаков представлена на Рисунке 3.2.4.

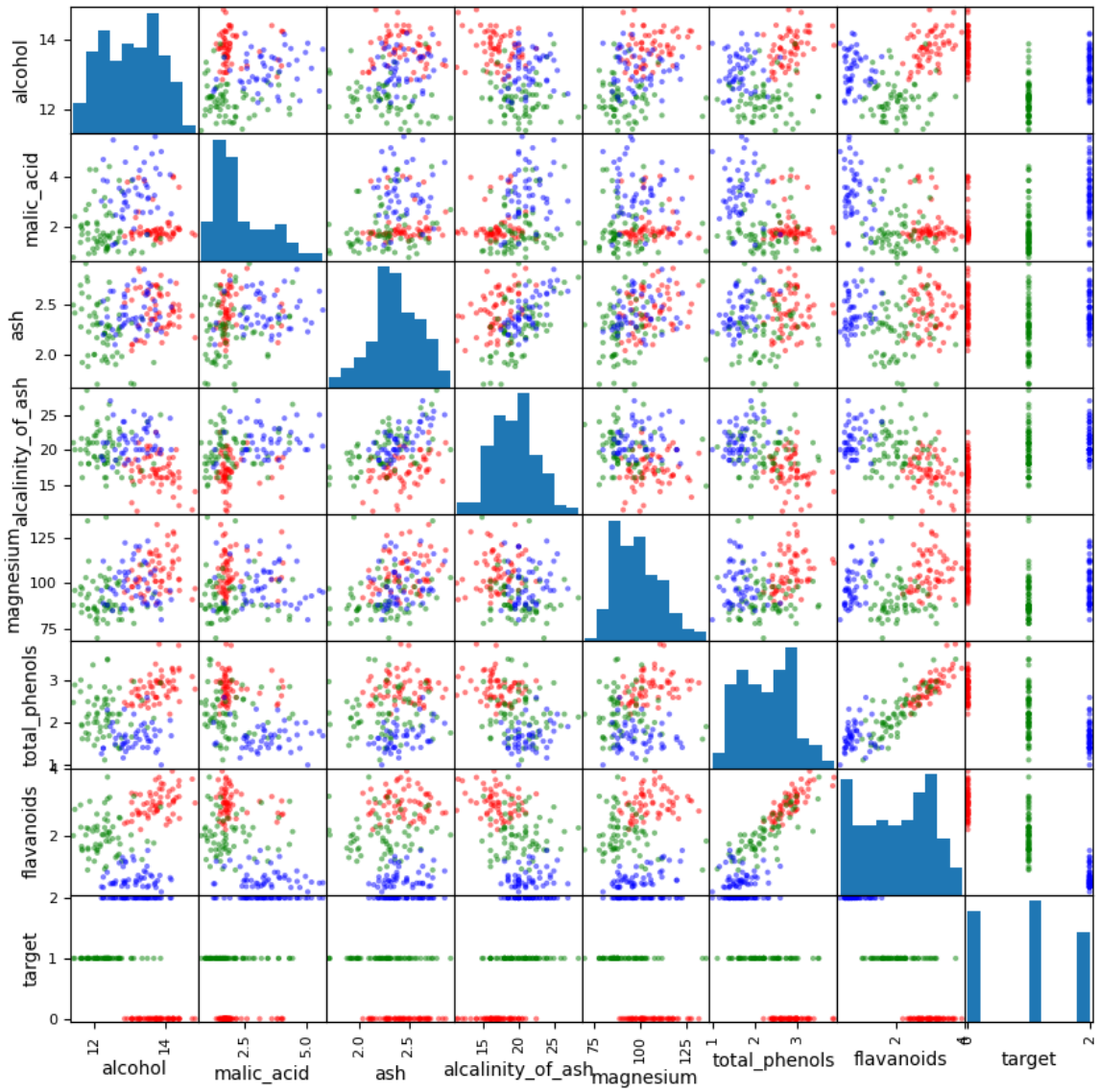

Рисунок 3.2.4 – Матрица рассеяния для первых семи признаков

На диагонали расположены гистограммы отдельных признаков (те же, что частично были на Рисунке 3.2.3, но только для первых семи). В ячейках показаны облака точек для пар признаков. Некоторые пары признаков обеспечивают достаточно чёткое различие классов. Между признаками Total phenols и Flavanoids прослеживается сильная корреляция.

Матрица корреляции признаков представлена на Рисунке 3.2.5.

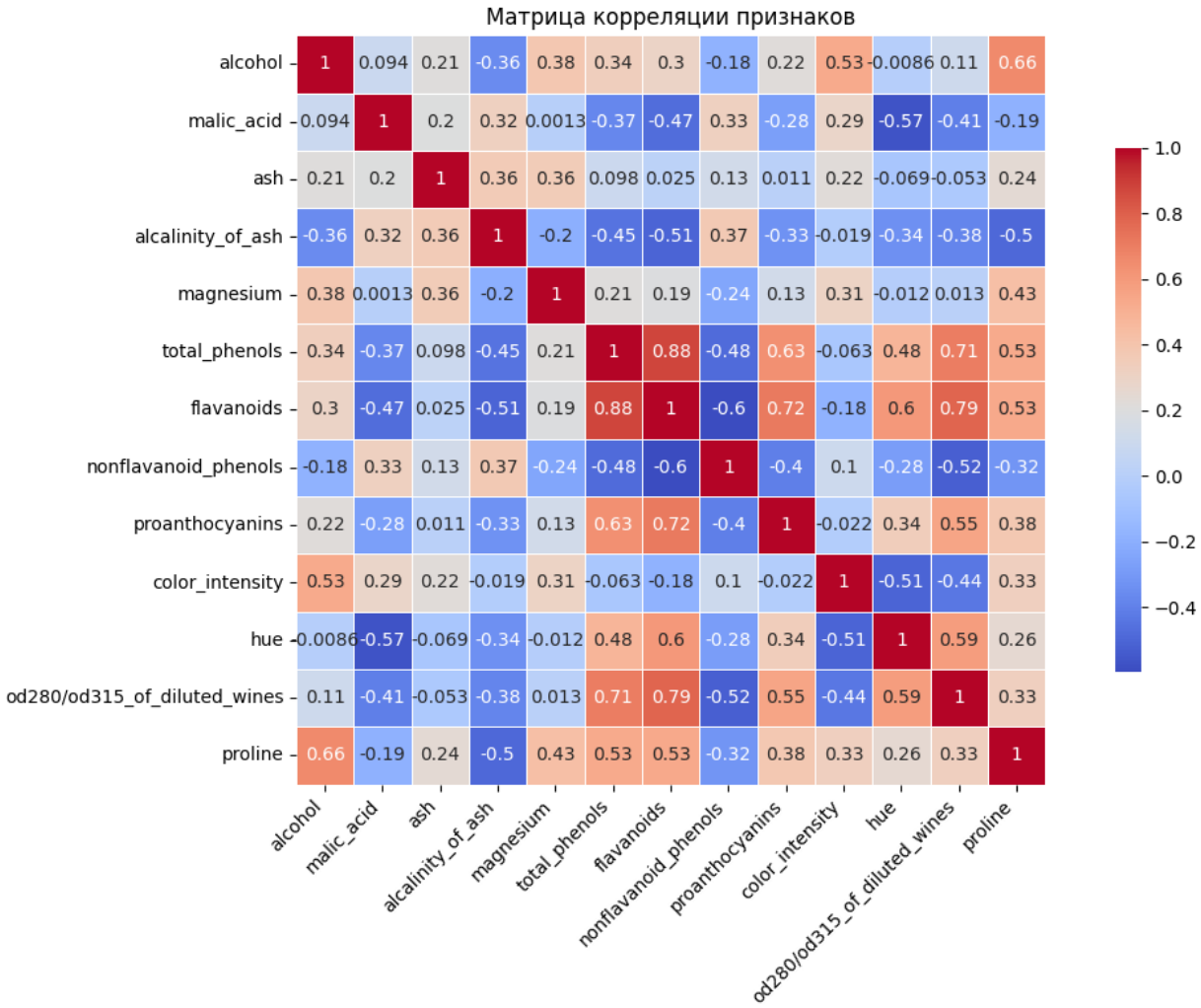

Рисунок 3.2.5 – Матрица корреляции признаков

Высокая корреляция между некоторыми признаками (Flavanoids и Total phenols) позволяет сократить размерность данных, исключив дублирующиеся признаки.

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 дубликатов не обнаружено.

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 x_{ij} признака j в образце i рассчитывается величина z_{ij} по Формуле 3.3.1.

$$z_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i}, \tag{3.3.1}$$

где μ_i – среднее отклонение признака j по всем образцам;

σ_i – стандартное отклонение признака j по всем образцам.

Образец i считается выбросом, если хотя бы один признак имеет $|z_{ij}| > z_{threshold}$.

В качестве порогового значения выбран $z_{threshold}$ = 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. В Wine Dataset признаки измеряются в разных физических и химических единицах. 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 j вычисляется среднее μ_j и стандартное отклонение σ_j. Каждое значение x_{ij} преобразуется по Формуле 3.3.2.

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}, \quad (3.3.2)$$

В результате стандартизированный признак x'_{ij} имеет среднее 0 и стандартное отклонение 1.

На Рисунке 3.2.5 показано, что коэффициент корреляции между «Total phenols» и «Flavanoids» составляет примерно 0.86. Это значит, что эти два признака фактически несут очень близкую информацию о составе вина. Когда признаки столь сильно коррелированы, они считаются практически линейно зависимыми: наличие одного позволяет почти однозначно восстановить второй. Поэтому признак «Total phenols» исключен из набора признаков, а соответствующий столбец в датасете удален.

Данные разделены на обучающую и тестовую выборки в соотношении 80:20. Обучающая выборка содержит 134 образца, а тестовая выборка – 34 образца.

Предобработанная обучающая выборка представлена на Рисунке 3.2.6.

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od315_of_diluted_wines	proline
12	0.906497	-0.560569	0.168155	-1.081011	-0.780942	0.763248	-0.588156	0.486458	0.216077	0.904074	0.425828	1.781616
30	0.691503	-0.624523	-0.038050	-0.084038	0.576556	1.201017	-1.154635	0.751614	0.797321	0.631179	0.397757	2.394206
36	0.021228	-0.633659	0.745529	-0.437803	-0.062266	0.427285	-0.588156	-0.214312	-0.387523	0.767627	-0.121558	1.106196
31	0.817970	-0.469208	-0.038050	-0.695086	0.416851	0.691983	0.463877	0.789494	-0.570838	1.267935	0.383722	0.744925
95	-1.205501	-0.240803	-2.759954	-0.598605	-0.142119	0.162588	-0.830933	-0.290071	-0.812278	1.449865	0.510041	-0.134691
...	...	...	...	...	...	...	...	...	...	...	...	...
39	0.666210	-0.578842	-0.244255	-1.016690	1.454937	1.302824	-0.183528	1.490264	0.453046	-0.005577	1.099534	0.132335
131	-0.130532	0.426137	1.364143	0.527010	-0.221972	-1.537585	1.354059	-1.521154	-0.231034	-0.824263	-0.402269	-0.480255
155	0.881204	1.842244	-0.450459	1.009416	-0.860794	-1.568127	1.273133	-0.763564	0.672130	-0.778781	-1.188259	-0.731574
26	0.337396	-0.569705	-0.945351	-0.759407	-0.381677	0.182949	-0.750007	-0.384770	-0.521656	0.312801	0.243366	1.671664
157	-0.269646	0.937763	-0.285496	0.044603	-0.860794	-1.374694	0.302026	-1.104480	2.299615	-1.051676	-1.188259	-0.213228

134 rows × 12 columns

Рисунок 3.2.6 – Предобработанная обучающая выборка

На данном этапе полученный очищенный набор признаков может использоваться в алгоритмах классификации.

4 ПРАКТИЧЕСКАЯ ЧАСТЬ

4.1 Алгоритм KNN

Для решения задачи классификации реализован алгоритм K-ближайших соседей с использованием библиотеки scikit-learn. Алгоритм написан в файле KNN.py, содержание которого представлено в Приложении Б.

В рамках эксперимента выбрано значение параметра k = 3, а стратегия взвешивания соседей — «distance», то есть чем ближе сосед, тем больший вес он имеет при голосовании.

После обучения модели на обучающих данных, сделаны предсказания на тестовой выборке. По результатам вычислены метрики (Таблица 4.1.1).

Таблица 4.1.1 – Метрики по результатам классификации с библиотечной реализацией

Метрика	Значение	Интерпретация
accuracy	0.9412	Модель правильно классифицировала 94,12% объектов из тестовой выборки
precision	0.9524	В среднем 95,24% объектов, отнесённых моделью к какому-либо классу, действительно принадлежат этому классу
recall	0.9487	Модель нашла в среднем 94,87% объектов, которые действительно принадлежат каждому из классов
f1	0.9466	Гармоническое среднее между точностью и полнотой. Значение F1 близко к 1, что подтверждает сбалансированное и точное поведение модели по всем классам

Также построена матрица ошибок (confusion matrix), позволяющая увидеть, какие именно классы были перепутаны моделью (Рисунок 4.1.1).

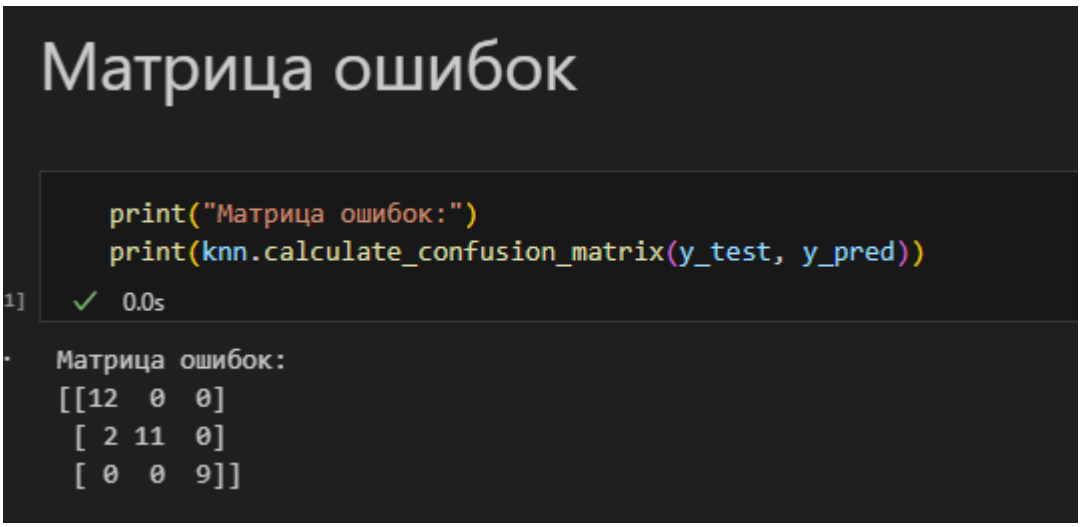

Рисунок 4.1.1 – Матрица ошибок по результатам классификации алгоритмом KNN с библиотечной реализацией

Для класса 1 два объекта классифицированы неверно, однако класс 0 и 2 распознаются идеально. В целом, модель KNN демонстрирует очень высокую точность классификации, и ошибки локализованы только между соседними классами, что допустимо и может быть вызвано схожестью признаков этих объектов.

Для демонстрации принципов работы алгоритма KNN также разработана его самостоятельная реализация на языке Python без использования библиотек машинного обучения. Код написан в файле KNN_custom.py, содержание которого представлено в Приложении В.

Использована евклидова метрика для вычисления расстояний между объектами, а также взвешенное голосование (вес голоса соседа обратно пропорционален расстоянию до классифицируемого объекта).

После обучения модели на обучающей выборке и предсказания меток для тестовых данных вычислены метрики качества (Таблица 4.1.2).

Таблица 4.1.2 – Метрики по результатам классификации с собственной реализацией

Метрика	Значение	Интерпретация
accuracy	0.8235	Модель правильно классифицировала 82,35% объектов из тестовой выборки

Продолжение Таблицы 4.1.2

precision	0.8426	В среднем 84,26% объектов, отнесённых моделью к какому-либо классу, действительно принадлежат этому классу
recall	0.8348	Модель нашла в среднем 83,48% объектов, которые действительно принадлежат каждому из классов
f1	0.8387	Гармоническое среднее между точностью и полнотой. Значение F1 близко к 0.85, что подтверждает сбалансированное и точное поведение модели по всем классам

Вероятнее всего, самописная реализация уступает библиотечной версии из-за недостаточной оптимизации.

Матрица ошибок для самописной реализации представлена на Рисунке 4.1.2.

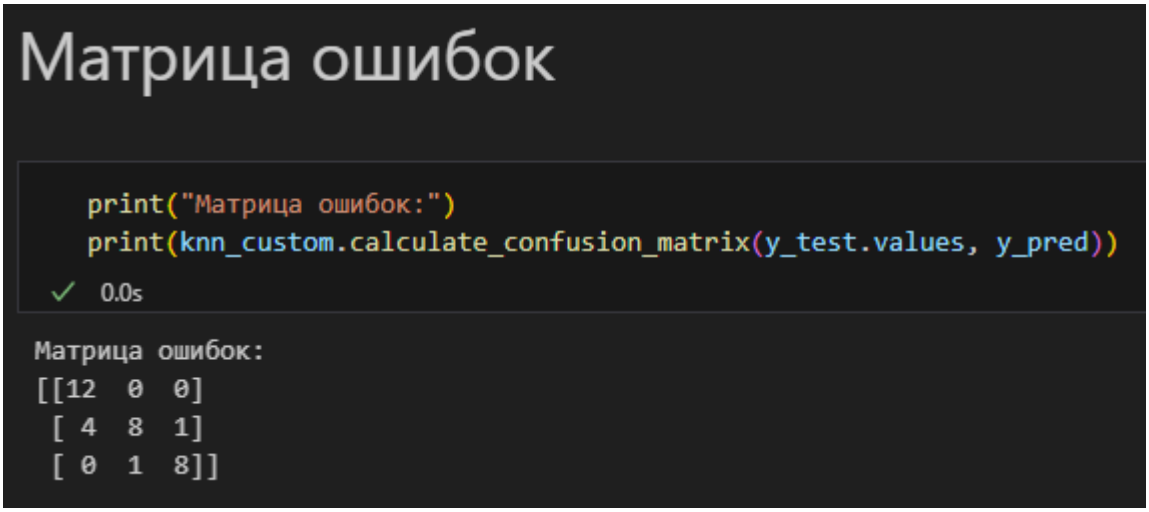

Рисунок 4.1.2 - Матрица ошибок по результатам классификации алгоритмом KNN с собственной реализацией

Увеличение числа ошибок в первом классе говорит о том, что классы все-таки недостаточно сбалансированы, модель не обучилась.

4.2 Дерево решений

Дополнительно реализована и протестирована модель классификации на основе дерева решений с использованием библиотеки scikit-learn. Классификатор написан в файле DecisionTree.py, содержание которого представлено в Приложении Г.

В качестве критерия расщепления использовался индекс Джини, а глубина дерева не ограничивалась (max_depth=None), что позволяет дереву расти до тех пор, пока каждый лист не станет чистым или пока не останется достаточно признаков для разбиения.

После обучения выполнена классификация тестовой выборки. Полученные метрики представлены в Таблице 4.2.1.

Таблица 4.2.1 – Метрики по результатам классификации с библиотечной реализацией Decision tree

Метрика	Значение	Интерпретация
accuracy	0.8235	Модель правильно классифицировала 82,35% объектов из тестовой выборки
precision	0.8426	В среднем 84,26% объектов, отнесённых моделью к какому-либо классу, действительно принадлежат этому классу
recall	0.8348	Модель нашла в среднем 83,48% объектов, которые действительно принадлежат каждому из классов
f1	0.8244	Гармоническое среднее между точностью и полнотой. Значение F1 близко к 0.85, что подтверждает сбалансированное и точное поведение модели по всем классам

Также с помощью встроенной функции plot_tree() выполнена визуализация структуры обученного дерева, позволяющая наглядно увидеть правила принятия решений и важность признаков (Рисунок 4.2.1).

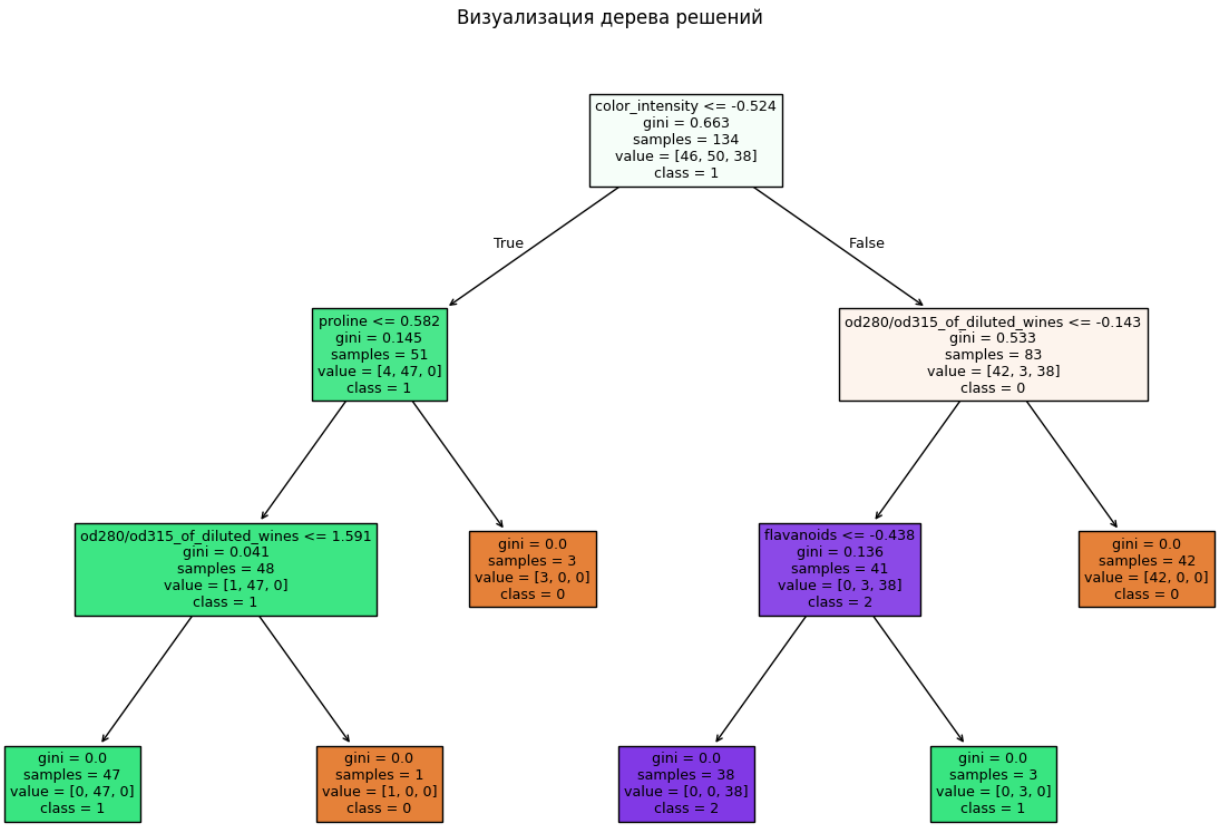

Рисунок 4.2.1 – Визуализация дерева принятия решений

4.3 Случайный лес

В завершении реализована модель классификации на основе алгоритма случайного леса с использованием библиотеки sklearn.

Модель написана в файле RandomForest.py, содержание которого представлено в Приложении Д.

Для построения случайного леса использовались следующие параметры:

- количество деревьев (n_estimators) – 100;
- критерий разбиения – индекс Джини (gini);
- глубина деревьев – без ограничений;
- количество признаков для выбора при расщеплении – sqrt.

После обучения модели проведена оценка качества классификации с использованием основных метрик (Таблица 4.3.1).

Таблица 4.3.1 – Метрики по результатам классификации с библиотечной реализацией Random Forest

Метрика	Значение	Интерпретация
accuracy	0.9412	Модель правильно классифицировала 94,12% объектов тестовой выборки
precision	0.9410	В среднем 94,10% объектов, отнесённых моделью к какому-либо классу, действительно принадлежат этому классу
recall	0.9487	Модель нашла в среднем 94,87% объектов, которые действительно принадлежат каждому из классов
f1	0.9413	Гармоническое среднее между точностью и полнотой. Значение F1 близко к 1, что подтверждает сбалансированное и точное поведение модели по всем классам

Модель показала высокую точность (94.12%) и сбалансированные значения precision, recall и F1-меры. Это говорит о надежной и устойчивой классификации объектов тестовой выборки, а также о хорошей способности модели выявлять все классы без сильных перекосов. Данный результат превзошёл качество классификации, полученное от дерева решений и KNN.

Дополнительно выполнена оценка важности признаков с использованием встроенной функции feature_importances_. Распределение важности представлено в Таблице 4.3.2.

Таблица 4.3.2 – Распределение важности признаков

Признак	Важность
Flavanoids	0.477
color_intensity	0.286
Proline	0.149
malic_acid	0.027
od280/od315_of_diluted_wines	0.025
Magnesium	0.016
nonflavanoid_phenols	0.009
alcalinity_of_ash	0.006
Alcohol	0.006
ash	0.000
proanthocyanins	0.000
hue	0.000

Как видно из таблицы, наибольший вклад в предсказание модели внесли признаки flavanoids (почти 48% важности), color_intensity (28.5%) и proline (14.8%). Остальные признаки имели значительно меньший вклад, а такие как ash, proanthocyanins и hue не использовались моделью в качестве информативных при построении деревьев.

ЗАКЛЮЧЕНИЕ

В рамках практической работы выполнена задача многоклассовой классификации образцов вин на основе их физико-химических характеристик. Для решения использовались алгоритмы KNN, дерево решений и случайный лес. Набор данных Wine Dataset предобработан: удалены выбросы, проведена стандартизация признаков, исключены избыточные признаки.

Наивысшую точность (94.12%) показал алгоритм случайного леса, что обусловлено его способностью снижать переобучение за счёт агрегации множества деревьев.

KNN с библиотечной реализацией также продемонстрировал высокую точность (94.12%), но его производительность сильно зависит от выбора метрики расстояния и параметра k.

Дерево решений показало худший результат (82.35%), что связано с переобучением при неограниченной глубине.

Поставленные задачи выполнены. Экспериментально подтверждено, что алгоритмы на основе ансамблей (случайный лес) эффективнее для задач классификации с малым объёмом данных. Для промышленного применения требуется дальнейшая оптимизация и валидация на более репрезентативных данных.

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. Метод K-ближайших соседей (KNN). Принцип работы, разновидности и реализация с нуля на Python [Электронный ресурс]: Habr. URL: https://habr.com/ru/articles/801885/ (Дата обращения: 16.05.2025).
4. Машинное обучение: Классификация методом KNN. Теория и реализация. С нуля. На чистом Python [Электронный ресурс]: Habr. URL: https://habr.com/ru/articles/866636/ (Дата обращения: 17.05.2025).
5. Бэггинг и случайный лес. Ключевые особенности и реализация с нуля на Python [Электронный ресурс]: Habr. URL: https://habr.com/ru/articles/801161/ (Дата обращения: 19.05.2025).

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 pandas.plotting import scatter_matrix
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from typing import Optional, Dict, Tuple, List
from pandas import DataFrame, Series
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

class DatasetManager:
    def __init__(
        self,
        source: str = "sklearn",
        csv_path: Optional[str] = None,
    ) -> None:
        """
        Инициализирует менеджер датасета для загрузки, анализа, предобработки и
визуализации.

        Параметры:
            source (str): Источник данных.
                - "sklearn": загружаем встроенный датасет Wine из sklearn.
                - "csv": читаем CSV-файл по пути csv_path.
            csv_path (Optional[str]): Путь к CSV-файлу при source="csv".
                Если source="sklearn", игнорируется.
        """
        self.source: str = source
        self.csv_path: Optional[str] = csv_path
        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._extract_features_target()

    def _load_data(self) -> None:
        """
        Загружает исходный датасет в self.df.

        При source="sklearn" загружается Wine-датасет из sklearn.
        При source="csv" загружается CSV-файл по пути csv_path.

        Выбрасывает:
            ValueError: если source="csv" и csv_path не указан или source не
равен "sklearn"/"csv".
        """
        if self.source == "sklearn":
            raw = load_wine(as_frame=True)
            df0 = raw.frame.copy()
            self.df = df0
        elif self.source == "csv":
            if self.csv_path is None:
```

Продолжение Листинга А

```
                raise ValueError("При source='csv' необходимо указать путь
csv_path")
            self.df = pd.read_csv(self.csv_path)
        else:
            raise ValueError("source должен быть 'sklearn' или 'csv'")

        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 метку (если столбец 'target'
присутствует).

        После выполнения:
            - self.features будет содержать DataFrame только с признаками.
            - self.target будет содержать Series с метками классов (или None,
если 'target' отсутствует).
        """
        if self.df is None:
            raise RuntimeError("Данные не загружены. Сначала вызовите
_load_data().")

        if "target" in self.df.columns:
            self.target = self.df["target"].copy()
            self.features = self.df.drop(columns=["target"]).copy()
        else:
            self.target = None
            self.features = self.df.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.
            - "class_distribution": распределение по классам (если есть
self.target).

        Возвращает:
            Dict[str, DataFrame]: Словарь с DataFrame-статистиками.
        """
        if self.features is None:
            raise RuntimeError(
                "Признаки не выделены. Сначала вызовите
_extract_features_target()."
            )

        desc = self.features.describe().T
        self.stats["describe"] = desc
        corr = self.features.corr()
        self.stats["correlation_matrix"] = corr

        if self.target is not None:
            class_counts: Series = self.target.value_counts().sort_index()
            self.stats["class_distribution"] =
class_counts.to_frame(name="count")
```

Продолжение Листинга А

```
        return self.stats

    def preprocess(
        self,
        drop_duplicates: bool = True,
        drop_outliers: bool = True,
        z_thresh: float = 3.0,
    ) -> None:
        """
        Полная предобработка данных:
            1. Удаление дубликатов.
            2. Удаление выбросов по Z-оценке (если drop_outliers=True).
            3. Масштабирование признаков StandardScaler.

        Параметры:
            drop_duplicates (bool): Удалять ли полные дубликаты строк
(True/False).
            drop_outliers (bool): Удалять ли выбросы по Z-оценке (True/False).
            z_thresh (float): Порог Z-оценки; объекты, у которых хотя бы один
признак
                              имеет |z_score| > z_thresh, считаются выбросами.

        После выполнения:
            - self.df обновляется без дубликатов и выбросов.
            - self.features обновляются (признаки из очищенного DataFrame).
            - self.target обновляется (метки из очищенного DataFrame).
            - self.scaled_features заполняется DataFrame-ом масштабированных
признаков.
        """
        if self.df is None:
            raise RuntimeError("Данные не загружены. Сначала вызовите
_load_data().")

        df_proc: DataFrame = self.df.copy()

        if drop_duplicates:
            before = df_proc.shape[0]
            df_proc = df_proc.drop_duplicates().reset_index(drop=True)
            after = df_proc.shape[0]
            print(f"Удалено дубликатов: {before - after}")

        if drop_outliers:
            df_no_target = df_proc.drop(columns=["target"], 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)
            after_out = df_proc.shape[0]
            print(f"Удалено выбросов: {before_out - after_out}")

        scaler = StandardScaler()
        feat: DataFrame = df_proc.drop(columns=["target"], errors="ignore")
        scaled_array = scaler.fit_transform(feat)
        scaled_df = pd.DataFrame(scaled_array, columns=feat.columns,
index=feat.index)

        self.df = df_proc
        if "target" in df_proc.columns:
```

Продолжение Листинга А

```
            self.target = df_proc["target"].copy()
            self.features = df_proc.drop(columns=["target"]).copy()
        else:
            self.target = None
            self.features = df_proc.copy()

        self.scaled_features = scaled_df
        print(
            "Предобработка завершена: дубликаты и выбросы (если указано)
удалены, 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_scatter_matrix(
        self,
        with_target: bool = True,
        figsize: Tuple[int, int] = (10, 10),
    ) -> None:
        """
        Строит матрицу рассеяния (pairplot) для первых 5–7 признаков.

        Параметры:
            with_target (bool): Если True и self.target определён, раскрашивает
точки по классам.
            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()."
            )

        num_to_plot = min(7, len(self.features.columns))
```

Продолжение Листинга А

```
        df_plot = self.features.iloc[:, :num_to_plot].copy()

        if with_target and self.target is not None:
            df_plot["target"] = self.target.values
            colors = {0: "red", 1: "green", 2: "blue"}
            scatter_matrix(
                df_plot,
                figsize=figsize,
                diagonal="hist",
                color=df_plot["target"].map(colors),
                alpha=0.5,
            )
        else:
            scatter_matrix(df_plot, figsize=figsize, diagonal="hist", alpha=0.5)

        plt.suptitle("Матрица рассеяния признаков", y=1.02)
        plt.show()

    def visualize_correlation_heatmap(self, figsize: Tuple[int, int] = (12, 8))
-> None:
        """
        Строит «приятную» тепловую карту корреляций между признаками с помощью
seaborn.

        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()."
            )

        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 get_preprocessed_data(self) -> Tuple[DataFrame, Optional[Series]]:
        """
        Возвращает масштабированные признаки и метки (если есть) для дальнейшего
анализа/кластеризации.

        Возвращает:
            Tuple[DataFrame, Optional[Series]]:
                - DataFrame: self.scaled_features (масштабированные признаки).
                - Series или None: self.target (метки классов, если были
изначально).
        Выбрасывает:
            RuntimeError: если self.scaled_features ещё не вычислены (не вызван
preprocess()).
```

Продолжение Листинга А

```
        """
        if self.scaled_features is None:
            raise RuntimeError(
                "Данные ещё не предобработаны. Сначала вызовите preprocess()."
            )
        return self.scaled_features, self.target

    def visualize_class_distribution(
        self,
        figsize: Tuple[int, int] = (8, 8),
        title: str = "Распределение по классам",
        colors: Optional[List[str]] = None,
        autopct: str = "%1.1f%%",
        startangle: int = 90,
    ) -> None:
        """
        Строит круговую диаграмму распределения объектов по классам.

        Параметры:
            figsize (Tuple[int, int]): Размер фигуры (ширина, высота) в дюймах.
            title (str): Заголовок диаграммы.
            colors (Optional[List[str]]): Список цветов для секторов.
            autopct (str): Формат отображения процентных значений.
            startangle (int): Угол начала первой секции.
        """
        if "class_distribution" not in self.stats:
            raise RuntimeError(
                "Распределение по классам не вычислено. Вызовите
compute_basic_statistics()."
            )

        class_dist = self.stats["class_distribution"]
        labels = class_dist.index.astype(str).tolist()
        sizes = class_dist["count"].tolist()

        if not colors:
            colors = ["#ff9999", "#66b3ff", "#99ff99", "#ffcc99"]

        plt.figure(figsize=figsize)
        plt.pie(
            sizes,
            labels=labels,
            colors=colors,
            autopct=autopct,
            startangle=startangle,
            textprops={"fontsize": 12},
            wedgeprops={"edgecolor": "black", "linewidth": 0.5},
        )
        plt.title(title, fontsize=14, pad=20)
        plt.axis("equal")
        plt.show()

    def remove_feature(self, feature_name: str) -> None:
        """
        Удаляет признак из текущего набора данных по его имени.

        Параметры:
            feature_name (str): Название удаляемого признака.

        Исключения:
            ValueError: если feature_name не является строкой.
```

Продолжение Листинга А

```
            KeyError: если признака с таким именем нет в self.features.
            RuntimeError: если self.features ещё не инициализирован (нет
данных).
        """
        if self.features is None:
            raise RuntimeError(
                "Набор признаков пуст. Сначала выполните загрузку данных и метод
_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 split_data(
        self,
        test_size: float = 0.2,
        random_state: int = 42,
        stratify: bool = True
    ) -> None:
        """
        Разделяет данные на обучающую и тестовую выборки.

        Параметры:
            test_size (float): Доля тестовых данных (по умолчанию 0.2).
            random_state (int): Seed для воспроизводимости.
            stratify (bool): Сохранять ли распределение классов (по умолчанию
True).
        """
        if self.scaled_features is None or self.target is None:
            raise RuntimeError("Сначала выполните предобработку данных
(preprocess())")

        stratify_param = self.target if stratify else None

        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,
```

Продолжение Листинга А

```
            stratify=stratify_param
        )

        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 balance_classes(
        self,
        sampler: str = "SMOTE",
        random_state: int = 42
    ) -> None:
        """
        Балансирует классы с помощью выбранного метода.

        Параметры:
            sampler (str): Метод балансировки ('SMOTE' или 'undersampling').
            random_state (int): Seed для воспроизводимости.
        """
        if not hasattr(self, 'X_train'):
            raise RuntimeError("Сначала выполните разделение данных
(split_data())")

        class_counts = self.y_train.value_counts()
        print("\nРаспределение классов до балансировки:")
        print(class_counts)

        if sampler == "SMOTE":
            sm = SMOTE(random_state=random_state)
            self.X_train, self.y_train = sm.fit_resample(self.X_train,
self.y_train)
        elif sampler == "undersampling":
            min_class = class_counts.idxmin()
            min_count = class_counts.min()

            dfs = []
            for class_label in self.y_train.unique():
                class_df = self.X_train[self.y_train == class_label]
                dfs.append(class_df.sample(min_count,
random_state=random_state))

            self.X_train = pd.concat(dfs)
            self.y_train = pd.Series([label for label, df in
zip(self.y_train.unique(), dfs)
                                      for _ in range(len(df))])
        else:
            raise ValueError("Доступные методы: 'SMOTE', 'undersampling'")

        print("\nРаспределение классов после балансировки:")
        print(self.y_train.value_counts())

    def get_training_data(self) -> Tuple[DataFrame, Series]:
        """Возвращает балансированные обучающие данные"""
        return self.X_train, self.y_train

    def get_testing_data(self) -> Tuple[DataFrame, Series]:
        """Возвращает тестовые данные"""
        return self.X_test, self.y_test

if __name__ == "__main__":
    manager = DatasetManager(source="sklearn")
```

Окончание Листинга А

```
    stats = manager.compute_basic_statistics()

    manager.visualize_class_distribution(
        title="Распределение вин по классам",
        colors=["#ff9999", "#66b3ff", "#99ff99"],
        autopct="%1.1f%%",
    )

    print("Описание признаков:")
    print(stats["describe"])
    if "class_distribution" in stats:
        print("\nРаспределение по классам:")
        print(stats["class_distribution"])

    manager.visualize_distributions()
    manager.visualize_scatter_matrix()
    manager.visualize_correlation_heatmap()

    manager.preprocess()

    manager.remove_feature("total_phenols")

    manager.split_data(test_size=0.2, stratify=True)

    manager.balance_classes(sampler="SMOTE")

    X_train, y_train = manager.get_training_data()
    X_test, y_test = manager.get_testing_data()
```

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_]
)
```