



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

Институт Информационных Технологий
Кафедра Вычислительной Техники

ПРАКТИЧЕСКАЯ РАБОТА №6

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
1 ПОСТАНОВКА ЗАДАЧИ	4
2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ	5
2.1 Метрики классификации	5
2.2 Дерево решений.....	6
2.3 Случайный лес.....	8
2.4 Лес экстремально случайных деревьев.....	9
2.5 AdaBoost.....	9
3 ДОКУМЕНТАЦИЯ К ДАННЫМ.....	11
3.1 Описание предметной области	11
3.2 Анализ данных.....	11
3.3 Предобработка данных	16
4 ПРАКТИЧЕСКАЯ ЧАСТЬ	19
4.1 Ансамбль с голосованием	19
4.2 Случайный лес.....	20
4.3 Лес экстремально случайных деревьев.....	22
4.4 AdaBoost.....	23
ЗАКЛЮЧЕНИЕ	25
СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ	27
ПРИЛОЖЕНИЯ.....	28

ВВЕДЕНИЕ

В условиях стремительного роста объёма и разнообразия данных методы машинного обучения играют всё более важную роль в извлечении ценной информации и принятии обоснованных решений. Задача классификации, в которой модель обучается на размеченном наборе данных и затем присваивает новые объекты к одному из заранее известных классов, является фундаментальной для таких областей, как распознавание образов, анализ потребительского поведения, диагностика заболеваний и многие другие. Однако зачастую ни одна «сильная» модель не способна одновременно обеспечивать высокую точность, устойчивость к шуму и отсутствие переобучения.

Ансамблевые методы (ensemble learning) позволяют объединять несколько «слабых» моделей в один «сильный» классификатор, компенсируя слабые стороны каждой отдельной модели и усиливая их совместные преимущества.

1 ПОСТАНОВКА ЗАДАЧИ

Цель работы: приобрести навыки создание, обучения и применения ансамблей моделей машинного обучения.

Задачи: определить предметную область решаемой задачи, найти или сгенерировать набор данных для выбранной задачи, проведя предварительную предобработку и подготовку данных, реализовать вручную ансамбль алгоритмов машинного обучения на основе простых ансамблевых приёмов, реализовать хотя бы один продвинутый ансамбль – бустинг, бэггинг или стекинг.

2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1 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 2.1.1).

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \quad (2.1.1)$$

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 2.1.2).

$$Precision = \frac{TP}{TP+FP} \quad (2.1.2)$$

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 2.1.3).

$$Recall = \frac{TP}{TP+FN} \quad (2.1.3)$$

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 2.1.4).

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (2.1.4)$$

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.

2.2 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 деления используются метрики, минимизирующие неоднородность данных:

1. Энтропия — мера хаоса в данных (Формула 2.2.1).

$$\text{Энтропия}(S) = - \sum_{i=1}^C p_i \log_2 p_i, \quad (2.2.1)$$

где p_i — доля объектов класса i в подмножестве S ;

C — число классов.

2. Индекс Джини — мера неоднородности (Формула 2.2.2).

$$\text{Джини}(S) = 1 - \sum_{i=1}^C p_i^2 \quad (2.2.2)$$

2.3 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 2.3.1).

$$y = \operatorname{argmax}_c \sum_{t=1}^T I(h_t(x) = c) , \quad (2.3.1)$$

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

T — число деревьев.

2.4 Лес экстремально случайных деревьев

Лес экстремально случайных деревьев – это ансамблевый метод, основанный на построении большого числа «урезанных» деревьев решений и объединении их предсказаний. В отличие от классического случайного леса (Random Forest), где для каждого узла подбирается оптимальное разбиение по критерию, в Extra-Trees разбиения выбираются полностью случайным образом: на каждом узле не только случайным образом выбирается подмножество признаков, но и для каждого выбранного признака формируется случайный порог (split threshold). Затем из всех этих произвольных разбиений выбирается то, которое даёт наилучшее разбиение. Таким образом, основная идея Extra-Trees заключается в максимальной стохастизации процесса построения каждого базового дерева. Вместо подвыборки объектов (bootstrap sampling), как в Random Forest, используются все доступные обучающие примеры (то есть bootstrap-агрегирование по объектам не применяется). При выборе разбиения в узле случайным образом генерируются пороги (split candidates) для каждого из случайно выбранных признаков, а среди них выбирается то, которое даёт лучший локальный прирост качества.

Этот уровень случайности значительно уменьшает корреляцию между отдельными деревьями, что, в свою очередь, снижает дисперсию ансамбля при агрегации их предсказаний (по принципу голосования для классификации или усреднения для регрессии).

2.5 AdaBoost

Алгоритм AdaBoost (Adaptive Boosting) относится к семейству методов бустинга — способов создания сильного классификатора путём последовательного объединения множества слабых классификаторов. В отличие от бэггинга, где все базовые модели обучаются независимо на разных подвыборках данных, бустинг строит модели последовательно, стараясь

исправить ошибки предыдущих этапов. Основная идея AdaBoost – адаптивно изменять веса обучающих образцов так, чтобы в последующих итерациях модель фокусировалась на наиболее сложных для классификации объектах.

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/дм³): остаточное содержание яблочной кислоты после ферментации.
3. Ash (зола, г/дм³): количество минеральных веществ, оставшихся после сжигания пробы.
4. Alkalinity of ash (щелочность золы): показатель щелочности зольных составляющих.
5. Magnesium (магний, мг/дм³): концентрация ионов магния.
6. Total phenols (общие фенолы, г/дм³): 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/дм³): 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.

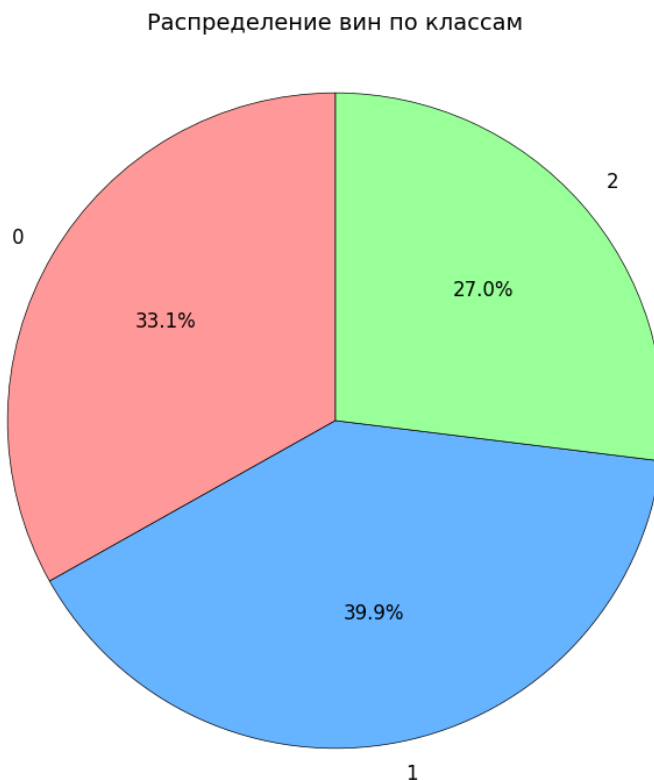


Рисунок 3.2.2 – Распределение вин по классам

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.

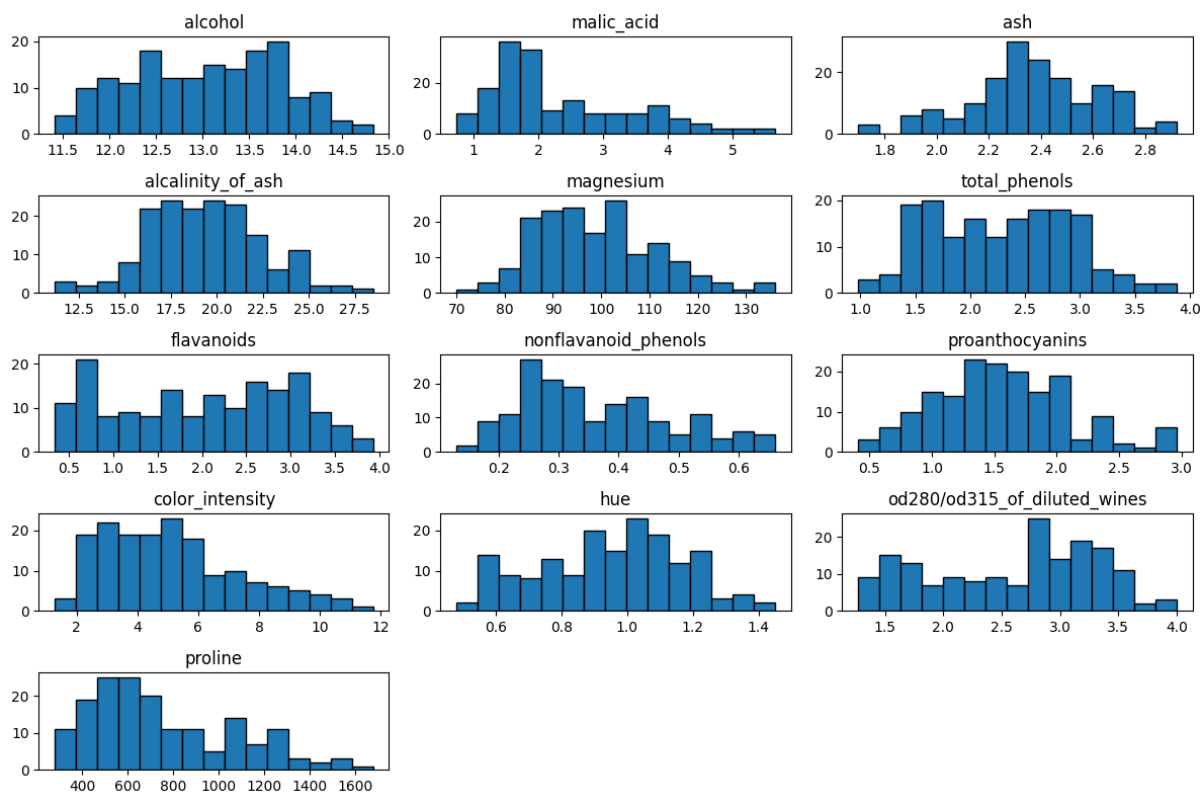


Рисунок 3.2.3 - Гистограммы распределений признаков

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.

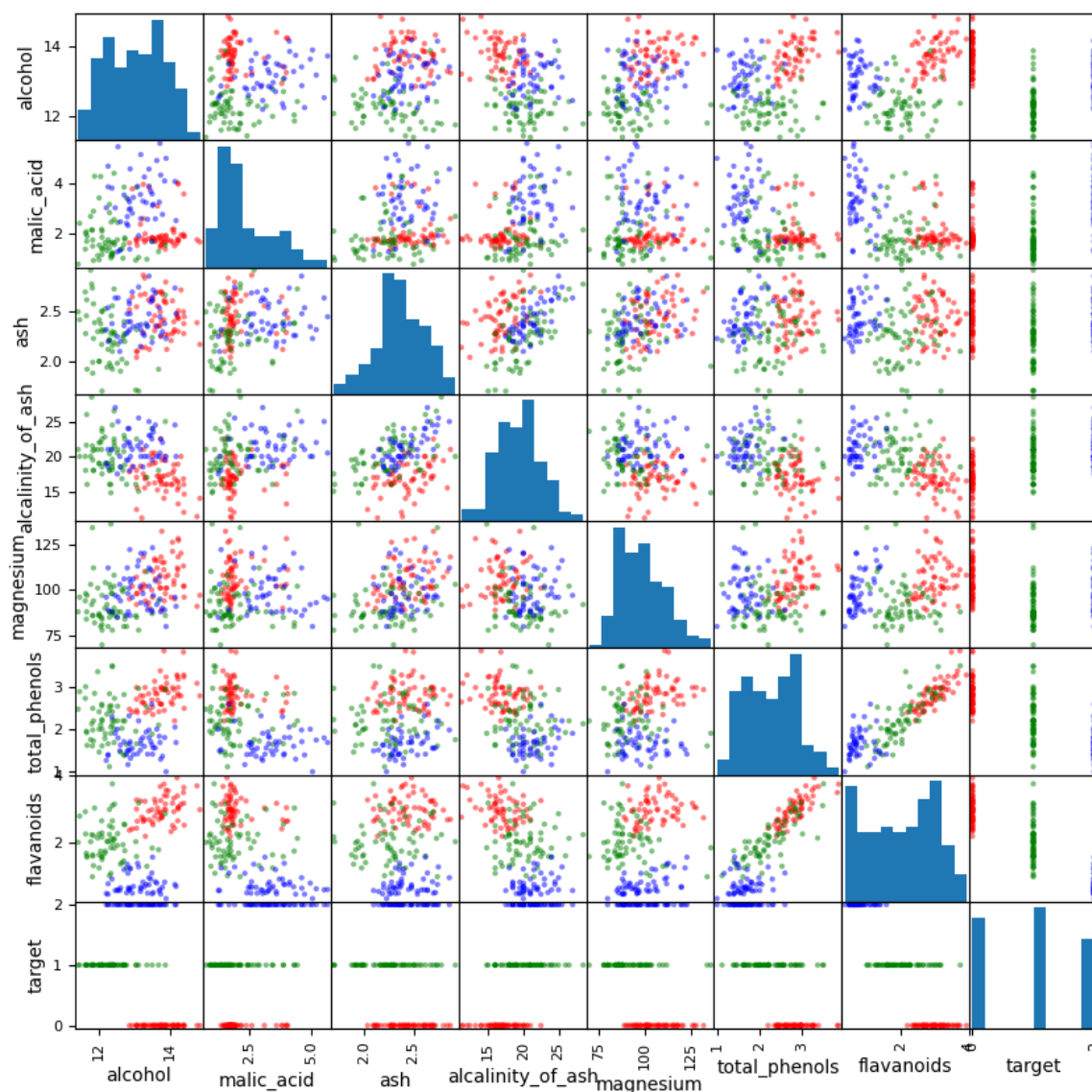


Рисунок 3.2.4 – Матрица рассеяния для первых семи признаков

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.

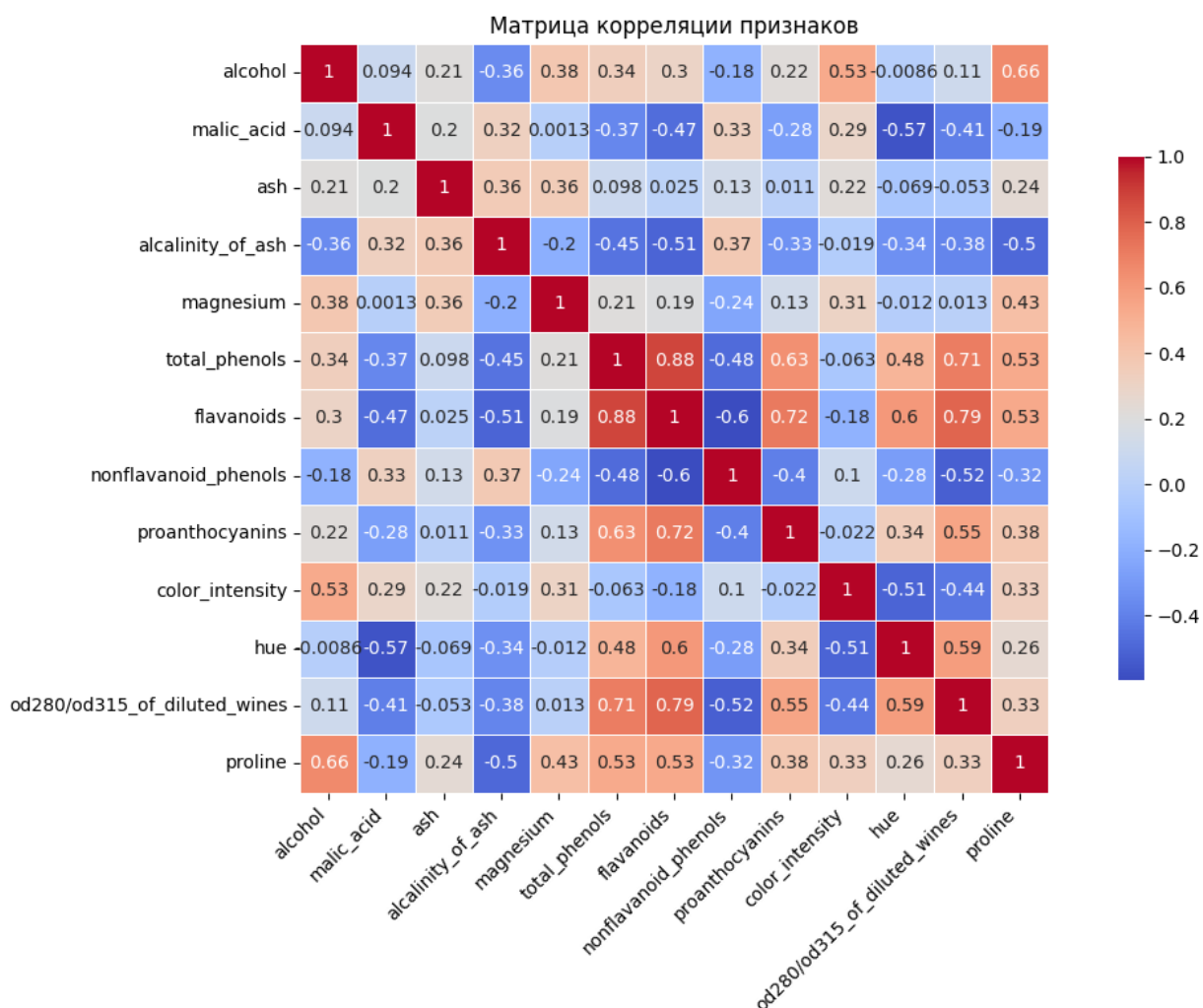


Рисунок 3.2.5 – Матрица корреляции признаков

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

$$z_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i}, \quad (3.3.1)$$

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 x 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 Ансамбль с голосованием

Написана собственная реализация ансамбля, в котором три различных базовых классификатора объединяются по принципу голосования (voting). В качестве базовых моделей используются:

- KNN (k-ближайших соседей);
- Decision Tree (дерево решений);
- Random Forest (случайный лес).

Ансамбль поддерживает функционирование в двух режимах голосования: hard voting (по большинству голосов) и soft voting (по усреднённым вероятностям). Код самописного ансамбля написан в файле `custom_ensemble.py`, содержание которого представлено в Приложении Б.

После обучения модели проведена оценка качества классификации с использованием основных метрик (Таблица 4.1.1).

Таблица 4.1.1 – Метрики по результатам классификации с собственной реализацией ансамбля моделей

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

Матрица ошибок отображена на Рисунке 4.1.1.

```
print("\nМатрица ошибок:")
print(voting.calculate_confusion_matrix(y_test, y_pred))

✓ 0.0s

Матрица ошибок:
[[12  0  0]
 [ 1 11  1]
 [ 0  0  9]]
```

Рисунок 4.1.1 – Матрица ошибок в результате классификации с собственной реализацией ансамбля моделей

Ансамбль допустил ошибку в небольшой доле случаев, во втором классе, объектов которого представлено меньше всего.

4.2 Случайный лес

Реализована модель классификации на основе алгоритма случайного леса с использованием библиотеки `sklearn`.

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

Для построения случайного леса использовались следующие параметры:

- количество деревьев (`n_estimators`) – 5;
- 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 4.2.1).

Таблица 4.2.1 – Метрики по результатам классификации с библиотечной реализацией `Random Forest`

Метрика	Значение	Интерпретация
accuracy	0.9412	Модель правильно классифицировала 94,12% объектов тестовой выборки

Продолжение Таблицы 4.2.1

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

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 и полностью совпал с результатом классификации собственным ансамблем моделей.

Дополнительно выполнена оценка важности признаков с использованием встроенной функции `feature_importances_`. Распределение важности представлено в Таблице 4.2.2.

Таблица 4.2.2 – 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.

4.3 Лес экстремально случайных деревьев

Реализована модель классификации на основе алгоритма экстремально случайных деревьев (ExtraTrees) с использованием библиотеки sklearn. Модель разработана в файле ExtraTrees.py, содержание которого представлено в Приложении Г.

Заданы следующие параметры модели:

- количество деревьев (n_estimators): 5;
- критерий разбиения: индекс Джини (gini);
- максимальная глубина деревьев: 5;
- 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 лесом экстремально случайных деревьев

Метрика	Значение	Интерпретация
accuracy	1	Модель правильно классифицировала 100% объектов тестовой выборки
precision	1	В среднем 100% объектов, отнесённых моделью к какому-либо классу, действительно принадлежат этому классу
recall	1	Модель нашла в среднем 100% объектов, которые действительно принадлежат каждому из классов
f1	1	Гармоническое среднее между точностью и полнотой

Модель продемонстрировала идеальную точность, что говорит о том, что данная модель бэггинга более эффективна, чем простой случайный лес.

4.4 AdaBoost

Для ансамбля AdaBoost в качестве «слабого» классификатора (weak learner) используется дерево решений глубины 1 (stump). Его можно явно передать в параметр `base_estimator`, но если в конструкторе `base_estimator=None`, то внутри автоматически создаётся пень решений. Реализация ансамбля AdaBoost написана в файле `AdaBoost.py`, содержание которого представлено в Приложении Д.

Заданы следующие гиперпараметры:

- количество итераций (`n_estimators`): 50;
- коэффициент обучения (`learning_rate`): 0.5;
- базовый классификатор: `DecisionTreeClassifier(max_depth=1)`.

После обучения модели проведена оценка качества классификации с использованием основных метрик (Таблица 4.4.1).

Таблица 4.4.1 – Метрики по результатам классификации с алгоритмом AdaBoost

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

Матрица ошибок представлена на Рисунке 4.4.1.

```
print("\nМатрица ошибок:")
print(ada.calculate_confusion_matrix(y_test, y_pred))
```

[51] ✓ 0.0s

...

Матрица ошибок:

```
[[12  0  0]
 [ 1 11  1]
 [ 0  0  9]]
```

Рисунок 4.4.1 - Матрица ошибок в результате классификации ансамблем AdaBoost

Модель показывает точность, сопоставимую со случайным лесом и собственным ансамблем моделей.

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были исследованы и реализованы ключевые ансамблевые методы машинного обучения для задачи классификации сортов вин на основе их физико-химических характеристик. Голосующий ансамбль (Voting Ensemble) продемонстрировал точность 94,12 % и показал сбалансированные результаты по метрикам precision, recall и F1-мере. Его основное преимущество заключается в комбинировании разнородных моделей (KNN, дерево решений и случайный лес), что помогает снизить риск переобучения. Однако ошибки отдельных базовых моделей частично накладываются друг на друга и ограничивают рост точности, поскольку группы слабых моделей, имеющие схожие ошибки, не компенсируют друг друга полностью.

Случайный лес (Random Forest) тоже показал точность 94,12 %. Этот метод продемонстрировал высокую стабильность и интерпретируемость, поскольку удалось выделить ключевые признаки (flavanoids, color_intensity, proline). Устойчивость к шуму достигается за счёт бэггинга и случайного выбора признаков при построении каждого дерева. При этом в эксперименте использовалось всего 5 деревьев ($n_estimators = 5$), что ограничило потенциал алгоритма, однако и в таком «легковесном» варианте Random Forest сохранил высокую обобщающую способность. При увеличении числа деревьев и глубины модель могла бы выявить ещё более тонкие закономерности, но риск переобучения при малом объёме данных остаётся велик.

Лес экстремально случайных деревьев (Extra Trees) оказался наиболее эффективным и достиг 100 % точности. Это объясняется максимальной стохастизацией при построении каждого дерева: не только выбираются случайные подмножества признаков, но и сами пороги разбиения выбираются случайно. Такая агрессивная рандомизация существенно снижает корреляцию между деревьями и, как следствие, дисперсию ансамбля. В результате модели справились с классификацией без ошибок, выявив сложные зависимости в

данных. Ключевые гиперпараметры (`max_depth = 5`, `max_features = "sqrt"`) оказались близкими к оптимальным: глубина обеспечивала баланс между выразительностью и обобщающей способностью, а квадратный корень из числа признаков снижал риск переобучения.

AdaBoost продемонстрировал точность 94,12 %, сопоставимую со случайным лесом. Алгоритм фокусируется на «сложных» объектах, постепенно адаптируя веса обучающих примеров: на каждом шаге он смещает акцент на те наблюдения, которые предыдущие слабые классификаторы предсказывали неправильно.

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 К-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).
6. Алгоритм AdaBoost [Электронный ресурс]: Habr. URL: <https://habr.com/ru/companies/otus/articles/503888/> (Дата обращения: 24.05.2025).
7. Алгоритмы AdaBoost (SAMME & R2). Принцип работы и реализация с нуля на Python [Электронный ресурс]: Habr. URL: <https://habr.com/ru/articles/800499/> (Дата обращения: 25.05.2025).

ПРИЛОЖЕНИЯ

Приложение А — Файл `dataset_manager.py` для предобработки и анализа датасета.

Приложение Б — Файл `custom_ensemble.py` с объединением моделей по принципу голосования.

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

Приложение Г — Файл `ExtraTrees.py` с использованием готовой реализации модели Extremely randomized trees.

Приложение Д — Файл `AdaBoost.py` с использованием готовой реализации ансамбля AdaBoost.

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")
```

```

        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))
```



```

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()).
    """

```

```
"""
    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,
```

```

        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 `custom_ensemble.py` с объединением моделей по принципу голосования

Листинг Б – Содержание файла `custom_ensemble.py`

```
import numpy as np
import pandas as pd
from typing import Union, Optional, Dict, List, Tuple
from sklearn.base import ClassifierMixin, clone
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix
from dataset_manager import DatasetManager
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

class VotingEnsemble:
    def __init__(
        self,
        estimators: List[Tuple[str, ClassifierMixin]],
        voting: str = "hard",
        weights: Optional[List[float]] = None,
    ) -> None:
        """
        Простая реализация ансамбля на основе голосования.

        Параметры:
            estimators (List[Tuple[str, ClassifierMixin]]):
                Список кортежей вида (имя модели, модель),
                где модель – объект, реализующий fit() и predict().
            voting (str): Тип голосования:
                - 'hard': большинство голосов (по умолчанию);
                - 'soft': усреднение предсказанных вероятностей (требуется, чтобы
                базовые модели поддерживали predict_proba()).
            weights (Optional[List[float]]): Список весов для моделей при
            голосовании.

                Если None, все модели считаются равнозначными. Длина списка
                должна совпадать с числом моделей.
        """
        if voting not in ("hard", "soft"):
            raise ValueError("Параметр voting должен быть 'hard' или 'soft'.")
        if weights is not None and len(weights) != len(estimators):
            raise ValueError("Длина weights должна совпадать с числом
            estimators.")

        self.estimators = estimators
        self.voting = voting
        self.weights = weights
        self.classes_: Optional[np.ndarray] = None
        self.fitted_estimators: List[ClassifierMixin] = []

    def fit(
        self,
        X_train: Union[pd.DataFrame, np.ndarray],
        y_train: Union[pd.Series, np.ndarray],
    ) -> None:
        """
        Обучает каждый базовый классификатор на переданных данных.

        Параметры:
            X_train (DataFrame | ndarray): Матрица признаков обучающей выборки.
```

```

        y_train (Series | ndarray): Вектор меток классов обучающей выборки.
    """
    X = np.array(X_train)
    y = np.array(y_train)
    self.classes_ = np.unique(y)
    self.fitted_estimators = []

    for _, estimator in self.estimators:
        model = clone(estimator)
        model.fit(X, y)
        self.fitted_estimators.append(model)

def predict(self, X_test: Union[pd.DataFrame, np.ndarray]) -> np.ndarray:
    """
    Предсказывает метки классов для тестовых данных ансамблем.

    Параметры:
        X_test (DataFrame | ndarray): Матрица признаков тестовой выборки.

    Возвращает:
        ndarray: Вектор предсказанных меток классов.

    Исключения:
        RuntimeError: если ансамбль не был обучен (нет fitted_estimators).
    """
    if not self.fitted_estimators:
        raise RuntimeError("Модели не обучены. Вызовите метод fit() перед predict().")

    X = np.array(X_test)
    n_samples = X.shape[0]
    n_models = len(self.fitted_estimators)

    if self.voting == "hard":
        all_preds = np.zeros((n_models, n_samples), dtype=object)
        for idx, model in enumerate(self.fitted_estimators):
            all_preds[idx] = model.predict(X)

        predictions = []
        for j in range(n_samples):
            votes = {}
            for i in range(n_models):
                label = all_preds[i, j]
                weight = self.weights[i] if self.weights is not None else 1.0
                votes[label] = votes.get(label, 0.0) + weight
            predicted_label = max(votes.items(), key=lambda x: x[1])[0]
            predictions.append(predicted_label)

        return np.array(predictions)

    else:
        probas = []
        for model in self.fitted_estimators:
            if not hasattr(model, "predict_proba"):
                raise RuntimeError(f"Модель {model} не поддерживает predict_proba(), невозможно выполнить soft-голосование.")
            probas.append(model.predict_proba(X))

        avg_proba = np.zeros_like(probas[0])
        for idx, proba in enumerate(probas):

```

Продолжение Листинга Б

```
        w = self.weights[idx] if self.weights is not None else 1.0
        avg_proba += w * proba
    avg_proba /= (sum(self.weights) if self.weights is not None else
n_models)

    return np.array([self.classes_[np.argmax(row)] for row in
avg_proba])

    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 ∈ [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.

    Параметры:
        y_true (ndarray): Истинные метки.
        y_pred (ndarray): Предсказанные метки.
        average (str): Стратегия 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 основным метрикам классификации.

        Параметры:
            y_true (ndarray): Истинные метки.
            y_pred (ndarray): Предсказанные метки.
            average (str): Стратегия усреднения ('macro', 'micro', 'weighted').

        Возвращает:
            Dict[str, float]: Словарь с метриками:
                - 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 = ("knn", KNeighborsClassifier(n_neighbors=3))
    dt = ("decision_tree", DecisionTreeClassifier(criterion="gini",
max_depth=None, random_state=42))
    rf = ("random_forest", RandomForestClassifier(n_estimators=5,
random_state=42))
```

Окончание Листинга Б

```
voting = VotingEnsemble(  
    estimators=[knn, dt, rf],  
    voting="hard",  
    weights=[1.0, 1.0, 1.0]  
)  
voting.fit(X_train, y_train)  
  
y_pred = voting.predict(X_test)  
  
report = voting.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(voting.calculate_confusion_matrix(y_test, y_pred))
```

Приложение В

Файл RandomForest.py с использованием готовой реализации модели Random Forest

Листинг В – Содержание файла 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 В

```
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 В

```
        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 В

```
        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 В

```
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_]
)
```

Приложение Г

Файл ExtraTrees.py с использованием готовой реализации модели Extremely randomized trees

Листинг Г – Содержание файла ExtraTrees.py

```
import numpy as np
import pandas as pd
from typing import Union, Optional, Dict, List
from sklearn.ensemble import ExtraTreesClassifier
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 ExtraTreesModel:
    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 экстра-деревьев (Extra Trees).

        Параметры:
            n_estimators (int): Количество деревьев в ансамбле.
            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[ExtraTreesClassifier] = 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 экстра-деревьев по предоставленным обучающим данным.

        Параметры:
            X_train (DataFrame | ndarray): Матрица признаков обучающей выборки.
            y_train (Series | ndarray): Вектор истинных меток классов.
        """
```



```
self.model = ExtraTreesClassifier(
    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 экстра-деревьев.

    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} экстра-деревя")
    plt.show()

def get_feature_importance(self) -> pd.DataFrame:
    """
    Возвращает важность признаков в обученной модели.
```

Продолжение Листинга Г

```
        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.
```

```

        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()

    et = ExtraTreesModel(
        n_estimators=5,
        criterion='gini',
        max_depth=5,
        max_features='sqrt',
        random_state=42
    )
    et.fit(X_train, y_train)

```

```
y_pred = et.predict(X_test)
report = et.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(et.calculate_confusion_matrix(y_test, y_pred))

print("\nВажность признаков:")
print(et.get_feature_importance().to_string(index=False))

et.plot_tree(
    tree_idx=0,
    feature_names=X_train.columns.tolist(),
    class_names=[str(cls) for cls in et.classes_]
)
```

Приложение Д

Файл AdaBoost.py с использованием готовой реализации ансамбля AdaBoost

Листинг Д – Содержание файла AdaBoost.py

```
import numpy as np
import pandas as pd
from typing import Union, Optional, Dict, List
from sklearn.ensemble import AdaBoostClassifier
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 AdaBoostModel:
    def __init__(
        self,
        n_estimators: int = 50,
        learning_rate: float = 1.0,
        base_estimator: Optional[DecisionTreeClassifier] = None,
        random_state: int = 42,
    ) -> None:
        """
        Инициализирует классификатор AdaBoost.

        Параметры:
            n_estimators (int): Количество слабых моделей (итераций) в ансамбле.
            learning_rate (float): Коэффициент снижения влияния каждой слабой
            модели.
            base_estimator (DecisionTreeClassifier, optional): Базовый алгоритм
            (слабый классификатор).
            Если None, по умолчанию используется дерево решений глубины 1
            (stump).
            random_state (int): Начальное значение генератора случайных чисел
            для воспроизводимости.
        """
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.base_estimator = (
            base_estimator
            if base_estimator is not None
            else DecisionTreeClassifier(max_depth=1, random_state=random_state)
        )
        self.random_state = random_state
        self.model: Optional[AdaBoostClassifier] = 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 AdaBoost по предоставленным обучающим данным.
```

```

    Параметры:
        X_train (DataFrame | ndarray): Матрица признаков обучающей выборки.
        y_train (Series | ndarray): Вектор истинных меток классов.
    """
    self.model = AdaBoostClassifier(
        estimator=self.base_estimator,
        n_estimators=self.n_estimators,
        learning_rate=self.learning_rate,
        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_stage(
    self,
    stage_idx: int = 0,
    feature_names: Optional[List[str]] = None,
    class_names: Optional[List[str]] = None,
) -> None:
    """
    Визуализирует структуру одного из слабых деревьев ансамбля AdaBoost.

    Параметры:
        stage_idx (int): Индекс слабого классификатора для отображения (0 ≤
idx < n_estimators).
        feature_names (list, optional): Список имён признаков (если
доступно).
        class_names (list, optional): Список имён классов (если доступно).
    """
    if self.model is None:
        raise RuntimeError("Сначала обучите модель.")
    if stage_idx < 0 or stage_idx >= len(self.model.estimators_):
        raise IndexError(f"stage_idx должно быть от 0 до
{len(self.model.estimators_) - 1}.")
    plt.figure(figsize=(16, 10))
    plot_tree(
        self.model.estimators_[stage_idx],
        filled=True,
        feature_names=feature_names,
        class_names=class_names,
    )
    plt.title(f"AdaBoost: дерево №{stage_idx}")
    plt.show()

```

```
def get_feature_importance(self) -> pd.DataFrame:
    """
    Возвращает важность признаков, рассчитанную ансамблем AdaBoost.

    Возвращает:
        DataFrame: Таблица с признаками и их важностью, отсортированная по
        убыванию.
    """
    if self.model is None:
        raise RuntimeError("Сначала обучите модель.")
    importances = self.model.feature_importances_
    return (
        pd.DataFrame(
            {"feature": self.model.feature_names_in_, "importance":
importances}
        )
        .sort_values("importance", ascending=False)
        .reset_index(drop=True)
    )

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.

    Параметры:
        y_true (ndarray): Истинные метки.
        y_pred (ndarray): Предсказанные метки.
        average (str): Стратегия 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()
```


Окончание Листинга Д

```
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()

ada = AdaBoostModel(
    n_estimators=50,
    learning_rate=0.5,
    base_estimator=DecisionTreeClassifier(max_depth=1, random_state=42),
    random_state=42,
)
ada.fit(X_train, y_train)

y_pred = ada.predict(X_test)

report = ada.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(ada.calculate_confusion_matrix(y_test, y_pred))

print("\nВажность признаков:")
print(ada.get_feature_importance().to_string(index=False))

ada.plot_stage(
    stage_idx=0,
    feature_names=X_train.columns.tolist(),
    class_names=[str(cls) for cls in ada.classes_],
)
```