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

Институт Информационных Технологий
Кафедра Вычислительной техники

ПРАКТИЧЕСКИЕ РАБОТЫ №2, 3, 6

по дисциплине
«Проектирование интеллектуальных систем (Часть 1/2)»

Студент группы ИКБО-04-22 — Егоров Л.А.
(Ф.И.О. студента)

Руководитель работы — Холмогоров В.В.
(Ф.И.О. преподавателя)

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сервисы для потокового воспроизведения музыки являются самыми используемыми вариантами для прослушивания аудио, обходя по востребованности физические носители. Одним из самых явных удобств данных сервисов является возможность подбора песен по набору ранее прослушанных, а также жанровые рекомендации. Поскольку алгоритмы, используемые для данных возможностей, не доступны в открытом доступе, принято решение самостоятельно провести анализ музыкальных композиций.

Для анализа нужно выполнить две задачи:

– кластеризация — определение, к какой группе композиций можно отнести новую. Применение: предложение новых песен для уже существующего плейлиста;
– классификация — для определения жанровой принадлежности композиции.

Самым распространённым алгоритмом кластеризации является KMeans, однако своё применение имеет и алгоритм кластеризации, основанный на плотности распределения точек (DBSCAN), также как и ансамбль алгоритмов кластеризации.

Самым простым алгоритмом классификации является KNN, более эффективными являются решающие деревья и их объединения в ансамбли (в том числе, в случайный лес).

1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1.1 Кластеризация

1.1.1 KMeans

Алгоритм минимизирует сумму внутрикластерных расстояний (1.1):

$$V = \sum_{i=1}^{m} \sum_{j=1}^{n} \rho(x, \mu_j) \longrightarrow \min \tag{1.1}$$

где m — количество данных во входной выборке;

n — количество кластеров;

μ_j — центроид j-го кластера;

ρ — мера расстояния между точками. Обычно выбирается Евклидово расстояние, однако можно применять и другие метрики для получения отличающихся результатов.

1.1.2 DBSCAN

Алгоритм разделяет точки на группы, которые лежат друг от друга на большом расстоянии. Поэтому алгоритм сам способен определить количество кластеров, а также выделить шумовые точки на основе того, что они будут далеко находиться от других точек.

1.1.3 Ансамбль алгоритмов кластеризации

Алгоритм построения ансамбля:

1. Алгоритм KMeans вызывается нечётное количество раз, для каждого вызова используются разные параметры и/или разные метрики.
2. Для каждого из алгоритмов определяется его точность. Для этого может использоваться индекс Rand (Формула 1.4) или число, обратное компактности кластеров (Формула 1.5). В первом случае необходимо иметь метки реальных классов, для второго метода это не требуется.

3. На основе рассчитанной точности определяются веса каждого из алгоритмов (1.2):

$$\omega_l = \frac{\mathrm{Acc}_l}{\sum_{l=1}^{L} \mathrm{Acc}_l} \tag{1.2}$$

где L — количество алгоритмов в ансамбле;

Acc_l — точность l-го алгоритма.

4. Для каждого алгоритма составляется матрица сходства/различий (1.3):

$$H = \{h_{ij}\},$$

$$h_{ij} = \begin{cases} 0\text{ , если элементы i и j попали в один кластер} \\ 1\text{ , если элементы i и j попали в разные кластеры} \end{cases} \tag{1.3}$$

5. Все полученные матрицы складываются друг с другом с учётом полученных весов.
6. Итоговая матрица подаётся на вход иерархической агломеративной кластеризации.

1.1.4 Метрики кластеризации

В качестве метрик кластеризации могут использоваться:

1. Индекс Rand:

$$\mathrm{Rand} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}} \tag{1.4}$$

где TP — количество пар, где элементы принадлежат одному кластеру и одному классу;

FP — количество пар, где элементы принадлежат одному кластеру, но разным классам;

FN — количество пар, где элементы принадлежат разным кластерам, но одному классу;

FN — количество пар, где элементы принадлежат разным кластерам и разным классам.

2. Компактность кластеров:

$$\mathrm{WSS} = \sum_{i=1}^{M} \sum_{j=1}^{|C_j|} \left(x_{ij} - \overline{x_j}\right) \tag{1.5}$$

где M — количество кластеров.

1.2 Классификация

1.2.1 KNN

Алгоритм заключается в определении класса для записи на основе информации о ближайших соседях. Как правило, используется либо наиболее часто встречающийся класс среди соседей, делается взвешенное голосование, где веса равны обратному расстоянию до соседей.

1.2.2 Дерево решений

Процесс построения дерева состоит из следующих этапов:

1. Для каждого узла выбирается признак и порог таким образом, чтобы максимизировать прирост информации. Прирост информации считается на основе индекса Джини (1.6) или энтропии (1.7):

$$I_G = 1 - \sum_{i=1}^{J} p_i^2 \tag{1.6}$$

$$\mathrm{Entropy} = -\sum_{i=1}^{J} (p_i \log_2 p_i) \tag{1.7}$$

1.2.3 Бэггинг

Обучаются несколько моделей (как правило, решающих деревьев), для каждой из моделей формируется бутстрэп-выборка, т.е. берутся данные случайным образом с повторением. Модели обучаются независимо, а предсказание ансамбля определяется на основе, например, голосования.

1.2.4 Случайный лес

Является подвидом бэггинга, где при построении каждого узла выбирается случайный набор признаков, который может использоваться для разделения.

1.2.5 Метрики классификации

Используемые метрики классификации:

1. Аккуратность:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}} \tag{1.8}$$

2. Точность:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1.9}$$

3. Полнота:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1.10}$$

4. F1-мера:

$$\text{f1-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{1.11}$$

Также существуют micro-f1. macro-f1 и weighted-f1 для задач многоклассовой классификации.

1.3 Расстояния

Используемые расстояния между объектами:

1. Евклидово расстояние:

$$\rho(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1.12}$$

2. Манхэттенское расстояние:

$$\rho(x, y) = \sum_{i=1}^{n} |x_i - y_i| \tag{1.13}$$

3. Расстояние Чебышева:

$$\rho(x, y) = \max_{i=1}^{n} |x_i - y_i| \tag{1.14}$$

4. Коэффициент Жаккара:

$$K(x, y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 - \sum_{i=1}^{n} x_i y_i} \tag{1.15}$$

1.4 Описание данных

Для решения поставленных задач выбран набор данных с платформы Kaggle. Структура данных приведена в Таблице 1.1.

Таблица 1.1 — Описание полей

Название поля	**Описание**
track_id	ID композиции из Spotify
artists	Исполнители композиции
album_name	Название альбома
track_name	Название композиции
popularity	Популярность композиции (от 0 до 100)
duration_ms	Длительность композиции в миллисекундах
explicit	Есть ли нецензурная лексика
danceability	Насколько композиция является танцевальной
energy	Энергичность
key	Тональность
loudness	Громкость
mode	Мажор/минор
speechiness	Насколько много проговоренных слов (по отношению к пропетым словам)
acousticness	Акустичность
instrumentalness	Насколько много в композиции инструментальности по отношению к вокалу
liveness	Наличие звуков аудитории в композиции
valence	Позитивность
tempo	Темп
time_signature	Размер
track_genre	Жанр (предсказываемая величина)

1.5 Предобработка данных

Для предобработки выполнено несколько этапов:

1. Удалены дубликаты - те записи, у которых совпадают одновременно список исполнителей и название композиции.
2. Проведён корреляционный анализ - для этого выведена матрица корреляции (Рисунок 1.1):

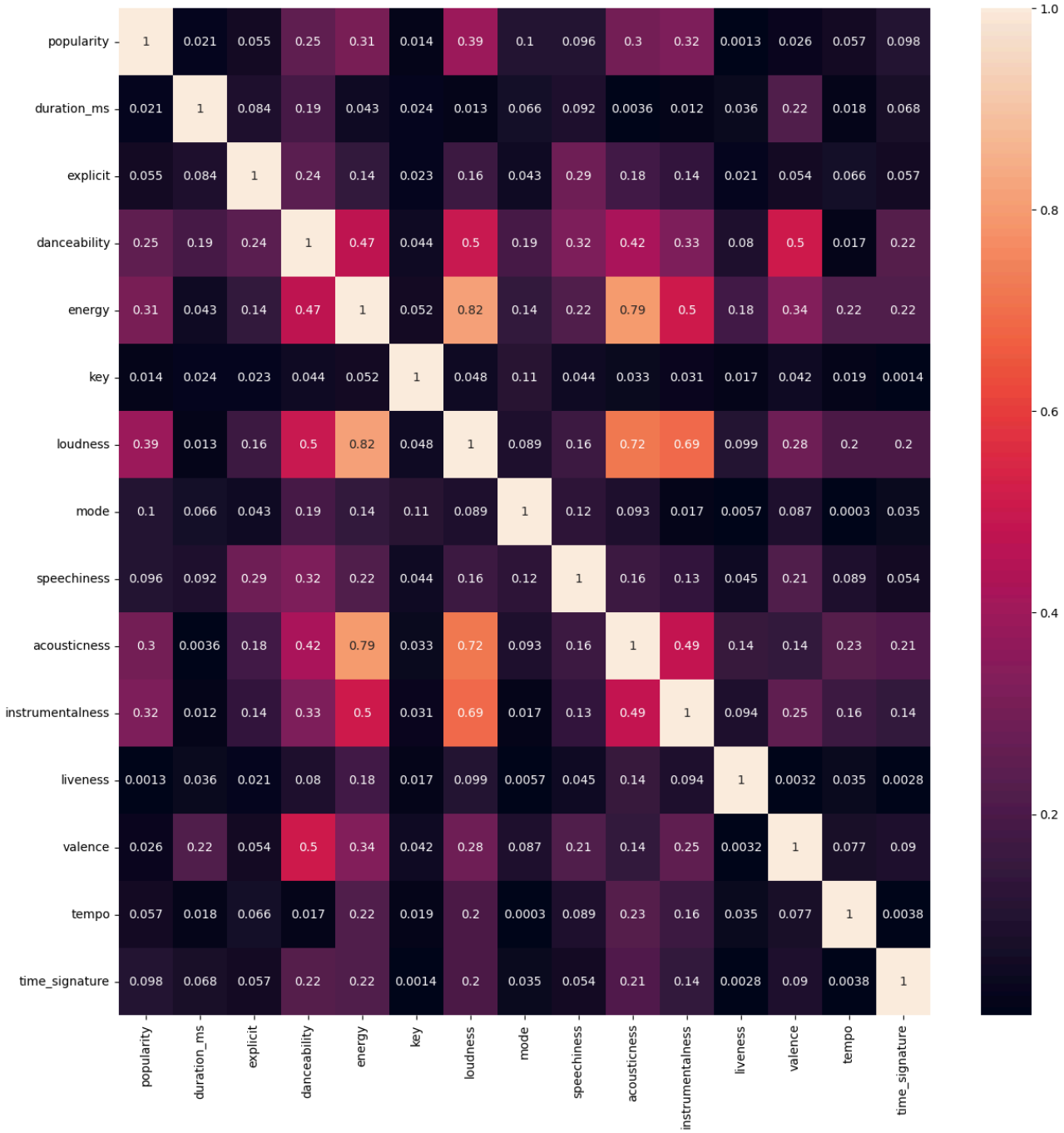

Рисунок 1.1 — Корреляционная матрица

На основе данной матрицы принято решение убрать поля loudness и energy.

3. Выполнена нормализация данных с помощью StandardScaler, который использует Формулу 1.16:

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}, i \in [1, n], j \in [1, m] \quad (1.16)$$

где n — количество объектов в выборке;

m — количество признаков;

μ — среднее значение среди всех объектов по одному признаку;

σ — стандартное отклонение всех объектов по одному признаку.

4. Дополнительно, для решения задачи классификации, проведена обработка датасета алгоритмом DBSCAN - с помощью алгоритма помечаются и удаляются шумовые точки.

1.6 Распределение данных

Для визуализации исходных данных применён метод главных компонент, суть которого заключается в спектральном разложении ковариационной матрицы. Визуализация исходных данных представлена на Рисунке 1.2

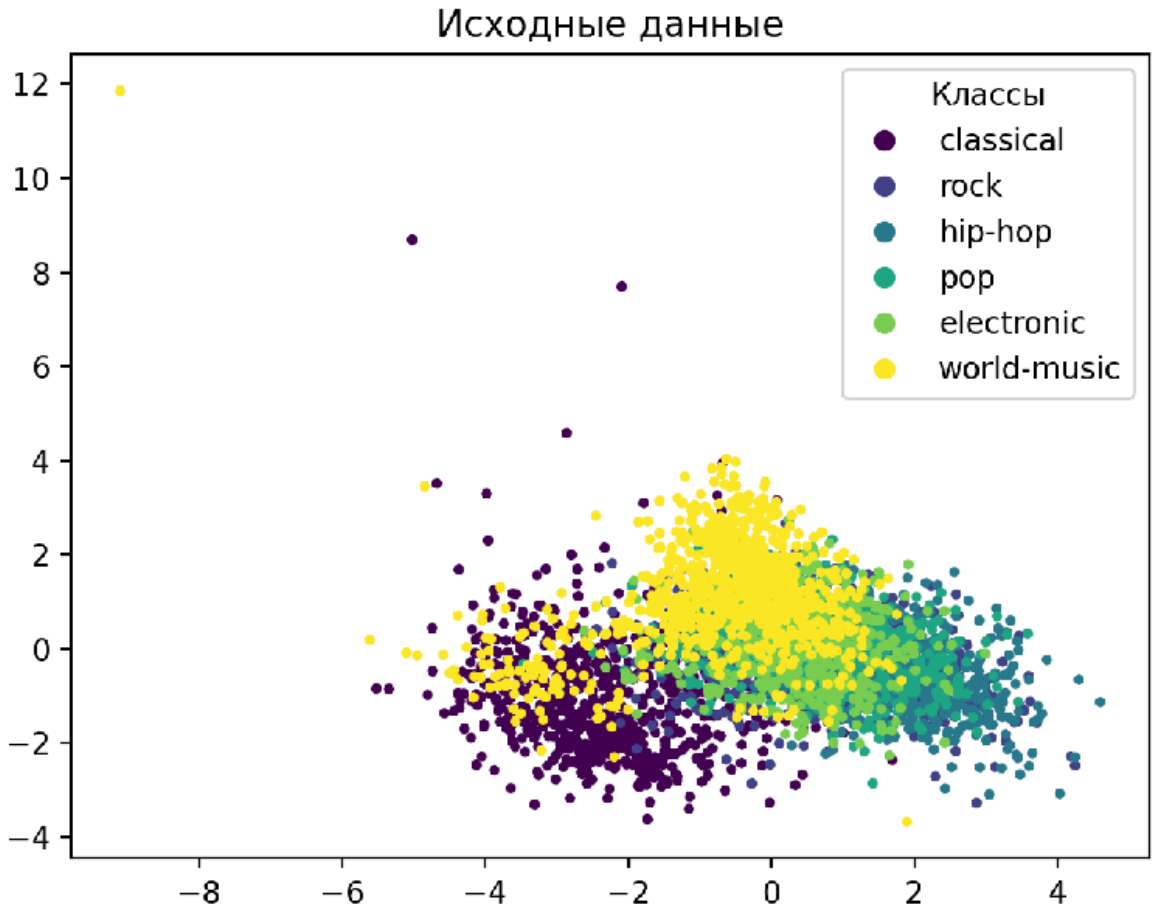

Рисунок 1.2 — Визуализация исходных данных

2 ПРАКТИЧЕСКИЙ РАЗДЕЛ

2.1 Описание программных сущностей

Для каждого алгоритма написаны классы, реализующие их. У каждого класса есть метод fit, который принимает входные данные и обучает модель на их основе. У алгоритмов классификации также есть метод predict, который для поданных векторов выдаёт предсказания классов. Специфичные методы для каждого из классов представлены ниже:

1. Общие функции:

– preprocessing — выполнение предобработки набора данных;

– visualize — отображение набора данных, преобразованного через метод главных компонент.

2. KMeans:

– _compute_distances — расчёт расстояний между точкой и каждыми центрами кластеров;

– _assign_clusters — расположение точки в кластер с ближайшим центром;

– rand_index — вычисление индекса Rand;

– cluster_cohesion — вычисление плотности кластера;

– cluster_similarity_matrix — расчёт матрицы сходства.

3. DBSCAN:

– _expand_cluster — расширение кластера на основе плотности;

– _get_neighbors — получение точек, входящих в окрестности данной точки;

– denoise — выдача индексов нешумовых точек.

4. ClusterEnsemble:

– _create_estimators — инициализация моделей KMeans со случайно выбранными расстояниями.

5. KNNClassifier: специфичные методы отсутствуют.

6. DecisionTreeClassifier:

– _grow_tree — построение дерева рекурсивным способом, от корня к листьям;

– _best_split — нахождение лучшего разбиения в узле, т.е. выбор лучшего признака с лучшим пороговым значением;

– _information_gain — нахождение прироста информации для текущего узла при выборе признака и порога;

– _gini_impurity — вычисление индекса Джини;

– _entropy — вычисление энтропии;

– _traverse_tree — проход по дереву до листьев, используя пороги.

7. Bagging — реализация бэггинга на основе DecisionTreeClassifier. Специфичные методы отсутствуют

8. RandomForestClassifier — библиотечная реализация случайного леса.

2.2 Тестирование

2.2.1 Кластеризация

В Таблицу 2.1 сведены данные о тестировании трёх алгоритмов кластеризации.

Таблица 2.1 — Тестирование алгоритмов кластеризации

Название алгоритма	**Специфичные параметры**	**Индекс Rand**	**Плотность кластеров**	**Время работы (мин:сек)**
KMeans	– количество кластеров = 6; – манхэттенское расстояние; – максимальное количество итераций: 300.	0.74478	40315.875	00:07
DBSCAN	– $\varepsilon = 2.1$; – min_samples = 8.	0.63134	33871.4727	01:55
ClusterEnsemble	– количество моделей = 13.	0.76585	30696.6308	02:42

Результат кластеризации алгоритмом DBSCAN представлен на Рисунке 2.1.

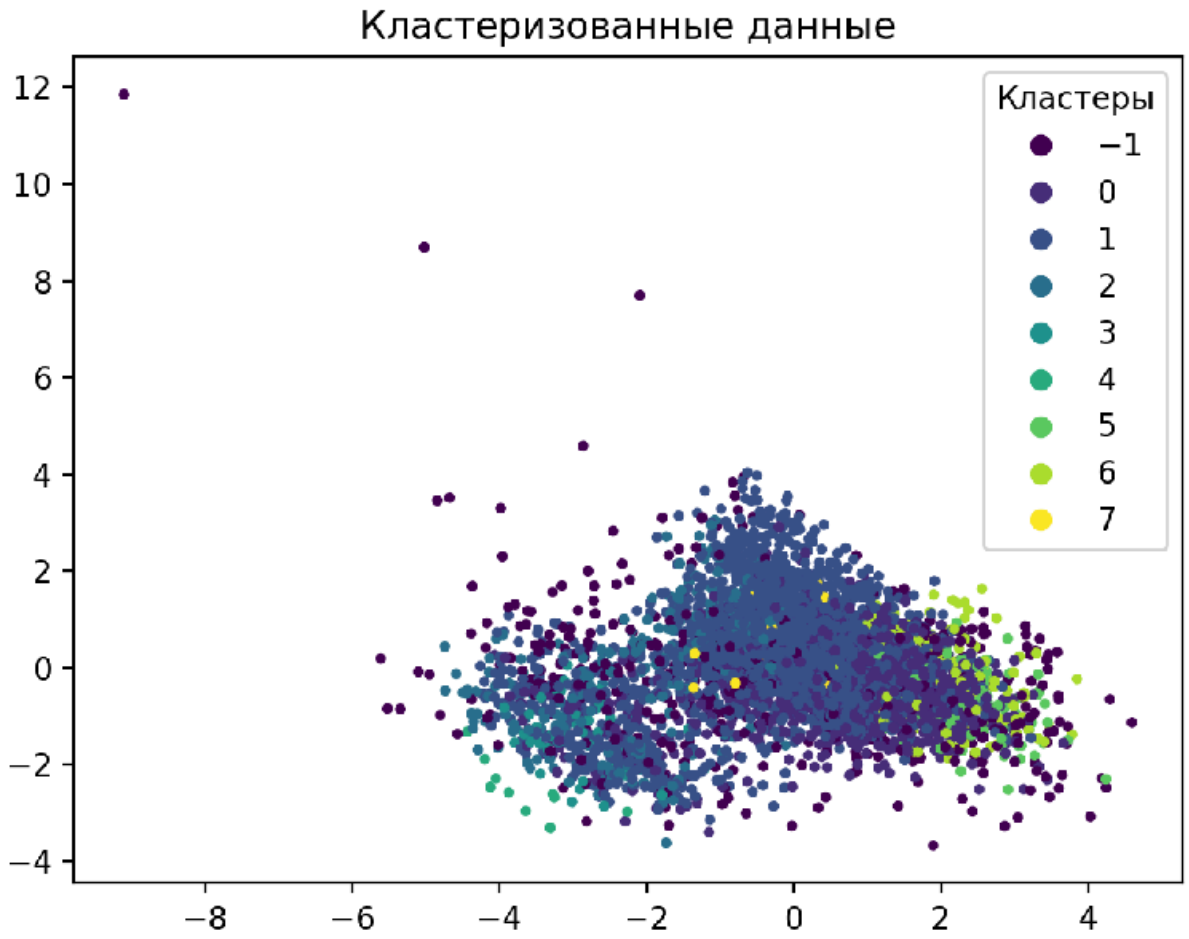

Рисунок 2.1 — Результат кластеризации DBSCAN

Для подбора оптимального количества моделей в ансамбле кластеризаторов проведено обучение нескольких ансамблей, с количеством моделей от 5 до 15 включительно (для каждого количества с учителем и без). График исследования представлен на Рисунке 2.2.

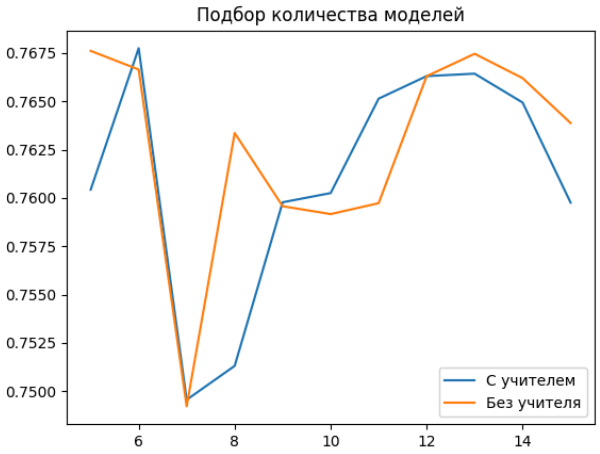

Рисунок 2.2 — Точность ансамбля в зависимости от количества моделей

Лучший результат обучения без учителя достигнут при количестве моделей, равном 13.

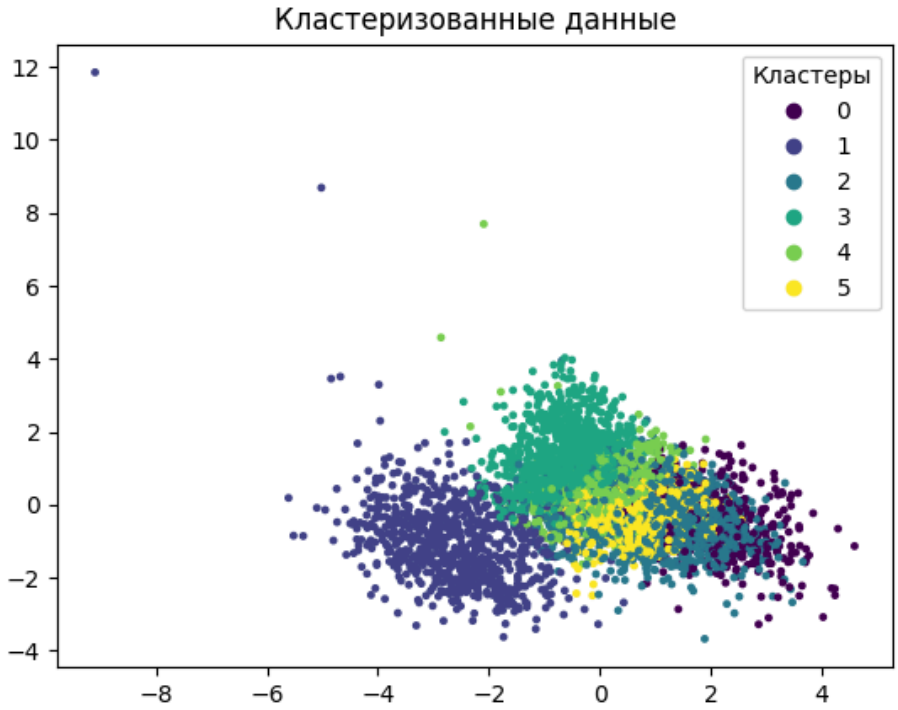

Рисунок 2.3 — Результат кластеризации ансамблем

2.2.2 Классификация

В Таблицу 2.2 сведены данные о тестировании трёх алгоритмов кластеризации.

Таблица 2.2 — Тестирование алгоритмов кластеризации

Название алгоритма	**Специфичные параметры**	**Accuracy**	**Macro-f1**	**Время обучения (мин:сек)**	**Время предсказания (мин:сек)**
KNN	– количество ближайших соседей = 5; – манхэттенское расстояние; – максимальное количество итераций: 300.	0.63	0.6	00:00	00:23
Дерево решений	– максимальная глубина дерева = 5; – минимальное количество	0.64	0.62	00:05	00:00.01

Продолжение Таблицы 2.2

	экземпляров в узле = 2; – критерий Джини.				
Бэггинг	– количество моделей = 50; – минимальное количество экземпляров в узле = 2; – максимальная глубина дерева = 5.	0.68	0.66	02:55	00:00.05
Случайный лес	– количество моделей = 100; – минимальное количество экземпляров в узле = 2; – максимальная глубина дерева = 10.	0.78	0.76	00:00.5	00:00.01

ЗАКЛЮЧЕНИЕ

Среди алгоритмов кластеризации алгоритм KMeans является идеальной серединой - работает быстрее всех и показывает хороший индекс Rand. Ансамбль моделей KMeans показывает большую точность, однако время его работы значительно дольше KMeans. DBSCAN показывает худшую точность, однако для него можно найти применение в удалении шумовых точек.

Среди алгоритмов классификации наименее точным оказался KNN, однако самым быстрым оказалось решающее дерево. Бэггинг и случайный лес показали улучшение результатов по сравнению с обычным решающим деревом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Разработка ансамбля алгоритмов кластеризации на основе изменяющихся метрик расстояний [Электронный ресурс]: Национальный исследовательский ядерный университет «МИФИ». URL: https://ceur-ws.org/Vol-1752/paper06.pdf.
2. Spotify Tracks Dataset [Электронный ресурс]: Kaggle. URL: https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset.

ПРИЛОЖЕНИЯ

Приложение А — Реализация алгоритма KMeans.

Приложение Б — Реализация алгоритма DBSCAN.

Приложение В — Реализация ансамбля кластеризации.

Приложение Г — Реализация алгоритма KNN.

Приложение Д — Реализация дерева решений.

Приложение Е — Реализация бэггинга.

Приложение Ж — Реализация случайного леса.

Приложение А

Реализация алгоритма KMeans

Продолжение Листинга А.1

```
from collections import defaultdict
from pathlib import Path
from typing import Literal, Callable

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import rand_score
from sklearn.decomposition import PCA
from tqdm import tqdm


CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']

METRICS = Literal["euclidean", "manhattan", "chebyshev", "jaccard"]
SEED = 78498
rng = np.random.default_rng(SEED)

class Distance:
    def __init__(self,
                 metric: METRICS = "euclidean"):
        self.metric = metric
        metrics: dict[METRICS, Callable[[np.ndarray, np.ndarray], np.float32]]
= {
            "euclidean": self._euclid,
            "manhattan": self._manhattan,
            "chebyshev": self._chebyshev,
            "jaccard": self._jaccard,
        }
        self._func = metrics[self.metric]

    @staticmethod
    def _euclid(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.linalg.norm(x - y)

    @staticmethod
    def _manhattan(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.sum(np.abs(x - y))

    @staticmethod
    def _chebyshev(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.max(np.abs(x - y))

    @staticmethod
    def _jaccard(x: np.ndarray, y: np.ndarray) -> np.float32:
      return np.dot(x, y) / (np.linalg.norm(x) + np.linalg.norm(y) - np.dot(x, y))

    def __call__(self, x: np.ndarray, y: np.ndarray) -> np.float32:
        return self._func(x, y)


class KMeans:
    def __init__(self,
                 data: np.ndarray,
                 n_clusters: int = 6,
                 max_iter: int = 300,
                 tol: float = 1e-4,
                 metric: METRICS = "euclidean"):
        self.X = data
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.labels_ = None
        self.cluster_centers_ = None
        self.metric = Distance(metric)
```

Продолжение Листинга А.1

```
    def _compute_distances(self, X, centers):
        distances = np.zeros((len(X), len(centers)))
        for i, point in enumerate(X):
            for j, center in enumerate(centers):
                distances[i, j] = self.metric(point, center)
        return distances

    def _assign_clusters(self, X, centers):
        distances = self._compute_distances(X, centers)
        labels = np.argmin(distances, axis=1)
        min_distances = np.min(distances, axis=1)
        return labels, min_distances

    def fit(self):

        random_indices = rng.permutation(len(self.X))[:self.n_clusters]
        self.cluster_centers_ = self.X[random_indices]

        for _ in tqdm(range(self.max_iter)):
            labels, _ = self._assign_clusters(self.X, self.cluster_centers_)

            new_centers = np.array([self.X[labels == i].mean(axis=0)
                                    for i in range(self.n_clusters)])

          if all(self.metric(old_center, new_center) < self.tol for (old_center,
new_center) in zip(self.cluster_centers_, new_centers)):
                break

            self.cluster_centers_ = new_centers

        self.labels_ = labels

    def rand_index(self, real_labels):
        return rand_score(real_labels, self.labels_)

    def cluster_cohesion(self):
        result = 0
        for i in range(self.n_clusters):
            center = self.cluster_centers_[i]
            for j in range(len(self.X)):
                if self.labels_[j] == i:
                    result += np.linalg.norm(self.X[j] - center) ** 2
        return result

    def cluster_similarity_matrix(self):
        n = len(self.labels_)
        matrix = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                if self.labels_[i] == self.labels_[j]:
                    matrix[i, j] = 1
        return matrix



def preprocessing(df: pd.DataFrame):
    df = df.loc[df.genre.isin(CLASSES)]
    df = df.drop_duplicates(subset=['artists', 'track_name']) \
           .reset_index(drop=True)
    df = df.drop(columns=df.columns[:5])

    df = df.drop(columns=['energy', 'loudness'])
    scaler = StandardScaler()
    df.iloc[:, :-1] = df.iloc[:, :-1].astype(float)
    df.iloc[:, :-1] = scaler.fit_transform(df.iloc[:, :-1])

    return df


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] | None = None):
    pca = PCA(n_components=2)
```

Окончание Листинга А.1

```
    transformed_data = pca.fit_transform(df)

       scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
          plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
       string_columns = ['track_id', 'artists', 'album_name', 'track_name',
'track_genre']
    for column in string_columns:
        column_types[column] = str

    df = pd.read_csv('dataset.csv', dtype=column_types)
    df = df.rename(columns={'track_genre': 'genre'})
    data = preprocessing(df)

    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
    # visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    kmeans = KMeans(data.drop(columns=['genre']).values, metric="manhattan")
    kmeans.fit()
    print("Метки кластеров:", kmeans.labels_)

     visualize(data.drop(columns=['genre']), kmeans.labels_, "Кластеризованные
данные")
    print(f'Значение Rand индекса: {kmeans.rand_index(track_labels)}')
    print(f'Плотност кластеров: {kmeans.cluster_cohesion()}')


if __name__ == '__main__':
    main()
```

Приложение Б

Реализация алгоритма DBSCAN

Продолжение Листинга Б.1

```
from collections import defaultdict
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import rand_score
from sklearn.decomposition import PCA
from tqdm import tqdm


CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']


class DBSCAN:
    def __init__(self,
                 data: pd.DataFrame,
                 epsilon: float = 2,
                 min_samples: int = 10,
                 seed: int = 78498):
        self.X = data.values
        self.labels_ = np.full(len(self.X), -1)
        self.visited = set()
        self.epsilon = epsilon
        self.min_samples = min_samples
        np.random.seed(seed)

    def fit(self, x: pd.DataFrame | None = None):
        if x is not None:
            self.X = np.vstack((self.X, x.values))
            self.labels_ = np.full(len(self.X), -1)

        cluster_id = 0

        for point_idx in tqdm(range(len(self.X))):
            if point_idx in self.visited:
                continue

            self.visited.add(point_idx)
            neighbors = self._get_neighbors(point_idx)

            if len(neighbors) < self.min_samples:
                self.labels_[point_idx] = -1
            else:
                self._expand_cluster(point_idx, neighbors, cluster_id)
                cluster_id += 1

    def _expand_cluster(self, point_idx, neighbors, cluster_id):
        self.labels_[point_idx] = cluster_id
        i = 0
        while i < len(neighbors):
            neighbor_idx = neighbors[i]

            if neighbor_idx not in self.visited:
                self.visited.add(neighbor_idx)
                new_neighbors = self._get_neighbors(neighbor_idx)

                if len(new_neighbors) >= self.min_samples:
                    neighbors += new_neighbors

            if self.labels_[neighbor_idx] == -1:
                self.labels_[neighbor_idx] = cluster_id

            i += 1
```

Продолжение Листинга Б.1

```
    def _get_neighbors(self, point_idx) -> list[int]:
        neighbors = []
        for other_idx in range(len(self.X)):
            if point_idx == other_idx:
                continue
        if np.linalg.norm(self.X[point_idx] - self.X[other_idx]) < self.epsilon:
                neighbors.append(other_idx)
        return neighbors

    @staticmethod
    def _distance(x, y):
        return np.linalg.norm(x - y)

    def denoise(self):
        not_noise_indexes = []
        for i in range(len(self.labels_)):
            if self.labels_[i] != -1:
                not_noise_indexes.append(i)
        return not_noise_indexes

    def rand_index(self, real_labels):
        new_label = np.max(self.labels_) + 1
        pred_labels = self.labels_
        for i in range(len(pred_labels)):
            if pred_labels[i] == -1:
                pred_labels[i] = new_label
                new_label += 1
        return rand_score(real_labels, pred_labels)

    def cluster_cohesion(self):
        result = 0
        for i in range(max(self.labels_)):
            centers = np.array([self.X[self.labels_ == i].mean(axis=0)
                                for i in range(max(self.labels_))])
            center = centers[i]
            for j in range(len(self.X)):
                if self.labels_[j] == i:
                    result += np.linalg.norm(self.X[j] - center) ** 2
        return result


def preprocessing(df: pd.DataFrame):
    df = df.loc[df.genre.isin(CLASSES)]
    df = df.drop_duplicates(subset=['artists', 'track_name']) \
           .reset_index(drop=True)
    df = df.drop(columns=df.columns[:5])

    df = df.drop(columns=['energy', 'loudness'])
    scaler = StandardScaler()
    df.iloc[:, :-1] = df.iloc[:, :-1].astype(float)
    df.iloc[:, :-1] = scaler.fit_transform(df.iloc[:, :-1])

    return df


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] | None = None):
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

      scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
        plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
```

Окончание Листинга Б.1

```
       string_columns = ['track_id', 'artists', 'album_name', 'track_name',
'track_genre']
    for column in string_columns:
        column_types[column] = str

    df = pd.read_csv('dataset.csv', dtype=column_types)
    df = df.rename(columns={'track_genre': 'genre'})
    data = preprocessing(df)

    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
     visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    dbscan = DBSCAN(data.drop(columns=['genre']), epsilon=2.1, min_samples=8)
    dbscan.fit()
    print(f"Количество кластеров: {np.max(dbscan.labels_) + 1}")
    print("Метки кластеров:", dbscan.labels_)
    print(f"Количество шумовых точек: {len(data) - len(dbscan.denoise())}")

     visualize(data.drop(columns=['genre']), dbscan.labels_, "Кластеризованные
данные")
    print(f'Значение Rand индекса: {dbscan.rand_index(track_labels)}')

    # processed_data = data.iloc[dbscan.denoise(), :]
    # processed_data.to_csv('processed.csv', encoding='utf-8', index=False)
    print(f'Плотност кластеров: {dbscan.cluster_cohesion()}')


if __name__ == '__main__':
    main()
```

Приложение В

Реализация ансамбля кластеризации

Листинг В.1 — Код файла cluster_ensemble.py

```
import typing
import warnings
import time
from functools import wraps
from collections import defaultdict

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import rand_score
from sklearn.decomposition import PCA
from scipy.cluster.hierarchy import linkage, fcluster
from scipy.cluster.hierarchy import dendrogram


from kmeans import KMeans, METRICS

def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
      print(f"Метод {func.__name__} выполнился за {execution_time:.4f} секунд")
        return result
    return wrapper


CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']
SEED = 78498

rng = np.random.default_rng(SEED)


class ClusterEnsemble:
    def __init__(self,
                 data: pd.DataFrame,
                 n_clusters: int = 6,
                 n_estimators: int = 5,
                 method: str = 'ward',
                 ):
        self.X = data.values
        self.n_clusters = n_clusters
        self.n_estimators = n_estimators
        self.weights = np.ones((self.n_estimators,))
        self.method = method
        self.labels_ = None
        self.linkage_matrix_ = None

    def _create_estimators(self):
        self.estimators: list[KMeans] = []
        for _ in range(self.n_estimators):
            metric = rng.choice(typing.get_args(METRICS))
            self.estimators.append(KMeans(self.X, metric=metric))

    @measure_time
    def fit(self, supervised=False, true_labels: list[int] | None = None):
        if supervised and true_labels is None:
            raise Exception("Не заданы метки для обучения")
        if not supervised and true_labels is not None:
            warnings.warn(Warning("Метки для обучения будут проигнорированы"))
        self._create_estimators()
```

Продолжение Листинга В.1

```
        dist_matrix = np.zeros((self.X.shape[0], self.X.shape[0]))
        accuracies = []
        for i in range(self.n_estimators):
            print(f'Количество кластеров: {self.estimators[i].n_clusters}\n'
                                        f'Максимальное количество итераций:
{self.estimators[i].max_iter}\n'
                  f'Выбранная метрика: {self.estimators[i].metric.metric}\n')
            self.estimators[i].fit()
            accuracies.append(self.estimators[i].rand_index(true_labels)
                              if supervised
                              else 1 / self.estimators[i].cluster_cohesion())
        accuracies = np.array(accuracies)
        self.weights = accuracies / np.sum(accuracies)
        print("Построение согласованной матрицы разбиений")

        for i in range(self.n_estimators):
                                        dist_matrix   +=   self.weights[i]   *
self.estimators[i].cluster_similarity_matrix()

        self.linkage_matrix_ = linkage(dist_matrix, method=self.method)
             self.labels_  =  fcluster(self.linkage_matrix_,  t=self.n_clusters,
criterion='maxclust') - 1


    def plot_dendrogram(self):
        plt.figure(figsize=(15, 10))
        dendrogram(self.linkage_matrix_)
        plt.title('Дендрограмма')
        plt.xlabel('Номер входного элемента')
        plt.ylabel('Расстояние')
        plt.show()

    def rand_index(self, real_labels):
        return rand_score(real_labels, self.labels_)

    def cluster_cohesion(self):
        result = 0
        for i in range(max(self.labels_)):
            centers = np.array([self.X[self.labels_ == i].mean(axis=0)
                                for i in range(max(self.labels_))])
            center = centers[i]
            for j in range(len(self.X)):
                if self.labels_[j] == i:
                    result += np.linalg.norm(self.X[j] - center) ** 2
        return result


def preprocessing(df: pd.DataFrame):
    df = df.loc[df.genre.isin(CLASSES)]
    df = df.drop_duplicates(subset=['artists', 'track_name']) \
           .reset_index(drop=True)
    df = df.drop(columns=df.columns[:5])

    df = df.drop(columns=['energy', 'loudness'])
    scaler = StandardScaler()
    df.iloc[:, :-1] = df.iloc[:, :-1].astype(float)
    df.iloc[:, :-1] = scaler.fit_transform(df.iloc[:, :-1])

    return df


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] | None = None):
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

      scatter  =  plt.scatter(transformed_data[:,  0],  transformed_data[:,  1],
c=labels, s=6)
    if label_names is not None:
        plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
```

Окончание Листинга В.1

```
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
       string_columns = ['track_id', 'artists', 'album_name', 'track_name',
'track_genre']
    for column in string_columns:
        column_types[column] = str

    df = pd.read_csv('dataset.csv', dtype=column_types)
    df = df.rename(columns={'track_genre': 'genre'})
    data = preprocessing(df)

    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
    # visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    ensemble = ClusterEnsemble(data.drop(columns=['genre']), n_estimators=13)
    ensemble.fit(supervised=False, true_labels=track_labels)
    print("Метки кластеров:", ensemble.labels_)

    visualize(data.drop(columns=['genre']), ensemble.labels_, "Кластеризованные
данные")
    print(f'Значение Rand индекса: {ensemble.rand_index(track_labels)}')
    print(f'Плотност кластеров: {ensemble.cluster_cohesion()}')

    ensemble.plot_dendrogram()


if __name__ == '__main__':
    main()
```

Приложение Г

Реализация алгоритма KNN

Листинг Г.1 — Код файла knn.py

```
from collections import defaultdict
from typing import Literal, Callable
from functools import wraps
import time

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from tqdm import tqdm


CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']

METRICS = Literal["euclidean", "manhattan", "chebyshev", "jaccard"]
SEED = 78498
rng = np.random.default_rng(SEED)

def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
      print(f"Метод {func.__name__} выполнился за {execution_time:.4f} секунд")
        return result
    return wrapper

class Distance:
    def __init__(self, metric: METRICS = "euclidean"):
        self.metric = metric
        metrics: dict[METRICS, Callable[[np.ndarray, np.ndarray], np.float32]]
= {
            "euclidean": self._euclid,
            "manhattan": self._manhattan,
            "chebyshev": self._chebyshev,
            "jaccard": self._jaccard,
        }
        self._func = metrics[self.metric]

    @staticmethod
    def _euclid(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.linalg.norm(x - y)

    @staticmethod
    def _manhattan(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.sum(np.abs(x - y))

    @staticmethod
    def _chebyshev(x: np.ndarray, y: np.ndarray) -> np.float32:
        return np.max(np.abs(x - y))

    @staticmethod
    def _jaccard(x: np.ndarray, y: np.ndarray) -> np.float32:
        intersection = np.sum(np.minimum(x, y))
        union = np.sum(np.maximum(x, y))
        return 1 - intersection / union if union != 0 else 0

    def __call__(self, x: np.ndarray, y: np.ndarray) -> np.float32:
        return self._func(x, y)
```

Продолжение Листинга Г.1

```
class KNNClassifier:
    def __init__(self,
                 n_neighbors: int = 5,
                 metric: METRICS = "euclidean",
                 weights: Literal["uniform", "distance"] = "uniform"):
        self.n_neighbors = n_neighbors
        self.metric = Distance(metric)
        self.weights = weights
        self.X_train = None
        self.y_train = None

    @measure_time
    def fit(self, X: np.ndarray, y: np.ndarray):
        self.X_train = X
        self.y_train = y

    @measure_time
    def predict(self, X: np.ndarray) -> np.ndarray:
        if self.X_train is None or self.y_train is None:
            raise ValueError("Model not fitted yet. Call fit() first.")

        predictions = np.empty(X.shape[0], dtype=self.y_train.dtype)

        for i, x in enumerate(tqdm(X, desc="Predicting")):
                distances = np.array([self.metric(x, x_train) for x_train in
self.X_train])

            k_nearest_indices = np.argpartition(distances, self.n_neighbors)
[:self.n_neighbors]
            k_nearest_labels = self.y_train[k_nearest_indices]

            if self.weights == "uniform":
                unique, counts = np.unique(k_nearest_labels, return_counts=True)
                predictions[i] = unique[np.argmax(counts)]
            else:
                k_nearest_distances = distances[k_nearest_indices]
                weights = 1 / (k_nearest_distances + 1e-10)
                weighted_votes = np.zeros(len(CLASSES))

                for label, weight in zip(k_nearest_labels, weights):
                    weighted_votes[label] += weight

                predictions[i] = np.argmax(weighted_votes)

        return predictions


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] | None = None):
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

    scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
        plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
    string_columns = ['genre']
    for column in string_columns:
        column_types[column] = str

    data = pd.read_csv('processed.csv', dtype=column_types)
```

Окончание Листинга Г.1

```
    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
    # visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)


    X = data.drop(columns=['genre']).values
    y = track_labels
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    knn = KNNClassifier(n_neighbors=5, metric="chebyshev", weights="distance")
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=CLASSES)
    print(f"Accuracy: {accuracy:.4f}")
    print("Classification Report:")
    print(report)

    visualize(X_test, y_test, "Истинные классы (тестовая выборка)", CLASSES)
   visualize(X_test, y_pred, "Предсказанные классы (тестовая выборка)", CLASSES)


if __name__ == '__main__':
    main()
```

Приложение Д

Реализация дерева решений

Листинг Д.1 — Код файла tree_classifie.py

```
import time
from functools import wraps
from collections import defaultdict, Counter
from pathlib import Path
from typing import List, Dict, Tuple, Optional, Union
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import math

CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']

def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
       print(f"Метод {func.__name__} выполнился за {execution_time:.4f} секунд")
        return result
    return wrapper

class DecisionNode:
    def __init__(self,
                 feature_idx: Optional[int] = None,
                 threshold: Optional[float] = None,
                 left: Optional['DecisionNode'] = None,
                 right: Optional['DecisionNode'] = None,
                 value: Optional[int] = None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf(self) -> bool:
        return self.value is not None


class DecisionTreeClassifier:
    def __init__(self,
                 max_depth: int = 5,
                 min_samples_split: int = 2,
                 criterion: str = 'gini'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.criterion = criterion
        self.root = None

    @measure_time
    def fit(self, X: np.ndarray, y: np.ndarray):
        self.n_classes = len(np.unique(y))
        self.root = self._grow_tree(X, y)

      def _grow_tree(self, X: np.ndarray, y: np.ndarray, depth: int = 0) -
> DecisionNode:
        n_samples, _ = X.shape
        n_labels = len(np.unique(y))

        if (depth >= self.max_depth
```

Продолжение Листинга Д.1

```
            or n_labels == 1
            or n_samples < self.min_samples_split):
            leaf_value = self._most_common_label(y)
            return DecisionNode(value=leaf_value)

        best_feature, best_threshold = self._best_split(X, y)

        if best_feature is None:
            return DecisionNode(value=self._most_common_label(y))

        left_idxs = X[:, best_feature] <= best_threshold
        right_idxs = X[:, best_feature] > best_threshold
        left = self._grow_tree(X[left_idxs], y[left_idxs], depth + 1)
        right = self._grow_tree(X[right_idxs], y[right_idxs], depth + 1)

        return DecisionNode(best_feature, best_threshold, left, right)

    def _best_split(self, X: np.ndarray, y: np.ndarray) -> Tuple[Optional[int],
Optional[float]]:
        best_gain = -1
        best_feature, best_threshold = None, None

        for feature_idx in range(X.shape[1]):
            thresholds = np.unique(X[:, feature_idx])
            for threshold in thresholds:
                gain = self._information_gain(X, y, feature_idx, threshold)

                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature_idx
                    best_threshold = threshold

        return best_feature, best_threshold

    def _information_gain(self, X: np.ndarray, y: np.ndarray, feature_idx: int,
threshold: float) -> float:
        if self.criterion == 'gini':
            parent_impurity = self._gini_impurity(y)
        else:
            parent_impurity = self._entropy(y)

        left_idxs = X[:, feature_idx] <= threshold
        right_idxs = X[:, feature_idx] > threshold

        if len(y[left_idxs]) == 0 or len(y[right_idxs]) == 0:
            return 0

        n = len(y)
        n_left, n_right = len(y[left_idxs]), len(y[right_idxs])

        if self.criterion == 'gini':
             child_impurity = (n_left / n) * self._gini_impurity(y[left_idxs])
+ \
                              (n_right / n) * self._gini_impurity(y[right_idxs])
        else:
            child_impurity = (n_left / n) * self._entropy(y[left_idxs]) + \
                              (n_right / n) * self._entropy(y[right_idxs])

        return parent_impurity - child_impurity

    @staticmethod
    def _gini_impurity(y: np.ndarray) -> float:
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return 1 - np.sum(probabilities ** 2)

    @staticmethod
    def _entropy(y: np.ndarray) -> float:
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return -np.sum([p * math.log(p) for p in probabilities if p > 0])
```

Окончание Листинга Д.1

```
    @staticmethod
    def _most_common_label(y: np.ndarray) -> int:
        counts = np.bincount(y)
        return np.argmax(counts)

    @measure_time
    def predict(self, X: np.ndarray) -> np.ndarray:
        return np.array([self._traverse_tree(x, self.root) for x in X])

    def _traverse_tree(self, x: np.ndarray, node: DecisionNode) -> int:
        if node.is_leaf():
            return node.value

        if x[node.feature_idx] <= node.threshold:
            return self._traverse_tree(x, node.left)
        else:
            return self._traverse_tree(x, node.right)


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] = None):
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

      scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
        plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
    string_columns = ['genre']
    for column in string_columns:
        column_types[column] = str

    data = pd.read_csv('processed.csv', dtype=column_types)

    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
     visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    X = data.drop(columns=['genre']).values
    y = track_labels
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    tree = DecisionTreeClassifier(max_depth=5, criterion='gini')
    tree.fit(X_train, y_train)

    y_pred = tree.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=CLASSES)

    print(f"Точносьб: {accuracy:.4f}")
    print("Отчёт классификации:")
    print(report)

    visualize(X_test, y_test, "Истинные классы (тестовая выборка)", CLASSES)
   visualize(X_test, y_pred, "Предсказанные классы (тестовая выборка)", CLASSES)


if __name__ == '__main__':
    main()
```

Приложение Е

Реализация бэггинга

Листинг Е.1 — Код файла bagging.py

```
import time
from functools import wraps
from collections import defaultdict
from typing import Tuple, Optional
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import math

def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
      print(f"Метод {func.__name__} выполнился за {execution_time:.4f} секунд")
        return result
    return wrapper

CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']
SEED = 78498
rng = np.random.default_rng(SEED)

class DecisionNode:
    def __init__(self,
                 feature_idx: Optional[int] = None,
                 threshold: Optional[float] = None,
                 left: Optional['DecisionNode'] = None,
                 right: Optional['DecisionNode'] = None,
                 value: Optional[int] = None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf(self) -> bool:
        return self.value is not None


class DecisionTreeClassifier:
    def __init__(self,
                 max_depth: int = 5,
                 min_samples_split: int = 2,
                 criterion: str = 'gini'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.criterion = criterion
        self.root = None

    def fit(self, X: np.ndarray, y: np.ndarray):
        self.n_classes = len(np.unique(y))
        self.root = self._grow_tree(X, y)

      def _grow_tree(self, X: np.ndarray, y: np.ndarray, depth: int = 0) -
> DecisionNode:
        n_samples, _ = X.shape
        n_labels = len(np.unique(y))
```

Продолжение Листинга Е.1

```
        if (depth >= self.max_depth
            or n_labels == 1
            or n_samples < self.min_samples_split):
            leaf_value = self._most_common_label(y)
            return DecisionNode(value=leaf_value)

        best_feature, best_threshold = self._best_split(X, y)

        if best_feature is None:
            return DecisionNode(value=self._most_common_label(y))

        left_idxs = X[:, best_feature] <= best_threshold
        right_idxs = X[:, best_feature] > best_threshold
        left = self._grow_tree(X[left_idxs], y[left_idxs], depth + 1)
        right = self._grow_tree(X[right_idxs], y[right_idxs], depth + 1)

        return DecisionNode(best_feature, best_threshold, left, right)

    def _best_split(self, X: np.ndarray, y: np.ndarray) -> Tuple[Optional[int],
Optional[float]]:
        best_gain = -1
        best_feature, best_threshold = None, None

        for feature_idx in range(X.shape[1]):
            thresholds = np.unique(X[:, feature_idx])
            for threshold in thresholds:
                gain = self._information_gain(X, y, feature_idx, threshold)

                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature_idx
                    best_threshold = threshold

        return best_feature, best_threshold

    def _information_gain(self, X: np.ndarray, y: np.ndarray, feature_idx: int,
threshold: float) -> float:
        if self.criterion == 'gini':
            parent_impurity = self._gini_impurity(y)
        else:
            parent_impurity = self._entropy(y)

        left_idxs = X[:, feature_idx] <= threshold
        right_idxs = X[:, feature_idx] > threshold

        if len(y[left_idxs]) == 0 or len(y[right_idxs]) == 0:
            return 0

        n = len(y)
        n_left, n_right = len(y[left_idxs]), len(y[right_idxs])

        if self.criterion == 'gini':
             child_impurity = (n_left / n) * self._gini_impurity(y[left_idxs])
+ \
                              (n_right / n) * self._gini_impurity(y[right_idxs])
        else:
            child_impurity = (n_left / n) * self._entropy(y[left_idxs]) + \
                              (n_right / n) * self._entropy(y[right_idxs])

        return parent_impurity - child_impurity

    @staticmethod
    def _gini_impurity(y: np.ndarray) -> float:
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return 1 - np.sum(probabilities ** 2)

    @staticmethod
    def _entropy(y: np.ndarray) -> float:
        counts = np.bincount(y)
        probabilities = counts / len(y)
```

Продолжение Листинга Е.1

```
        return -np.sum([p * math.log(p) for p in probabilities if p > 0])

    @staticmethod
    def _most_common_label(y: np.ndarray) -> int:
        counts = np.bincount(y)
        return np.argmax(counts)

    def predict(self, X: np.ndarray) -> np.ndarray:
        return np.array([self._traverse_tree(x, self.root) for x in X])

    def _traverse_tree(self, x: np.ndarray, node: DecisionNode) -> int:
        if node.is_leaf():
            return node.value

        if x[node.feature_idx] <= node.threshold:
            return self._traverse_tree(x, node.left)
        else:
            return self._traverse_tree(x, node.right)


class BaggingClassifier:
    def __init__(self,
                 n_estimators: int = 10,
                 max_samples: float = 1.0,
                 max_features: float = 1.0,
                 max_depth: int = 5):
        self.n_estimators = n_estimators
        self.max_samples = max_samples
        self.max_features = max_features
        self.max_depth = max_depth
        self.estimators = []

    @measure_time
    def fit(self, X: np.ndarray, y: np.ndarray):
        self.estimators = []
        n_samples = X.shape[0]
        n_features = X.shape[1]

        sample_size = int(n_samples * self.max_samples)
        feature_size = int(n_features * self.max_features)

        for _ in tqdm(range(self.n_estimators), desc="Обучение деревьев"):
            sample_indices = rng.choice(n_samples, sample_size, replace=True)
          feature_indices = rng.choice(n_features, feature_size, replace=False)

            X_sample = X[sample_indices][:, feature_indices]
            y_sample = y[sample_indices]

            tree = DecisionTreeClassifier(max_depth=self.max_depth)
            tree.fit(X_sample, y_sample)

            self.estimators.append((tree, feature_indices))

    @measure_time
    def predict(self, X: np.ndarray) -> np.ndarray:
        predictions = np.zeros((X.shape[0], self.n_estimators))

        for i, (tree, feature_indices) in enumerate(self.estimators):
            X_subset = X[:, feature_indices]
            predictions[:, i] = tree.predict(X_subset)

        final_predictions = np.apply_along_axis(
            lambda x: np.argmax(np.bincount(x.astype(int))),
            axis=1,
            arr=predictions
        )

        return final_predictions

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        proba = np.zeros((X.shape[0], len(np.unique(self.y_))))
```

Окончание Листинга Е.1

```
        for tree, feature_indices in self.estimators:
            X_subset = X[:, feature_indices]
            preds = tree.predict(X_subset)

            for i, pred in enumerate(preds):
                proba[i, pred] += 1

        proba /= self.n_estimators
        return proba



def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] = None):
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

      scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
          plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    column_types = defaultdict(np.float32)
    string_columns = ['genre']
    for column in string_columns:
        column_types[column] = str

    data = pd.read_csv('processed.csv', dtype=column_types)

    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
     # visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    X = data.drop(columns=['genre']).values
    y = track_labels
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    bagging = BaggingClassifier(
        n_estimators=50,
        max_depth=5
    )
    bagging.fit(X_train, y_train)

    y_pred = bagging.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=CLASSES)

    print(f"Точность: {accuracy:.4f}")
    print("Отчёт классификации:")
    print(report)

    visualize(X_test, y_test, "Истинные классы (тестовая выборка)", CLASSES)
   visualize(X_test, y_pred, "Предсказанные классы (тестовая выборка)", CLASSES)


if __name__ == '__main__':
    main()
```

Приложение Ж

Реализация случайного леса

Листинг Ж.1 — Код файла random_forest.py

```
import time
from functools import wraps
from collections import defaultdict
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier


CLASSES = ['classical', 'rock', 'hip-hop', 'pop', 'electronic', 'world-music']
SEED = 78498
rng = np.random.default_rng(SEED)


def measure_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
      print(f"Метод {func.__name__} выполнился за {execution_time:.4f} секунд")
        return result
    return wrapper

@measure_time
def fit(forest, X_train, Y_train):
    forest.fit(X_train, Y_train)

@measure_time
def predict(forest, X_test):
    return forest.predict(X_test)


def visualize(df: pd.DataFrame, labels: list[int], name: str, label_names:
list[str] = None):
    """Визуализация данных (без изменений)"""
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(df)

      scatter = plt.scatter(transformed_data[:, 0], transformed_data[:, 1],
c=labels, s=6)
    if label_names is not None:
        plt.legend(handles=scatter.legend_elements()[0], labels=label_names,
title='Классы')
    else:
        plt.legend(*scatter.legend_elements(), title='Кластеры')
    plt.title(name)
    plt.show()


def main():
    """Основная функция с использованием бэггинга"""
    column_types = defaultdict(np.float32)
    string_columns = ['genre']
    for column in string_columns:
        column_types[column] = str

    data = pd.read_csv('processed.csv', dtype=column_types)

    # Кодирование меток классов
```

Окончание Листинга Ж.1

```
    encoder = LabelEncoder()
    track_labels = encoder.fit_transform(data.genre)
    # visualize(data.drop(columns=['genre']), track_labels, "Исходные данные",
CLASSES)

    X = data.drop(columns=['genre']).values
    y = track_labels
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    # Обучение бэггинга
    forest = RandomForestClassifier(
        n_estimators=100,
        max_depth=10,
    )
    fit(forest, X_train, y_train)

    y_pred = predict(forest, X_test)
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=CLASSES)

    print(f"Accuracy: {accuracy:.4f}")
    print("Classification Report:")
    print(report)

    visualize(X_test, y_test, "Истинные классы (тестовая выборка)", CLASSES)
   visualize(X_test, y_pred, "Предсказанные классы (тестовая выборка)", CLASSES)


if __name__ == '__main__':
    main()
```